Noter om R

Morten Larsen og Peter Sestoft

Institut for Grundvidenskab og Miljø
Det Biovidenskabelige Fakultet
Københavns Universitet

Version 6.0 af 2010-11-25.
Den seneste udgave af disse noter kan hentes fra denne hjemmeside:
http://matdat.life.ku.dk/R-noter

Indhold

Tak til Bo Martin Bibby, Claus Ekstrøm, Henrik Laurberg Pedersen, Jacob Engelbrecht, Mogens Flensted-Jensen og Thomas Vils Pedersen for kommentarer og forslag, og selvfølgelig tak til de mange der har bidraget til udviklingen af R-systemet.

Forord

Programsystemet R er velegnet til matematiske og statistiske beregninger, graftegning, og behandling af forsøgsdata. Man kan lovligt og gratis installere en kopi af R på din egen pc; systemet er open source software. Beskrivelsen i starten af disse noter af den konkrete brug af programmet er baseret på udgaverne til Windows og MacOS X, men R fås også til Linux. De generelle principper og selve sproget, man skriver beregninger i, er det samme uanset operativsystem.

Disse "Noter om R" er oprindelig skrevet til kurset *Matematik og Databehandling* på Det Biovidenskabelige Fakultet (LIFE) på Københavns Universitet, hvor de udgør den ene del af et kompendium hvori også indgår "Noter om regneark". Henvendelser fra studerende og undervisere på LIFE har vist et behov for "Noter om R" også uden for kurset; et behov der forhåbentlig bliver imødekommet af denne elektronisk publicerede version.

Den forhåndenværende udgave, uden regneark og uden visse dele meget specifikke for *Matematik og Databehandling*, burde være egnet som en selvstændig, generel introduktion til R, men dog (næsten) uden statistik! Noterne fokuserer på *sproget* implementeret af R og gennemgår funktioner til at håndtere og grafisk fremstille (forsøgs)data, men altså stort set ikke til at analysere dem. Derimod gennemgås elementer af programmering med R (`for`, `if`, `while` og programmering af funktioner).

Det bedste udbytte af noterne fås hvis man selv eksperimenterer med de præsenterede funktioner i R under læsningen. Alle eksempelindtastninger fra noterne samt de benyttede eksempelfiler findes her:

`http://matdat.life.ku.dk/R-noter/eksempler`

Opgaver (og facitliste) er bibeholdt fra versionen der anvendes i *Matematik og Databehandling*. Faktisk er der endnu flere opgaver i denne udgave end i kursusudgaven, fordi også opgaver, der har vist sig ikke at være tid til i kurset, er med. På hjemmesiden nævnt ovenfor findes også en fil med R-scripts, der er de egentlige løsninger til opgaverne.

En del eksempler og opgaver er baseret på anvendelseseksempler fra *Noter om matematik* af Henrik Laurberg Pedersen og Thomas Vils Pedersen – også skrevet til kurset *Matematik og databehandling* – men det er på ingen måde ikke nødvendigt at have disse noter ved hånden for at læse "Noter om R".

Noterne er tænkt egnet til opslag når man først har været hovedteksten igennem. Appendiks G er tænkt som "referencemanual" til de R-funktioner, der gennemgås i noterne (samt nogle få yderligere som er for nyttige til at jeg ville være bekendt at udelade dem). I indekset bagest kan man også slå de enkelte funktioner op alfabetisk.

Frederiksberg, november 2010
Morten Larsen

1 Brug af R

Den følgende gennemgang af den basale brug af R i dette afsnit tager udgangspunkt i brugergrænsefladen som den ser ud under Microsoft Windows eller MacOS X. Der er store forskelle i de to brugergrænseflader, men princippet med de forskellige vinduer (console, editor og grafer) samt det hensigtsmæssige i hovedsagelig at arbejde fra editoren (afsnit 1.4) er det samme.

Brug specifik for den ene eller den anden brugergrænseflade er markeret med [Windows] hhv. [MacOS] foran.

1.1 Vinduerne i R

Når man arbejder med R har man typisk flere vinduer. Under Windows er vinduerne samlet i et stort, ydre vindue kaldet RGui,[1] som det ses i figur 1. Under MacOS er vinduerne direkte på skrivebordet og menulinjen for oven skifter til R's menu når et af vinduerne er aktive, se figur 2.

- RGui er det store ydre vindue (kun Windows). Menuen i RGui afhænger af, hvilket indre vindue der ligger øverst, dvs. hvilket vindue der er aktivt. I disse noter henviser menuvalg, når intet andet er angivet, til menuen når R Console er det aktive vindue.
- R Console er det vindue hvor man kan skrive regneudtryk der skal udregnes; se afsnit 1.2. Det kan betragtes som R's "hovedvindue".
- R Graphics er et vindue der viser det seneste plot; se afsnit 1.3. Under MacOS hedder grafikvinduet "Quartz".
- R Editor er et vindue hvor man kan redigere R kode så som funktionsdefinitioner, se afsnit 1.4.
- R Help vinduer (ikke vist) dukker op hvis man har brugt hjælpefunktionen i R; se afsnit 1.5.

1.2 R Console vinduet

I R Console vinduet kan du skrive regneudtryk og definitioner. For at udregne `2 + 2` skal du skrive udtrykket efter prompten `>` og taste ⏎:

```
> 2 + 2
[1] 4
```

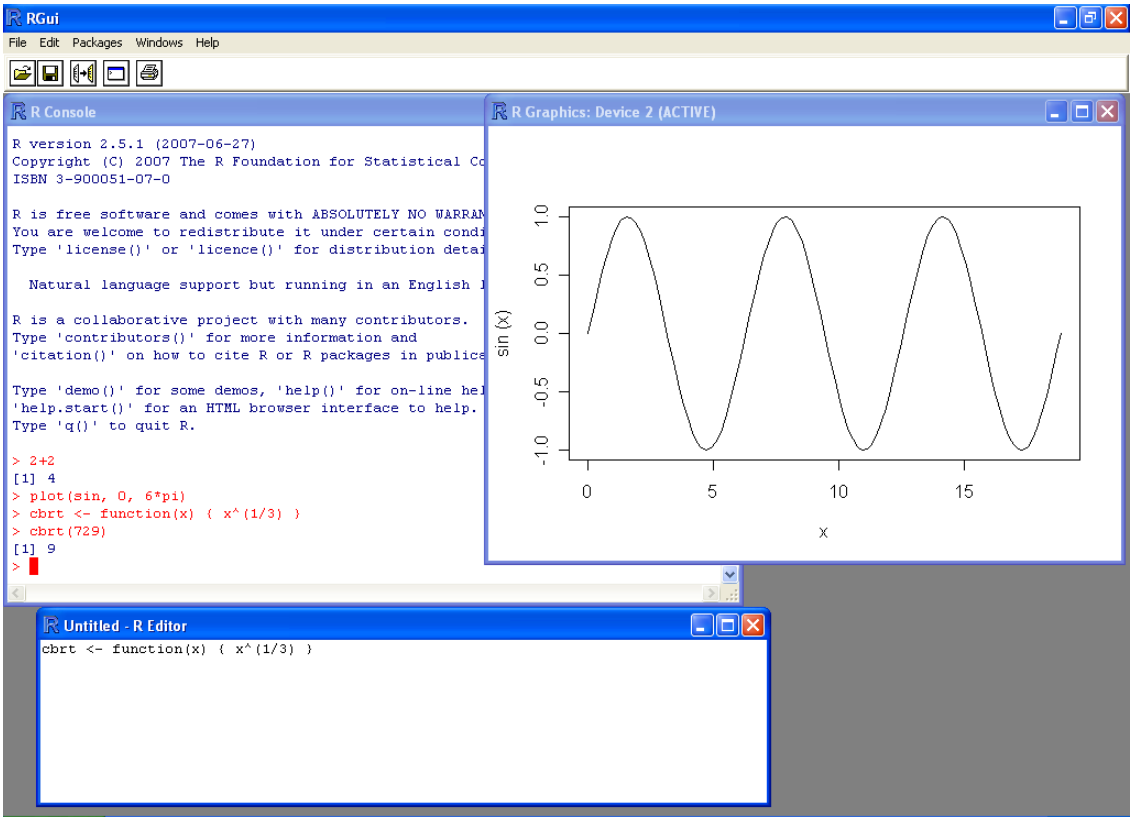
Svaret fra R kommer på den næste linje. Resultatet af beregningen er naturligvis `4`, og klammen `[1]` betyder at `4` er det første tal i resultatet. Senere skal vi se at et svar fra R kan bestå af flere tal.

Indtastninger i R Console og resultater fra R vil i disse noter blive vist som ovenfor. Bemærk, at den tekst man indtaster selv, som fx `2 + 2`, er vist med en federe skrift end den tekst som R skriver, som fx prompten `>` og resultatet `[1] 4`. Farverne (rød for indtastninger og blå for resultater) følger konventionen fra Windows-udgaven af R Console — de er anderledes i MacOS X udgaven.

Du kan bladre i de seneste indtastningslinjer ved hjælp af pil-op og pil-ned tasterne, rette i et udtryk og igen taste ⏎ for at udregne det. Det er ikke smart at skrive lange definitioner direkte i R Console; brug i stedet R Editor som forklaret i afsnit 1.4.

Hvis du kommer til at taste ⏎ før udtrykket er færdigt vil R lave en `+` prompt på næste linje. Det angiver at R venter på resten af udtrykket:

[1] Det er muligt at slå det store RGui vindue fra og have sine vinduer direkte på skrivebordet også under Windows, men disse noter går ud fra at man benytter standardopsætningen.



Figur 1: Windows RGui med et R Console vindue, et R Graphics vindue og et R Editor vindue.

```
> 5 -
+
```

Her ville vi have udregnet `5 - 3` men kom til at taste ↵ allerede efter minus. Da vi ikke har tastet et afsluttet udtryk kommer R med + prompten. Så kan vi indtaste det manglende tretal og taste ↵, hvorefter R kommer med resultatet:

```
> 5 -
+ 3
[1] 2
```

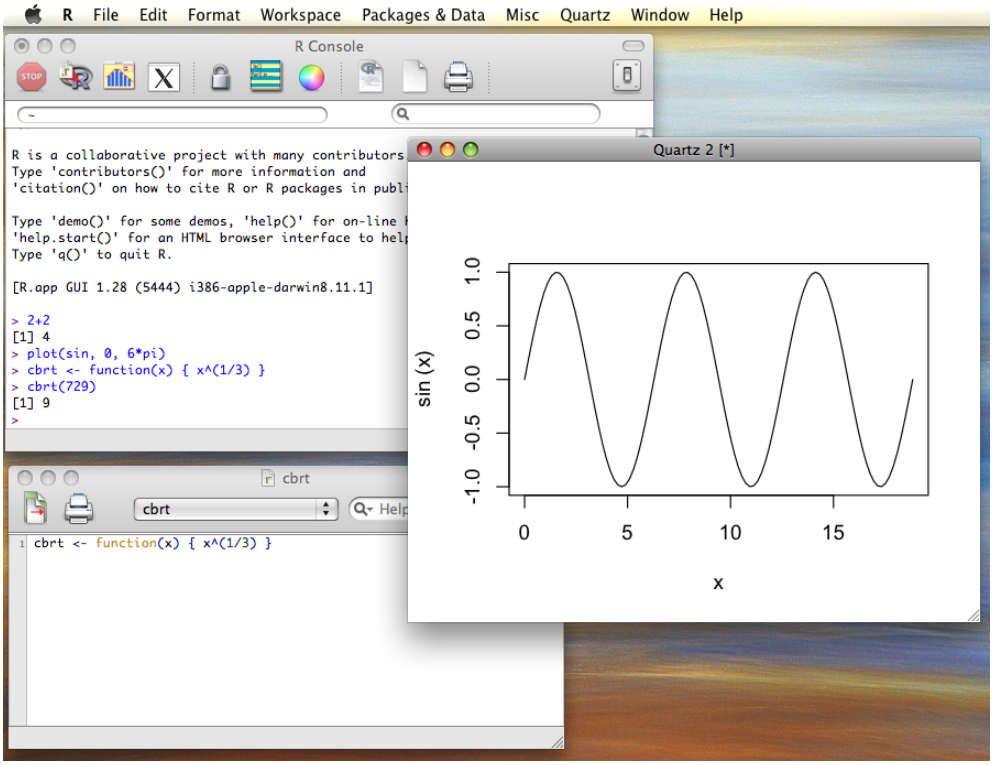
I disse noter vil enkelte lange indtastninger være delt over flere linjer på denne måde.

I eksemplet `5 - 3` var det nemt at gennemskue hvad det var vi manglede at indtaste og fejlen kunne udbedres ved blot at indtaste det manglende. Nogle gange kan man i et langt indviklet udtryk komme til at glemme en slutparentes eller noget andet og så få en uventet + prompt. Desværre kan man ikke fra + prompten gå direkte op på den foregående linje og rette fejlen. I disse tilfælde må man afbryde indtastningen med Esc og så kalde den fejlagtige linje frem igen med pil-op tasten og rette i den. Under MacOS X skal Esc efterfølges af ↵; i en Linux terminal skal man taste Ctrl+C i stedet for Esc.

1.3 R Graphics vinduet

Du kan plotte funktioner og data ved at skrive kommandoer i R Console; den resulterende graf vises i R Graphics vinduet (som hedder Quartz under MacOS X). For eksempel kan du plotte $\sin(x)$ for x fra 0 til 6π ved at skrive i R Console:

```
> plot(sin, 0, 6*pi)
```



Figur 2: R i MacOS X udgaven med et R Console vindue, et Quartz vindue og et editor vindue.

Under **Windows** vil R Graphics vinduet normalt kun gemme og vise det seneste plot. Du kan få vinduet til at huske alle plot ved først at vælge History || Recording mens R Graphics vinduet er aktivt. Du kan så bladre frem og tilbage gennem dine plots med tasterne PageUp og PageDown.

Under **MacOS X** vil Quartz vinduet automatisk gemme alle plots. Man kan bladre frem og tilbage fra Quartz menuen eller med ⌘+← og ⌘+→. Hvis man ønsker at slette alle undtagen det aktuelle plot kan man taste ⇧+⌘+L.

Du kan under **Windows** gemme det plot du ser ved at højreklikke på det og vælge Copy as metafile og derefter indsætte i Word, OpenOffice, eller PowerPoint; se afsnit 6.1.

Under **MacOS X** Kan du kopiere til udklipsholderen med ⌘+C. Udklippet er i PDF format hvilket ikke nødvendigvis kan sættes direkte ind i dit tekstbehandlingsprogram; se afsnit 6.1.

1.4 R Editor vinduet

Når du arbejder på en opgaveløsning skal du skrive definitionerne i R Editor, så du kan rette, udføre og gemme dem. Du kan nemlig ikke gemme indtastningerne du laver direkte i R Console.

- Du åbner en ny, tom script-fil i R Editor [Windows] ved at vælge File || New script, [MacOS] ved at vælge File || New Document (⌘+N) eller klikke på ikonen med det blanke dokument.

- Skriv dine definitioner i R Editor. I eksemplet vist i figur 1/figur 2 er der i R Editor defineret en funktion **`cbrt`** til at udregne kubikroden af **`x`**:

```
cbrt <- function(x) { x^(1/3) }
```

  I disse noter vises indtastninger i R Editor som ovenfor.

- Du kan under **Windows** udføre definitionerne ved at markere dem, højreklikke, og vælge Run line or selection; så bliver de kopieret over i R Console og udført. I **MacOS X** udgaven af R skal man markere og vælge Edit || Execute.

  I R Console kan man så bruge de definerede funktioner i regneudtryk, fx udregne `cbrt(729)`.

- I stedet for at højreklikke for at udføre markeret tekst fra R Editor kan man [Windows] taste Ctrl+R eller [MacOS] taste ⌘+⏎.

  Det er under Windows faktisk oftest nemmest i R Editor slet ikke at markere de linjer der skal udføres, men simpelthen flytte markøren til den første af linjerne og så taste Ctrl+R på hver linje. Markøren rykker automatisk en linje ned hver gang du taster Ctrl+R. Hermed får du udført dine definitioner i R Console én efter én og det er nemt at finde ud af hvor der eventuelt opstår fejl. Den tilsvarende funktion findes desværre ikke i MacOS X udgaven.

- Du kan rette dine definitioner i R Editor og udføre dem så ofte du vil.

- **Husk** at gemme indholdet af R Editor med File || Save eller gemme-ikonen (eller [Windows] Ctrl+S / [MacOS] ⌘+S) (editor vinduet skal være det aktive vindue). Vælg et passende filnavn, fx `Dat-A-1.R`. Det er vigtigt at angive filnavnsendelse `.R`, som er standard for R-script filer.

- En gemt R-fil kan senere åbnes ved at vælge [Windows] File || Open script / [MacOS] File || Open Document.

1.4.1 Fejlfinding

Når du arbejder med R vil du typisk have et større R-script på flere linjer som du kører ved at kopiere det fra R Editor over i R Console med Ctrl+R / ⌘+⏎, retter lidt til, kører igen, retter igen, osv. Hvis det for eksempel er en graf du sidder og opbygger er det let at komme til at fokusere udelukkende på grafvinduet og glemme at læse hvad der sker i R Console vinduet når du kører scriptet. Du skal imidlertid huske også at kigge i R Console for at se om der eventuelt skulle være nogle fejlmeddelelser – selvfølgelig især hvis du ikke rigtig kan få det til at virke.

Foretag fejlfinding i et større script på en struktureret og systematisk måde som følger:

- Kopiér scriptet over én linje ad gangen og tjek for hver linje om den er udført korrekt. Tjek om der er meddelelser i R Console. Hvis du er ved at opbygge en graf så tjek også i grafvinduet om linjen har den forventede effekt.

- Hold øje med prompten i R Console: Der skal stå et > i begyndelsen af linjen - hvis der står et + er den foregående linje ikke rigtig afsluttet. Hvis der pga. en fejl ikke står > skal du taste Esc for at få den rigtige prompt inden du kopierer mere over i R Console (Esc efterfulgt af ⏎ i MacOS X udgaven).

- Omvendt, hvis du med vilje deler et udtryk over flere linjer skal der selvfølgelig stå et + i starten af hver linje indtil udtrykket er helt afsluttet.

- Start fra begyndelsen af scriptet — fejl langt nede i scriptet er måske følgefejl fra en tidligere linje, så ret altid fejl i den rækkefølge de opstår når scriptet køres linje for linje.

- Når du har rettet en fejl: Start forfra med kørslen.

Se også appendiks B for en forklaring på de mest almindelige fejlmeddelelser.

1.5 R Help vinduer

Hjælp til brugen af R fås under menupunktet Help. Man kan under Windows få hjælp til en bestemt funktion ved at vælge Help || R functions og skrive funktionsnavnet, fx `log`. Der dukker så et R Help vindue op med en beskrivelse af funktionen og nogle eksempler. I MacOs X udgaven er denne søgefunktion indbygget i selve R Help vinduet, som man aktiverer med Help || R Help (⌘+?). Forklaringerne i hjælpen kan være ret uforståelige. Det vil for funktioner, der omtales i disse noter, generelt være langt bedre at slå dem op her, fx ved hjælp af det omfattende alfabetiske indeks sidst i noterne.

Hvis man ikke kan huske navnet på en funktion eller en regneoperator kan man i Windows udgaven af R være heldig at finde den med Help || Search help eller Help || Apropos (i MacOS X udgaven er dette bygget ind i den almindelige søgning). Ofte finder man dog blot et antal meget specielle statistiske funktioner på denne måde.

Menuen Help giver også adgang til en kort vejledning til det aktive vindue og til R's manualer.

2 Regneudtryk og indbyggede funktioner

Der er mange indbyggede regneoperatorer og funktioner i R, for det meste med de velkendte navne, men notationen i R er ofte lidt anderledes end i matematik. Funktioner og operatorer kan bruges i regneudtryk:

Regneudtryk i R	Resultat	Matematik	Betydning
`5 + 7 * 8`	`61`	$5+7\cdot 8$	Fem plus syv gange otte
`sqrt(10)`	`3.162278`	$\sqrt{10}$	Kvadratroden af 10
`10^3`	`1000`	10^3	10 opløftet i tredje
`10**3`	`1000`	10^3	10 opløftet i tredje, alternativ notation
`log10(17)`	`1.230449`	$\log(17)$	Titalslogaritme af 17
`exp(3)`	`20.08554`	e^3	Eksponentialfunktionen af 3
`log(17)`	`2.833213`	$\ln(17)$	Naturlig logaritme af 17
`sin(3/2*pi)`	`-1`	$\sin(\frac{3}{2}\pi)$	Sinus af $\frac{3}{2}\pi$

For eksempel kan den tredje rod af 17 beregnes som $\sqrt[3]{17} = 17^{1/3}$ hvad der i R skrives sådan her:

```
> 17^(1/3)
[1] 2.571282
```

Bemærk, at hat-symbolet (`^`) er en såkaldt død tast på de fleste computere, så man skal taste [^] efterfulgt af [mellemrum] for at få tegnet frem.

Bemærk også, at den implicitte multiplikation fra normal matematiknotation ikke gælder i R. For fx at beregne $2x$ skal man i R skrive `2*x`.

2.1 Visning af tal, og intern regnenøjagtighed

Som det fremgår af eksemplet ovenfor viser R normalt kun 7 betydende cifre i resultater. Dette tal kan fx sættes op til 12 på denne måde:

```
> options(digits=12)
> 17^(1/3)
[1] 2.57128159066
```

Der er ikke nogen mening i at bede R om at vise mere end 16 betydende cifre, for moderne computere regner kun med 15–16 betydende cifre internt. Det gør pc'er og R også, uanset hvor mange eller få cifre der vises i resultaterne. Visningen af resultater sættes tilbage til det normale sådan her:

```
> options(digits=7)
```

2.2 Resultater der ikke er tal

Nogle regneudtryk har ikke nogen fornuftig talværdi: hvad skulle resultatet af `1/0` eller `log(0)` eller `0/0` være? Disse udtryk giver nogle specielle resultater, nemlig `Inf`, der betyder uendelig; og `-Inf`, der betyder minus uendelig; og `NaN`, der betyder "Not a Number". Hvis `NaN` dukker op i resultatet af en beregning, så er der som regel gået noget galt og man må hellere tjekke sine regneudtryk.

Derimod betegner den specielle værdi `NA`, der betyder "not available" eller "not applicable", en manglende observation i et datasæt, fx en manglende aflæsning fra et forsøg.

Funktion i R	Matematik	Betydning
`x + y`	$x+y$	`x` plus `y`
`x - y`	$x-y$	`x` minus `y`
`x * y`	$x \cdot y$	`x` gange `y`
`x / y`	x/y	`x` divideret med `y`
`x %/% y`		`x` heltalsdivideret med `y`; fx vil `7%/%3` give `2`
`x %% y`	$x \bmod y$	`x` modulo `y`, rest ved heltalsdivision; fx vil `7%%3` give `1`
`x ^ y`	x^y	`x` opløftet i `y`
`x ** y`	x^y	`x` opløftet i `y`, alternativ notation
`abs(x)`	$\lvert x \rvert$	numerisk (absolut) værdi af `x`
`sign(x)`		fortegn af `x` (-1, 0 eller 1)
`sqrt(x)`	$\sqrt{x}$	kvadratrod af `x`
`log(x)`	$\ln(x)$	naturlig logaritme af `x`
`log10(x)`	$\log(x)$	titalslogaritme af `x`
`exp(x)`	e^x	eksponentialfunktionen af `x`
`sin(x)`	$\sin(x)$	sinus til `x` radianer
`cos(x)`	$\cos(x)$	cosinus til `x` radianer
`tan(x)`	$\tan(x)$	tangens til `x` radianer
`asin(x)`	$\sin^{-1}(x)$	arcus sinus til `x`
`acos(x)`	$\cos^{-1}(x)$	arcus cosinus til `x`
`atan(x)`	$\tan^{-1}(x)$	arcus tangens til `x`
`floor(x)`	$\lfloor x \rfloor$	`x` rundet *ned* til nærmeste heltal
`ceiling(x)`	$\lceil x \rceil$	`x` rundet *op* til nærmeste heltal
`round(x)`	$[x]$	`x` afrundet til nærmeste heltal; halve rundes til lige tal
`pi`	π	enhedscirklens areal $\pi = 3.14159\ldots$

Figur 3: Numeriske operatorer, funktioner og konstanter i R.

3 Variable

Man kan med en *tildeling* binde resultatet af et regneudtryk til en variabel, for eksempel **z**. Pilen (**<-**) betyder at udtrykket på højresiden udregnes og værdien gemmes i variablen **z**:

```
> z <- 17^(1/3)
```

Resultatet af beregningen bliver ikke vist, men lagret i variablen **z**. Man kan få oplyst en variabels værdi blot ved at skrive dens navn:

```
> z
[1] 2.571282
```

Variable kan bruges i efterfølgende beregninger:

```
> z^6 - z^3
[1] 272
```

Et variabelnavn kan indeholde punktummer, så **antal.planter** er et lovligt variabelnavn. I R anses store og små bogstaver for at være forskellige, så **z** og **Z** er to forskellige variable. Undgå at bruge navnene **c**, **F**, **t** og **T**, da de allerede bruges af R til forskellige formål.

Værdien af en variabel kan ændres ved en ny tildeling:

```
> z <- 5
> z
[1] 5
> z <- z + 3
> z
[1] 8
```

Det er muligt med funktionen **ls** at få vist en liste over hvilke variable, man har defineret:

```
> ls()
[1] "z"
```

Endelig kan man en sjælden gang have brug for helt at fjerne en variabel igen. Dette gøres med funktionen **rm**:

```
> rm(z)
> z
Error: object "z" not found
```

4 Funktioner

Man kan nemt definere sine egne funktioner i R. For eksempel kan man definere en funktion `cbrt(x)` til at finde tredje rod ("cubic root") af et vilkårligt positivt tal `x` sådan her:

```
cbrt <- function(x) { x^(1/3) }
```

Variablen `x` kaldes en *formel parameter* for funktionen `cbrt`, og krølle-parentesen[2] `{ x^(1/3) }` er funktionens *krop*. Udtrykket `x^(1/3)` beregner $\sqrt[3]{x}$. Den nye funktion bruges som alle andre funktioner ved at skrive funktionsnavnet efterfulgt af en argumentparentes:

```
> cbrt(1000)
[1] 10
```

Værdien af dette udtryk udregnes ved at sætte `x` lig 1000 og så udregne funktionskroppen `x^(1/3)`, hvilket giver 10.

Funktioner med flere formelle parametre defineres på samme måde. Her er en funktion `root(x, n)` til at beregne den `n`'te rod af `x`:

```
root <- function(x, n) { x^(1/n) }
```

Funktioner med flere parametre kaldes som man skulle forvente ved at skrive argumentudtrykkene i parentes efter funktionsnavnet, adskilt med komma:

```
> root(625, 4)
[1] 5
```

Som et yderligere eksempel kan den såkaldte logistiske funktion:

$$\text{logis}(t) = \frac{K}{1 + ae^{-rt}}$$

defineres således i R:

```
logis <- function(t) { K / (1 + a*exp(-r*t)) }
```

Bemærk at der ikke er givet nogle konkrete værdier for `a`, `r` og `K`. Man kan derfor ikke bruge funktionen til noget før `a`, `r` og `K` er blevet defineret:

```
> logis(5)
Error in logis(5) : Object "K" not found
```

Når `a`, `r` og `K` er blevet defineret kan man kalde funktionen for at beregne fx `logis(5)`, eller plotte funktionen. Hvis man ændrer `a`, `r` eller `K`, så ændres funktionen:

```
> a <- 50
> r <- 1
> K <- 1
> logis(5)
[1] 0.7480006
> K <- 10
> logis(5)
[1] 7.480006
```

[2] De krøllede parenteser fårs på et PC-tastatur med AltGr+7 og AltGr+0. På en Mac fås de med ⌥+⇧+8 og ⌥+⇧+9.

5 Funktionsplot

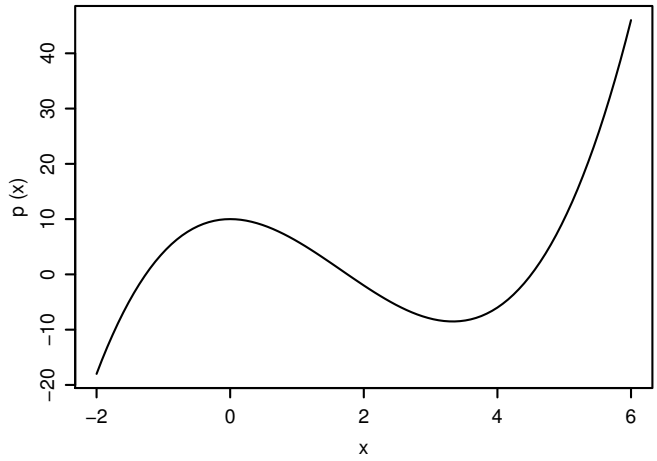
Man kan plotte indbyggede funktioner såsom $\sin(x)$ for x fra 0 til 6π ved brug af funktionen `plot`:

```
plot(sin, 0, 6*pi)
```

For at plotte mere indviklede udtryk, for eksempel polynomiet $p(x) = 10 - 5x^2 + x^3$ for x løbende fra -2 til 6, kan man definere en funktion `p` med en parameter `x` og med polynomiet som funktionskrop:

```
p <- function(x) { 10 - 5*x^2 +  x^3 }
plot(p, -2, 6)
```

Det resulterende plot ses i figur 4.



Figur 4: Plot af polynomiet $p(x) = 10 - 5x^2 + x^3$ fremstillet med `plot(p, -2, 6)`.

5.1 Tilføjelse af funktionsgrafer til plot

Man kan tilføje flere funktionsgrafer til et eksisterende plot ved at bruge `add=TRUE` i `plot`-kaldet. Med tilføjelse af følgende to linjer til de to linjer ovenfor, der gav plottet i figur 4, får vi plottet i figur 5:

```
q <- function(x) { 7*x - x^2 }
plot(q, -2, 6, add=TRUE)
```

Man kan tilføje lige så mange funktionsgrafer, man ønsker.

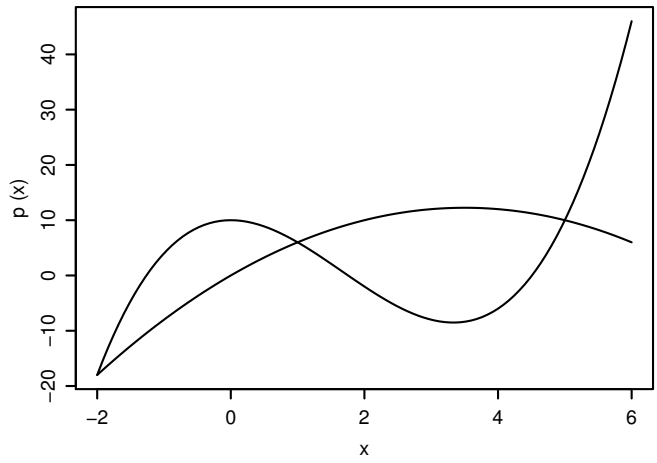
5.2 Aksegrænser i plot

Når man tilføjer en ny funktionsgraf til et eksisterende plot, bevares alle andre egenskaber ved plottet. Især bliver y-aksens grænser ikke tilpasset til den nye funktion, så man risikerer at en stor del af grafen falder uden for plottet.

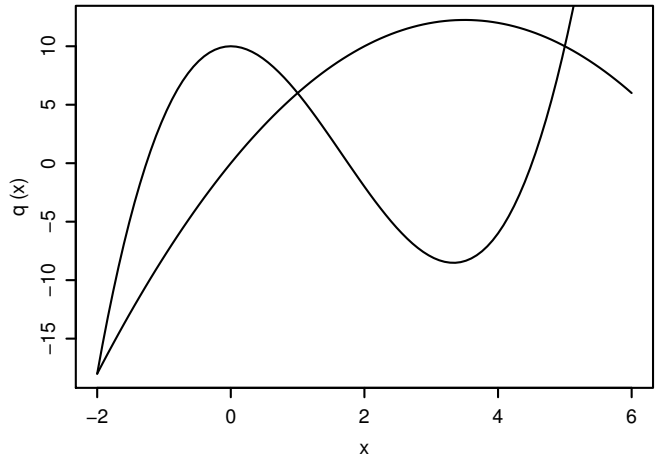
For eksempel, hvis man plotter de to funktioner `p` og `q` for $x \in [-2, 8]$ og plotter `q` først, så bliver y-aksen så kort at meget af grafen for `p` falder uden for plottet, jævnfør figur 6:

```
plot(q, -2, 6)
plot(p, -2, 6, add=TRUE)
```

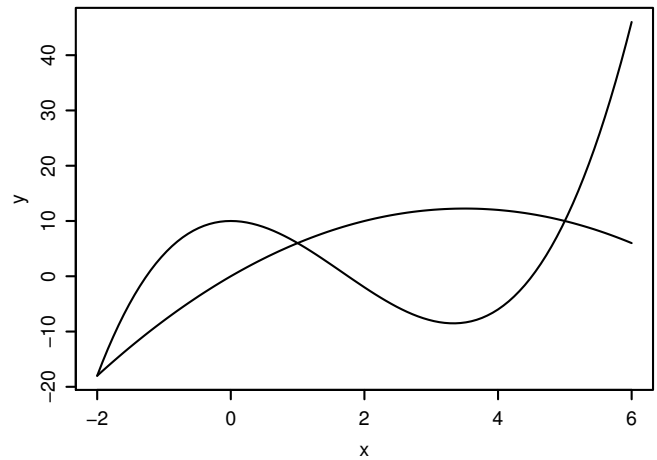
I denne situation må man finde (fx gætte) største og mindste y-værdi for de to funktioner, og sætte passende aksegrænser for y-aksen allerede når den første funktion tegnes. Det gøres med parameteren `ylim`. For at få y-aksen til at gå fra cirka -16 til cirka 100 kan man gøre sådan her:



Figur 5: Som figur 4, men med `plot(q, -2, 6, add=TRUE)` tilføjet.



Figur 6: Som figur 5, men med `q` tegnet før `p` og y-aksen derfor for kort.



Figur 7: Som figur 5, men med `ylab="y"`.

```
plot(q, -2, 6, ylim=c(-16, 100))
plot(p, -2, 6, add=TRUE)
```

Tilsvarende kan man sætte aksegrænser for x-aksen med parameteren `xlim`. Notationen `c(-16, 100)` binder de to tal sammen til en *vektor*, hvilket vi skal se flere eksempler på andre steder hvor vi skal angive en parameterværdi der består af mere end ét tal. Meget mere om vektorer følger i afsnit 14.

5.3 Aksetitler og overskrifter i plot

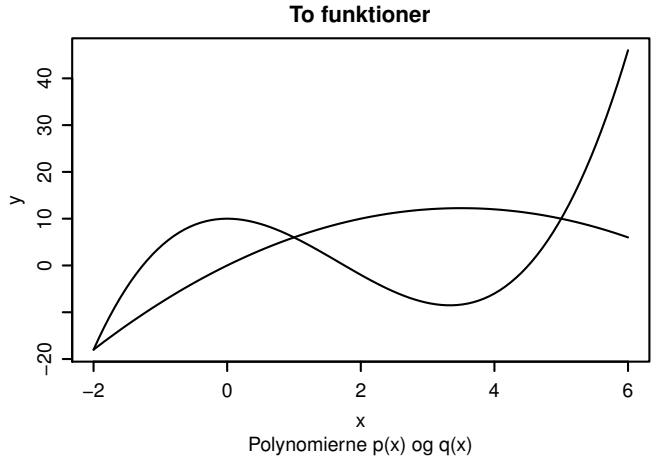
Som ses af det foregående sætter `plot` (uden `add=TRUE`) automatisk titler på akserne. Titlen på x-aksen bliver altid "x" og på y-aksen sættes den efter funktionsnavnet, fx "p(x)" hvis funktionen hedder `p`. I tilfælde som plottet i figur 5, hvor der jo er mere end én funktion, giver dette dårlig mening. Man har heldigvis mulighed for med parametre til `plot` at angive alternative aksetekster. Når man fremstiller et plot med mere end én graf, skal man sætte aksetekserne i første kald af `plot`, fx som følger:

```
plot(p, -2, 6, ylab="y")
plot(q, -2, 6, add=TRUE)
```

Her angiver parameteren `ylab` titlen på y-aksen (dvs. andenaksen). Resultatet ses i figur 7. Tilsvarende fidnes en parameter `xlab`, som ændrer titlen på x-aksen. Hvis man ikke ønsker nogen aksetitel må man angive den tilsvarende parameter med en tom tekst, fx `xlab=""`.

Plottitel (overskrift) kan sættes med parameteren `main` i det første kald af `plot`. Tilsvarende findes en parameter `sub`, der giver en titel *under* plottet. Så for at få nogle titler på plottet fra figur 7 kan man skrive:

```
plot(p, -2, 6, ylab="y", main="To funktioner",
     sub="Polynomierne p(x) og q(x)")
plot(q, -2, 6, add=TRUE)
```



Figur 8: Som figur 7, med titler tilføjet.

Resultatet ses i figur 8.

Det kan være mere bekvemt at tilføje titler til et plot, efter at det er tegnet. Hertil bruger man funktionen `title`, der tager de samme parametre som `plot`. Parameteren (`main` sætter tekst for oven, `sub` sætter tekst for neden, `xlab` sætter tekst (aksetitel) langs x-aksen, og `ylab` sætter tekst langs y-aksen. For eksempel er følgende en alternativ måde, hvorpå man kan fremstille plottet i figur 8:

```
plot(p, -2, 6, ylab="")
plot(q, -2, 6, add=TRUE)
title(main="To funktioner", sub="Polynomierne p(x) og q(x)")
title(ylab="y")
```

OBS: Da `plot` som standard ville tilføje "p(x)" som aksetitel på y-aksen er man nødt til at angive `ylab=""` i første kald af `plot` ovenfor, for ellers ville den aksetitel, der tilføjes med `title`, blot blive skrevet oven i den eksisterende. Bemærk, at de to kald af `title` ovenfor lige så godt kunne have været samlet til ét kald, der satte alle tre titler, eller være delt ud i tre kald, der satte hver sin titel.

5.4 Farver i plot

Når man tegner flere funktionskurver i samme plot er det nyttigt at give kurverne hver sin farve. Det kan man gøre med parameteren `col`:

```
plot(p, -2, 6, col="blue")
plot(q, -2, 6, add=TRUE, col="red")
```

Figur 9 er en oversigt over nogle få af standardfarverne i R.

Figur 9: Nogle standardfarver i R.

Tast **`colors()`** i R Console for at få en liste af de mere end 600 andre navngivne farver.[3] Mange flere farver kan fremstilles med funktionen `rgb` der omtales i afsnit 21.1.

Farven på akser, tekster, signaturforklaringer og titler i et plot kan styres på lignende vis; fx kan parametrene `col.main`, `col.sub` og `col.lab` gives til enten `plot` eller `title` for at styre farven på de forskelloge titler.

5.5 Linjetype og linjetykkelse i plot

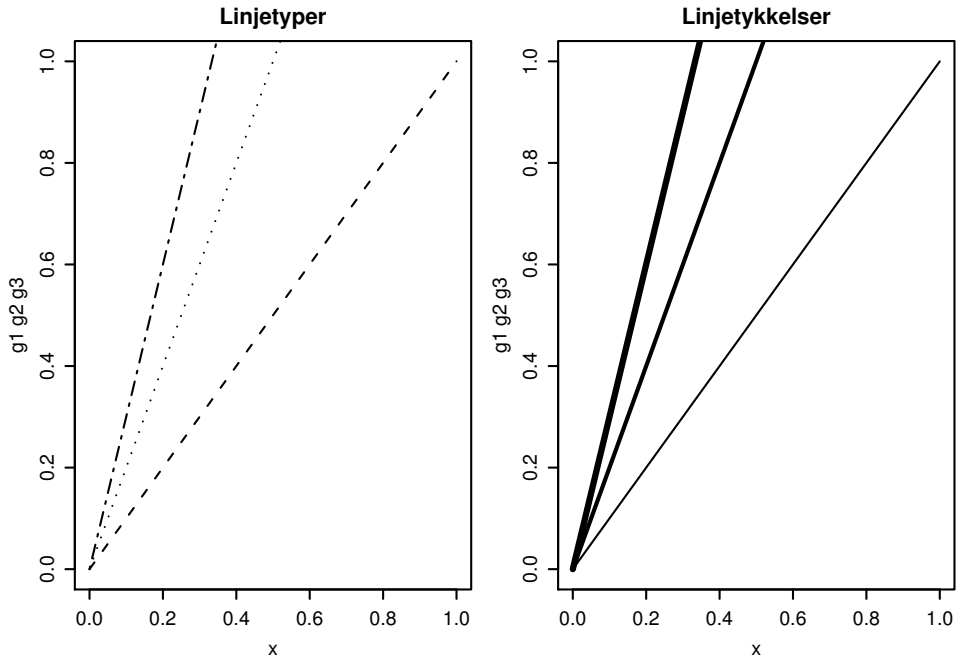
Linjetype og linjetykkelse styres også med parametre til `plot`. Her plottes tre funktioner to gange for at demonstrere forskellige linjetyper og linjetykkelser:

```
g1 <- function(x) { x }
g2 <- function(x) { 2 * x }
g3 <- function(x) { 3 * x }
plot(g1, xlab="x", ylab="g1 g2 g3", main="Linjetyper",
     lty="dashed", las=0)
plot(g2, add=TRUE, lty="dotted")
plot(g3, add=TRUE, lty="twodash")

plot(g1, xlab="x", ylab="g1 g2 g3", main="Linjetykkelser",
     lwd=1, las=1)
plot(g2, add=TRUE, lwd=2)
plot(g3, add=TRUE, lwd=3)
```

[3] I eksemplet i oversigtsskemaet for `colors` på side 167 er en opskrift på hvordan man får en grafisk oversigt over alle navngivne farver.

De resulterende plot ses i figur 10.



Figur 10: Linjetyper og -tykkelser. Til venstre ses fra neden linjetyperne (`lty`) `"dashed"`, `"dotted"` og `"twodash"`; og til højre linjetykkelserne (`lwd`) 1, 2 og 3.

Det kan være mere bekvemt at tilføje titler til et plot efter at det er tegnet. Hertil bruger man funktionen `title`, der tager de samme parametre som `plot`. Parameteren (`main` sætter tekst for oven, `sub` sætter tekst for neden, `xlab` sætter tekst (aksenavn) langs x-aksen, og `ylab` sætter tekst langs y-aksen.

Talrige andre egenskaber ved funktionsplot styres med andre parametre til `plot`. Nogle få er vist i figur 11; flere kan findes i R's hjælp under **`?par`**, **`?plot.default`** og **`?title`** eller i oversigtsskemaerne i appendiks G.

5.6 Indsættelse af tekster i plot

Funktionen `text` bruges til at tilføje en forklarende tekst til et eksisterende plot. Man angiver først (x, y)-koordinaterne for tekstens midtpunkt, og så den tekst der skal indsættes:

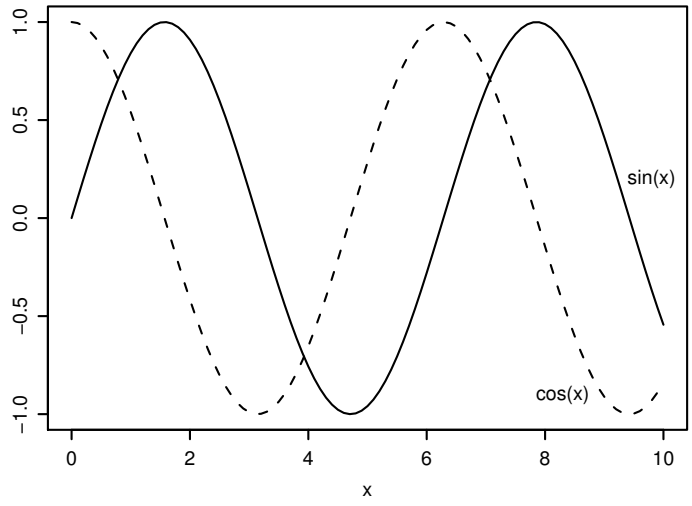
```
plot(sin, 0,10, ylab="")
plot(cos, 0,10, add=TRUE, lty="dashed")
text(9.8, 0.2, "sin(x)")
text(8.3, -0.9, "cos(x)")
```

Resultatet ses i figur 12. Parameteren `col` kan bruges til at styre tekstens farve. En anden parameter, `cex`, kan bruges til at skalere teksten, hvor fx `cex=2` betyder dobbelt størrelse og `cex=0.5` betyder halv størrelse.

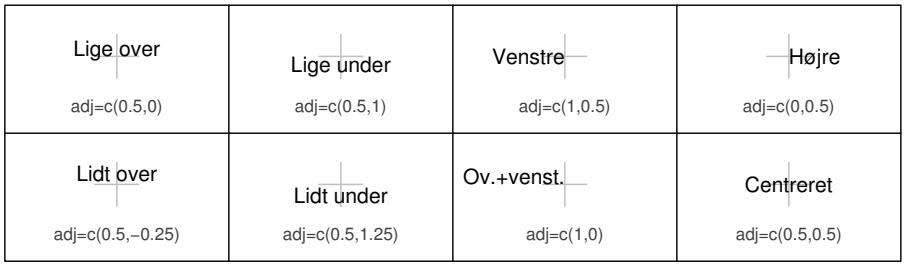
Ofte kan man ønske at sætte en tekst "til venstre for" eller "nedenunder" et bestemt punkt. Frem for at eksperimentere sig frem til nogle brugbare koordinater (som kommer til at afhænge af tekstens længde og skriftens størrelse og så videre) kan man bede om at få teksten placeret med bestemt justering i forhold til de koordinater man giver. Det gøres med parameteren `adj`. Man angiver justeringen som `adj=c(`b`,`h`)` hvor b og h er tal der skal opfattest som *andelen* af tekstens bredde hhv. højde. Se figur 13 for forskellige eksempler. Notationen `c(...)` omtales i afsnit 14.

Parameter	Effekt	Mulige værdier
`asp`	aspect ratio	Tal, fx `asp=1` for lige lange enheder på akserne eller `asp=2` for dobbelt så lange enheder på y-aksen som på x-aksen
`axes`	aksestyring	Brug `axes=FALSE` hvis du styrer akserne med `axis` fra afsnit 5.10
`col`	farve	Se figur 9, fx `col="red"`
`las`	akseetiketstil	`0` (parallel med akse), `1` (vandret), `2` (vinkelret på akse), `3` (lodret)
`log`	logaritmisk plot	`log="x"` (log til x), `log="y"` (log til y), `log="xy"` (dobbeltlog)
`lty`	linjetype	`"solid" "dashed" "dotted" "dotdash" "longdash" "twodash"`
`lwd`	linjetykkelse	1, 2, 3, ..., hvor standard er 1
`main`	overtitel	Tekst, fx `main="Logistisk funktion"`
`sub`	undertitel	Tekst, fx `sub="Logistisk funktion"`
`xlab`	x-aksetekst	Tekst, fx `xlab="Tid i timer"`
`xlim`	x-aksegrænser	Interval, fx `xlim=c(-2, 6)`
`ylab`	y-aksetekst	Tekst, fx `ylab="Antal individer"`
`ylim`	y-aksegrænser	Interval, fx `ylim=c(-20, 40)`

Figur 11: Nogle grafikparametre til brug i funktionen `plot`.



Figur 12: Tilføjelse af tekster med funktionen `text(...)`.



Figur 13: Effekt af `adj` parameteren i `text(`x`,` y`,` *tekst*`, adj=...)`. Punktet (x, y), som teksten er placeret i forhold til, er markeret som et gråt "+" og den sorte tekst er placeret med den angivne værdi af `adj`. Eksemplet "Centreret" svarer til ikke at angive `adj`.

Hvis man har brug for at placere tekster i forhold til kanten af plottet[4] kan man med funktionskaldet `par("usr")` få oplyst koordinaterne (x_1, y_1) for nederste venstre hjørne og (x_2, y_2) for øverste højre hjørne som en vektor (x_1, x_2, y_1, y_2). For at placere en tekst (lidt indrykket) i øverste højre hjørne af plottet kunne man skrive:

```
usr <- par("usr")
text(usr[2], usr[4], "Teksten", adj=c(1.05, 1.25))
```

Notationen `usr[2]` og `usr[4]` for henholdsvis andet og fjerde tal fra vektoren `usr` – altså x_2 henholdsvis y_2 – forklares i afsnit 14.2. Ved at specificere `adj=c(1.05, 1.25)` bliver teksten placeret lidt neden for og til venstre for referencepunktet (x_2, y_2).

5.7 Signaturforklaring i plot

Funktionen `legend` bruges til at indsætte en kasse med signaturforklaring i et eksisterende plot. Man skal enten angive (x, y)-koordinat for øverste venstre hjørne eller også anvende en af følgende specielle "koder" for at angive placeringen af signaturforklaringen inden for plottet: `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"` eller `"center"`. For eksempel kan vi plotte sinus med fuldt optrukken linje og cosinus med stiplet linje, og tilføje en forklaring:

```
plot(sin, 0, 10, ylab="")
plot(cos, 0, 10, add=TRUE, lty="dashed")
legend(2.6, 1, c("sin(x)","cos(x)"), lty=c("solid","dashed"))
```

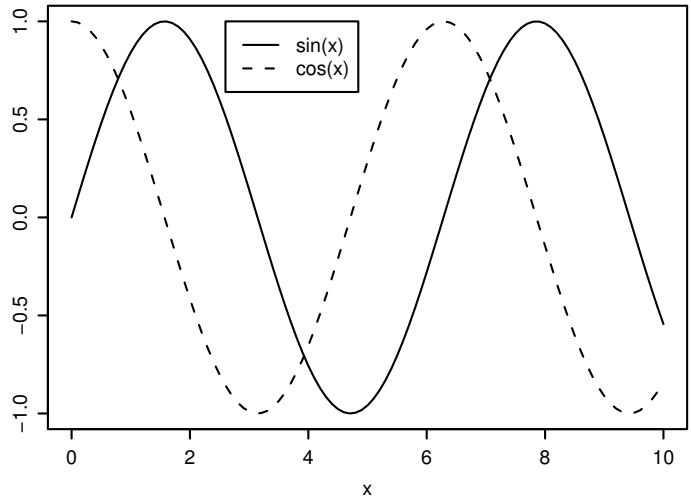
Resultatet ses i figur 14.

Hvis vi havde ønsket signaturforklaringen placeret i nederste venstre hjørne af plotområdet kunne vi stedet have skrevet:

```
legend("bottomleft", c("sin(x)","cos(x)"), lty=c("solid","dashed"))
```

R har ikke nogen erindring om hvad man har plottet og i hvilken rækkefølge, så der kan ikke genereres en automatisk signaturforklaring. Såvel teksterne som signaturerne er man derfor nødt til selv at angive og man må selv sørge for at signaturernes udseende rent faktisk kommer til at svare til det man har plottet. Det vil sige at type, bredde og farve for linjerne er man nødt til at specificere ligesom man har gjort i de kald af `plot` der har tegnet graferne. Heldigvis hedder parametrene det samme i `legend` som i `plot`, så man bruger altså `lty` og `lwd` henholdsvis `col` til at angive type og bredde henholdsvis

[4] Der er en funktion specielt til dette; den hedder `mtext`. Det vil føre for vidt at beskrive den her, se dog oversigtsskemaet på side 163.



Figur 14: Tilføjelse af signaturforklaring med funktionen `legend(...)`.

farve af linjerne i signaturforklaringen. **Bemærk:** Man er nødt til at angive mindst én af parametrene `lty` eller `lwd` for overhovedet at få tegnet linjer med `legend`; det er ikke tilstrækkeligt at angive `col`. Hvis man i kaldene af `plot` alene har brugt `col` og ingen af de andre to parametre (og kurverne dermed har forskellig farve men i øvrigt er "standardlinjer") kan man så angive `lty="solid"` i parametrene til `legend` for at få tegnet linjer i signaturforklaringen.

Bemærk den specielle måde såvel signaturteksterne som parameteren `lty` angives på i eksemplet ovenfor. Da der er to signaturer skal der angives såvel to signaturtekster som to værdier af `lty`, og dette gøres i begge tilfælde ved som parameterværdi at angive en vektor med de to værdier pakket ind i `c(...)`. Den første tekst `"sin(x)"` hører til den første værdi af `lty`, altså `"solid"`, og tilsvarende hører den anden tekst til den anden værdi af `lty`, altså `"dashed"`. Hvis nu den første linje skulle være blå og den anden rød ville vi tilsvarende skulle angive `col=c("blue","red")`. Havde der været tre signaturer skulle der have været angivet tre værdier i hver af vektorerne `c(...)`. Vektorer `c(...)` omtales for alvor i afsnit 14.

5.8 Aflæsning af punkter i plot

Med funktionen `locator` kan man aflæse (x, y)-koordinaterne til punkter i et plot. Dette er nyttigt når man skal finde koordinater til brug i `text` og `legend` funktionerne omtalt ovenfor.

Desuden kan det bruges til at lave omtrentlige aflæsninger af skæringspunkter mellem kurver og akser. For eksempel kan vi forsøge at aflæse (x, y)-koordinaterne for det første skæringspunkt mellem grafen og x-aksen i figur 16 på denne måde:

- Tegn plottet
- I R Console kalder man `locator` med argumentet 1, for at sige at man vil aflæse 1 punkt i plottet:

```
> locator(1)
```

- Dernæst (venstre)klikker man med musen i plottet der hvor kurven skærer x-aksen, og så svarer `locator`-funktionen i R Console med (x, y)-koordinaterne:

```
$x
[1] -1.270422
$y
[1] 0.06869091
```

Som det ses er x omtrent lig -1.27, men det lykkedes åbenbart ikke at ramme x-aksen præcist, for så ville y have været 0.

Hvis man vil aflæse alle tre skæringspunkter i én arbejdsgang, kalder man blot `locator(3)` og klikker på alle tre punkter, hvorefter funktionen returnerer en vektor af x-værdier med tre elementer, og en vektor af y-værdier med tre elementer.

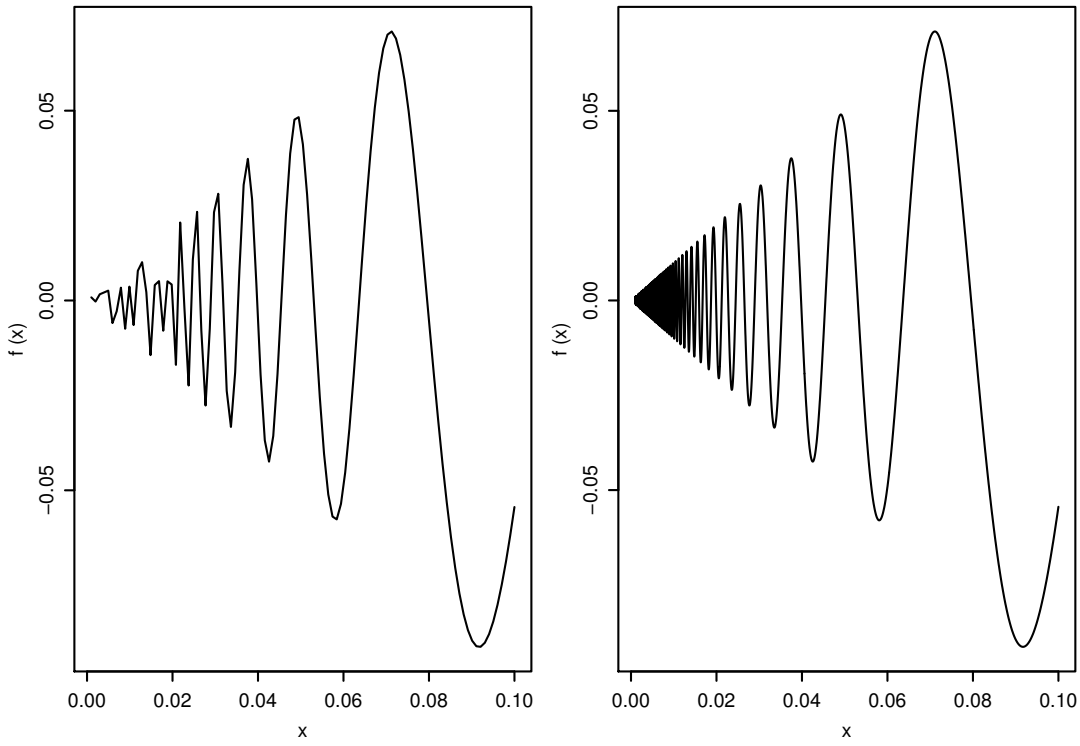
Man kan afbryde `locator`, før man har klikket alle punkter, ved at højreklikke og vælge Stop (Windows), ved at ctrl-klikke (MacOS X) eller ved at taste Esc (begge).

5.9 Antal støttepunkter

Funktionen `plot` beregner normalt den givne funktions værdier i 101 støttepunkter. Hvis grafen for en funktion svinger meget i et lille interval kan det være nødvendigt at sætte antal punkter `n` til en højere værdi end 101. Prøv fx at tegne funktionen $f(x) = x\sin(\frac{1}{x})$, der svinger voldsomt når x er tæt på 0:

```
f <- function(x) { x * sin(1/x) }
plot(f, 0.001, 0.1)
plot(f, 0.001, 0.1, n=5000)
```

De resulterende plot ses i figur 15.

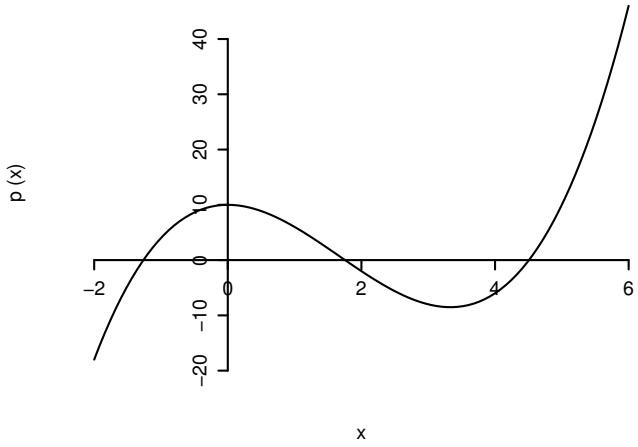


Figur 15: Plot af $f(x) = x\sin(\frac{1}{x})$ for $x \in [0.001, 0.1]$, tegnet med 101 henholdsvis 5000 støttepunkter. Det venstre plot er klart forkert i nærheden af $x = 0$.

5.10 Akser i plot

Som det fremgår af det foregående, sætter R gerne akserne som en kasse uden om selve funktionsgrafen. I matematik er det mere normalt at akserne skærer hinanden i $(0, 0)$. For at opnå dette skal man i `plot`-funktionen angive `axes=FALSE` (med store bogstaver), og dernæst bruge funktionen `axis` til at angive at første-aksen skal gå gennem $y = 0$ og at anden-aksen skal gå gennem $x = 0$:

```
plot(p, -2, 6, axes=FALSE)
axis(1, pos=0)
axis(2, pos=0)
```



Figur 16: Som figur 4, men med `axes=FALSE`, `axis(1, pos=0)` og `axis(2, pos=0)`.

Det resulterende plot ses i figur 16. Man kan også bestemme præcis hvilke aksemærker ("ticks") der skal sættes, med parameteren `at`. For eksempel kan man sætte mærker på x-aksen for hvert heltal fra -2 til 6 på denne måde:

```
plot(p, -2, 6, axes=FALSE)
axis(1, at=seq(-2,6), pos=0)
axis(2, pos=0)
```

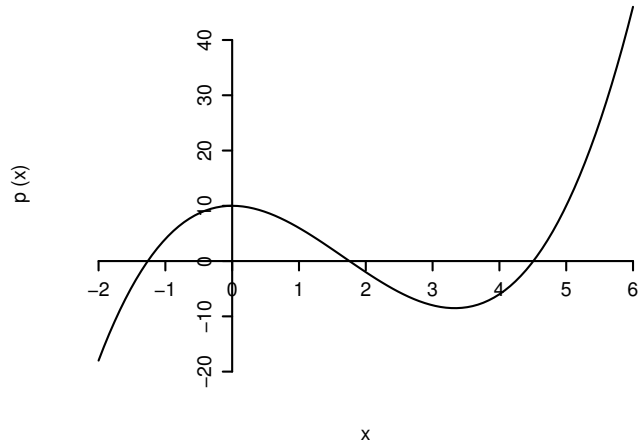
Det resulterende plot ses i figur 17. Funktionen `seq`, der her bruges til at generere talrækken -2, -1, . . . , 6, forklares i afsnit 14.1.

Man kan med `labels=FALSE` sætte aksemærker *uden* etiketter. Dette kan være nyttigt til med to kald af `axis` først at sætte mærker *med* etiketter og derefter sætte mærker i en finere inddeling *uden* etiketter. De sekundære mærker bør man så få tegnet med en lidt kortere længde, for eksempel:

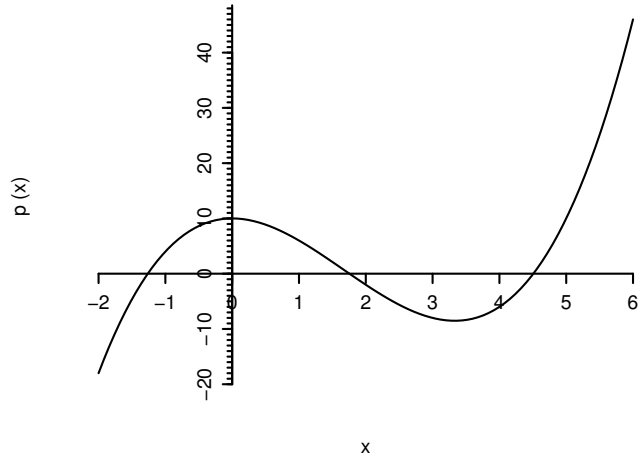
```
plot(p, -2, 6, axes=FALSE)
axis(1, at=seq(-2,6), pos=0)
axis(2, pos=0)
axis(2, pos=0, at=seq(-20,50), labels=FALSE, tcl=-0.25)
```

Det resulterende plot ses i figur 18. Parameteren `tcl` regulerer længden af stregerne, der tegnes som mærker; de normale mærker svarer til `tcl=-0.5` og med `tcl=-0.25` får vi altså halvdelen af den normale længde. Med positive værdier af `tcl` tegnes mærkerne på den anden side af akserne (altså modsat teksterne).

Hvis man udelader parameteren `pos` i `axis` vil aksen blive tegnet i siden af plottet, på samme position som hvis man havde ladet `plot` tegne den i første omgang. Hvis man gerne vil bibeholde den sædvanlige position af akserne som en kasse uden om plottet men selv vil regulere mærkerne kan man gøre som følger:



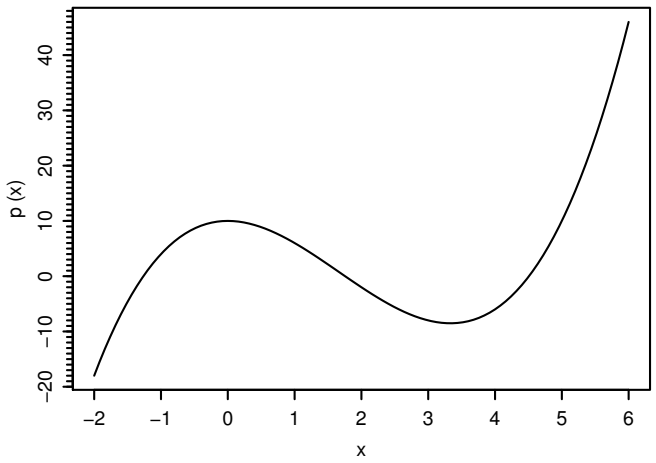
Figur 17: Som figur 16, men med `axis(1, at=seq(-2,6), pos=0)`.



Figur 18: Som figur 17, men tilføjet `axis(2, pos=0, at=seq(-20,50), labels=FALSE, tcl=-0.25)`.

```
plot(p, -2, 6, axes=FALSE)
axis(1, at=seq(-2,6))
axis(2)
axis(2, at=seq(-20,50), labels=FALSE, tcl=-0.25)
box()
```

Resultatet ses i figur 19. Den sidste funktion, `box`, tegner selve rammen. I stedet for at bruge `box` kunne man i selve kaldet af `plot` have angivet `frame.plot=TRUE` (sammen med `axes=FALSE`).



Figur 19: Som figur 18, men uden `pos=0` og til gengæld tilføjet `box()`.

Der er flere andre parametre man kan bruge til at styre udseendet af mærker og akseetiketter; den vigtigste af disse er nok `las` som styrer hvilken led, akseetiketterne skrives på. Standard er `las=0`, som giver etiketter parallelt med akserne. Andre muligheder er `las=1`, hvilket giver vandrette etiketter, `las=2`, hvilket giver etiketter vinkelret på akserne, og `las=3`, hvilket giver lodrette etiketter. Parameteren `las` kan også anvendes direkte i `plot`, hvis man ikke angiver `axes=FALSE`

5.11 Hovedprincipper for fremstilling af plot

Når man arbejder med fremstilling af grafer/plot i R er det vigtigt at holde sig følgende hovedprincipper for øje:

1. Et plot opbygges altid med først et kald af `plot` og derefter tilføjes eventuelt flere ting med `plot(..., add=TRUE)` og/eller andre funktioner (`axis`, `text`, `legend`, osv).

2. Det første kald af `plot` bestemmer plotområdet (aksegrænserne). Det kan *ikke* ændres af de senere funktioner. Brug derfor `ylim` (og eventuelt `xlim`) i første kald af `plot` hvis R's automatisk beregnede plotområde bliver forkert.

3. Der kan aldrig fjernes noget fra et plot, kun tilføjes nye ting. Således vil fx `title` kun tilføje tekster, ikke ændre, fjerne eller flytte eksisterende tekst; `axis` vil indtegne en akse og ikke ændre, fjerne eller flytte eksisterende akser, osv.

4. For at *fjerne* noget fra eller *ændre* noget i et plot er man nødt til at opbygge hele plottet igen, men ændre eller slette det funktionskald, der tegnede det uønskede. Derfor er det en *meget* god idé at arbejde i *R Editor* vinduet så det er nemt at udføre alle linjerne, der opbygger plottet, hver gang man laver en ændring.

6 Eksport af plot fra R

6.1 Indsættelse af plot i rapporter og præsentationer

Under **Windows** kan man inkludere et plot i en rapport eller præsentation ved at kopiere plottet i grafikformatet Windows (Enhanced) Metafile og derefter indsætte det i sit dokument:

- Lav plottet i R.
- Højreklik på plottet og vælg Copy as metafile; eller peg på plottet og tast Ctrl+W.
- Skift til tekstbehandlings- eller præsentationsprogrammet og indsæt figuren ved at vælge Rediger || Sæt ind eller ved at taste Ctrl+V.

Efter indsættelse kan plottet skaleres (formindskes eller forstørres) uden kvalitetsforringelse hvis det er overført som Windows Metafile. Hvis plottet er overført som bitmap, bliver det grimt ved skalering.

Under **MacOS** bruger man PDF for at eksportere figurer (vælg File || Save fra menuen når plotvinduet er aktivt eller brug ⌘+C for at kopiere til udklipsholderen).[5]

6.2 Grafikfiler; Indsættelse af plot på websider

Hvis man skal lave mange plot, så er det praktisk at gemme hvert plot direkte i en separat grafikfil. At gemme et plot i filen `poly.emf` i formatet Windows Metafile gøres sådan her (kun under Windows):

```
p <- function(x) { 10 - 5*x^2 + x^3 }
win.metafile("poly.emf")
plot(p, -2, 6)
dev.off()
```

Funktionen `dev.off` afslutter opbygning af plot-filen, og først derefter er filen `poly.emf` klar. En fil af denne type kan indsættes i OpenOffice (med Indsæt || Grafik || Fra fil) eller Word eller PowerPoint (med Indsæt || Billede || Fra fil). Men direkte overførsel af plots er nu oftest nemmere; se afsnit 6.1.

To andre gode grafikformater er Portable Document Format (PDF) og Encapsulated Postscript. Begge formater kan skaleres uden kvalitetstab og bruges i professionel bogproduktion. PDF-filer kan bruges uanset operativsystem; Postscript er mest egnet til tekstbehandling i Linux og til dels MacOS X. Plotfilerne laves på samme måde som ovenfor:

```
postscript("poly.eps")
plot(p, -2, 6)
dev.off()
pdf("poly.pdf")
plot(p, -2, 6)
dev.off()
```

Hvis et plot skal bruges på en webside duer formaterne ovenfor ikke. I stedet kan grafen laves i grafikformatet Portable Network Graphics (PNG):

```
png("poly.png")
plot(p, -2, 6)
dev.off()
```

PNG-filer er gode til websider fordi de er ret kompakte, men de bliver grimme hvis de skaleres, som det normalt er nødvendigt i dokumenter og præsentationer.

[5] Det er ikke sikkert at ens tekstbehandlingsprogram kan håndtere PDF (specifikt kan OpenOffice ikke). Så er man desværre nødt til at gå på kompromis med kvaliteten og konvertere til et rasterformat. Det kan man gøre fx via Billedfremviser (brug ⌘+N i Billedfremviser for at indlæse fra udklipsholderen) — man kan kopiere fra Billedfremviser tilbage til udklipsholderen på sædvanlig vis med ⌘+C og derfra sætte det grimme, rasteriserede plot ind i sit dokument. Grrr, hvor irriterende.

7 Nulpunkt for funktion, og løsning af ligning med én ubekendt

I R kan man løse vilkårlige ligninger med én ubekendt x ved at finde nulpunkt for en passende funktion. For eksempel kan man løse ligningen $x = \cos(x)$ ved at definere funktionen $f(x) = x - \cos(x)$ og så bruge R til at finde en værdi af x for hvilken $f(x) = 0$:

```
> f <- function(x) { x - cos(x) }
> res <- uniroot(f, c(-10, 10))
> res
$root
[1] 0.7390799
$f.root
[1] -8.831189e-06
$iter
[1] 5
$estim.prec
[1] 6.103516e-05
```

Argumentet `c(-10,10)` til `uniroot` er en vektor (afsnit 14) som angiver at nulpunktet skal søges i intervallet $[-10, 10]$.

Vigtigt: Værdien af funktionen `f` skal have forskelligt fortegn i de to endepunkter af søgeintervallet. R finder nulpunktet ved iterativt at indsnævre søgeintervallet og hele tiden sørge for at `f` har forskelligt fortegn i de to ender – for da må der jo være et nulpunkt inde i intervallet (hvis funktionen er kontinuert).

Resultatet fra `uniroot`, der ovenfor gemmes i funktionen `res`, er en associationsliste (afsnit 20.1) med fire komponenter: `root` er løsningen til ligningen; `f.root` er funktionens værdi beregnet i det fundne punkt; `iter` angiver hvor mange iterationer, der skulle til for at finde løsningen; og `estim.prec` er den maksimale fejl på løsningen. Som regel er det kun `root` komponenten, der skal bruges.

For at få komponenten `root` ud af `res` (så man kan bruge tallet i et regneudtryk) skal man skrive `res$root`[6]; mere herom i afsnit 20.1:

```
> res$root
[1] 0.7390799
```

Det er ikke alle funktioner der har nulpunkter og ikke alle ligninger der har løsninger, så man skal tjekke at `f.root` faktisk er tæt på nul. I eksemplet ovenfor er den `-8.831189e-06`, der som bekendt betyder $-8.831189 \cdot 10^{-6}$, altså -0.000008831178, hvilket er ganske tæt på nul. På grund af computerens begrænsede regnenøjagtighed og `uniroot`'s fremgangsmåde kan man ikke forvente at resultatet er eksakt nul.

Med `uniroot` kan man kun finde ét nulpunkt ad gangen. For at finde alle nulpunkter kan man plotte funktionen, og derefter søge nulpunkter separat for passende små intervaller hvor $f(x)$ skifter fortegn. For eksempel ser man for polynomiet $p(x) = 10 - 5x^2 + x^3$ i figur 16 at der må være et nulpunkt i intervallet $[-2, 0]$, et nulpunkt i intervallet $[0, 2]$, og et nulpunkt i intervallet $[4, 6]$:

```
> uniroot(p, c(-2,0)) $ root
[1] -1.263545
> uniroot(p, c(0,2)) $ root
[1] 1.755642
> uniroot(p, c(4,6)) $ root
[1] 4.507902
```

Men netop i specialtilfældet hvor funktionen er et polynomium $f(x) = a_0 + a_1 x + \cdots + a_n x^n$ kan man finde alle rødder på én gang ved at bruge den specielle funktion `polyroot(c(`a_0`,` a_1`, ...,` a_n`))`.

[6] Dollartegnet `$` fås på et PC-tastatur med AltGr+4 og på en Mac med ⌥+⇧+3.

Med `polyroot` får man altid rødderne som komplekse tal:[7]

```
> polyroot(c(10, 0, -5, 1))
[1]  1.755640-0i -1.263543+0i  4.507903-0i
```

For at få ovenstående omdannet til almindelige reelle tal, uden $\pm$`0i`, kan vi benytte funktionen `Re`, der simpelthen giver realdelen af et komplekst tal:

```
> Re( polyroot(c(10, 0, -5, 1)) )
[1]  1.755640 -1.263543  4.507903
```

(Rødderne kommer ikke nødvendigvis i nogen bestemt orden...)

[7] *Komplekse tal* er, løst sagt, en udvidelse af de reelle tal til tal af formen $a+bi$, hvor a og b er reelle tal og i er den såkaldt *imaginære enhed*, der opfylder $i^2 = -1$. For $b = 0$ har man de almindelige reelle tal. R kan repræsentere og regne med komplekse tal. De vises på formen a+b`i` eller a−b`i`, hvor a og b vises afrundet efter R's sædvanlige regler (se afsnit 2.1). (Se eventuelt også note 17 på side 73 for lidt mere om komplekse tal.)

Polynomier af n'te grad har altid n rødder når man regner med komplekse tal; nogle eller alle af disse rødder kan selvfølgelig være reelle tal (have $b = 0$). Afrundingsfejl i computeren betyder, at `polyroot` som regel vil finde en lille imaginærdel selv for reelle rødder.

8 Numerisk optimering: minimum og maksimum

Numerisk optimering af en funktion f går ud på at finde den værdi af x som giver den mindste (eller største) værdi $f(x)$ af funktionen. For eksempel kunne $f(x)$ udtrykke det økonomiske udbytte ved at tilføre x enheder kunstgødning til en mark. Dette udbytte består jo af en indtægt fra salg af høstudbyttet minus en udgift til indkøb og spredning af kunstgødningen. Lad os antage at høstudbyttet som funktion af gødningstilførslen t beskrives af funktionen

$$u(t) = \frac{20t}{t+1} + 10$$

og at vi tjener 4000 kroner per ton høstudbytte men skal betale 5000 kroner per ton gødning, så det økonomiske udbytte som funktion af gødningstilførslen t er

$$f(t) = 4000u(t) - 5000t$$

Nu kan vi bruge R til at finde den værdi af t der maksimerer fortjenesten:

```
> u <- function(t) { 20 * t / (t+1) + 10 }
> f <- function(t) { 4000 * u(t) - 5000 * t }
> optimize(f, interval=c(0,100), maximum=TRUE)
$maximum
[1] 2.999998
$objective
[1] 85000
```

Argumenterne til funktionen **`optimize`** er først funktionen $f(t)$, der skal maksimeres, og dernæst det interval af t-værdier der skal undersøges, her $[0, 100]$. Parameteren **`maximum=TRUE`** angiver at $f(t)$ skal maksimeres, ikke minimeres. Resultatet er en associationsliste (afsnit 20.1) med to komponenter: `maximum` er den værdi af t der maksimerer $f(t)$, og `objective` er den tilsvarende maksimale værdi af $f(t)$.

Man skal tænke sig om når man bruger numerisk optimering, for der er flere faldgruber:

1. Det kan være at funktionen slet ikke har noget minimum (eller maksimum), enten fordi den er ubegrænset eller fordi den konvergerer mod en øvre (eller nedre) grænse for $x \to \pm\infty$. For eksempel har funktionen $g(x) = -x^2$ ikke noget minimum, og $h(x) = 1 - |\frac{1}{x}|$ har ikke noget maksimum.

2. Det kan være at funktionen har flere minima (eller maksima); i så fald giver R's optimeringsrutine bare et af dem, men ikke nødvendigvis det man forventede. Således er det vigtigt i eksemplet ovenfor at søge maksimum blandt ikke-negative værdier af gødningstilførslen t, her intervallet $[0, 100]$. Dels er det meningsløst i praksis at tilføre en negativ mængde gødning, dels kan funktionen f faktisk antage vilkårlig store værdier for store negative t.

3. Det kan være at funktionen har lokale minima (eller maksima) og at R's optimeringsrutine finder et af de lokale minima (eller maksima) i stedet for et af de globale.

Funktionen `optimize` kan kun bruges til at optimere (maksimere eller minimere) en funktion $f(x)$ af én variabel. En anden funktion, `optim`, kan bruges til at optimere funktioner af flere variable.

9 Numerisk integration

Man kan bruge R til at foretage numerisk integration af en funktion. For eksempel kan man integrere sinusfunktionen fra 0 til π:

```
> integrate(sin, 0, pi)
2 with absolute error < 2.2e-14
```

Dette virker naturligvis på samme måde med en funktion man selv har defineret:

```
> f <- function(x) { 7*x^2+x^4 }
> integrate(f, 0, 3)
111.6 with absolute error < 1.2e-12
```

Ud over værdien af integralet får man også en vurdering af størrelsen af fejlen.

Funktionen **integrate** giver ikke bare en talværdi, men en associationsliste (afsnit 20.1) med bl.a. komponenterne `value` (værdien af integralet) og `abs.error` (størrelsen af fejlen).[8] Hvis man ønsker at regne videre på resultatet af integrationen skal man altså udtage `value` fra listen med notationen **res$value**, fx som følger:

```
> res <- integrate(f, 0, 3)
> res$value * 3
[1] 334.8
```

Det er muligt at integrere til eller fra uendelig (`Inf`), så længe resultatet er endeligt:

```
> f1 <- function(x) { exp(-2*x) }
> integrate(f1, 0, Inf)
0.5 with absolute error < 7.7e-05
> f2 <- function(x) { 1/x^2 }
> integrate(f2, 1, Inf)
1 with absolute error < 1.1e-14
> f3 <- function(x) { exp(-x^2) }
> integrate(f3, -Inf, Inf)
1.772454 with absolute error < 4.3e-06
```

Funktionen **integrate** virker ved at beregne værdien af den givne funktion i passende mange punkter. Antallet kan begrænses med parameteren **subdivisions**, hvis standardværdi er 100. For funktioner såsom $f(x) = \sin(\frac{1}{x})$, som svinger så meget, at det er vanskeligt at beregne integralet numerisk, kan det være nødvendigt at forøge **subdivisions** for at få et resultat:

```
> f4 <- function(x) { sin(1/x) }
> integrate(f4, 0, 1)
Error in integrate(f4, 0, 1) :
        maximum number of subdivisions reached
> integrate(f4, 0, 1, subdivisions=10000)
0.5041151 with absolute error < 9.7e-05
```

Man kan ikke bruge R til at beregne stamfunktioner ved symbolsk integration.

[8] Som det ses af eksemplerne vises denne associationsliste på en anden måde end dem man får fra `uniroot` og `optimize` beskrevet i de foregående afsnit, men det vil føre for vidt at forklare hvorfor.

10 Datasæt

Datasæt (data frames) er et centralt begreb i R, der jo netop er udviklet primært med henblik på statistisk analyse. Selv om det i disse noter ikke er de statistiske faciliteter der fokuseres på er det alligevel nyttigt og nødvendigt at beskrive de mest basale aspekter af datasæt i R, specielt hvad angår indlæsning, transformering og grafisk fremstilling af forsøgsdata.

Man kan tænke på et datasæt som en tabel eller et skema, hvor hver række er en sammenhængende observation og hver søjle er en variabel der er observeret. I figur 20 er som eksempel i tabelform vist et datasæt der stammer fra et forsøg med slagtesvin der har fået væksthormon. Datasættet har de tre variable `Tid`, `Kontrol` og `Vaekst`, og 20 observationer. Hver observation angiver et antal minutter efter slagtning (søjlen `Tid`) sammen med målt pH i kødet fra grise opdrættet normalt (søjlen `Kontrol`, dvs. kontrolgruppen) og fra grise der har fået væksthormon (søjlen `Vaekst`).

Tid	Kontrol	Vaekst
30	6.45	6.07
45	6.32	5.94
60	6.17	5.82
75	6.06	5.72
90	6.02	5.62
105	5.98	5.54
120	5.94	5.47
150	5.85	5.38
180	5.80	5.37
210	5.76	5.34
240	5.71	5.34
270	5.63	5.35
300	5.59	5.31
330	5.52	5.31
360	5.49	5.32
390	5.47	5.32
420	5.45	5.33
450	5.44	5.33
480	5.43	5.34
1440	5.43	5.34

Figur 20: Datasæt fra forsøg med slagtesvin. Der er tre variable og 20 observationer.

Afsnit 10.1 til 10.4 handler om hvordan datasæt kan indlæses fra tekstfiler til R. Afsnit 10.5 beskriver kort hvordan datasæt kan eksporteres til tekstfiler der derefter kan indlæses i regnearksprogrammer. Afsnit 10.6 handler om hvordan man kan få et hurtigt overblik over et datasæt og dermed kontrollere at det er indlæst korrekt. Afsnit 10.7 handler om hvordan man kan regne på værdierne fra et datasæt og transformere variable.

Afsnit 12 beskriver hvordan man kan lave en grafisk fremstilling af målingerne i et datasæt. Afsnit 13 beskriver lineær regression og regressionskurver bl.a. ud fra datasæt.

Afsnit 20 beskriver lidt mere formelt hvilken type objekt et datasæt er i R, hvordan man kan oprette datasæt (uden indlæsning), hvordan man kan ændre i datasæt og hvordan man kan udtage deldatasæt hvor rækkerne opfylder givne betingelser.

10.1 Indlæsning af forsøgsdata

Forsøgsdata i tekstfiler foreligger normalt enten i *mellemrumssepareret format* eller *kommasepareret format*. Figur 21 viser hvordan de samme data ser ud i de to formater hvis man bruger fx NotePad eller WordPad til at se på tekstfilerne. Indlæsning fra mellemrumssepareret format er beskrevet i afsnit 10.1.1 og indlæsning fra kommasepareret format er beskrevet i afsnit 10.1.2.

Kopiering med klippe-klistre fra regneark (fx Excel eller OpenOffice) til R er beskrevet i afsnit 10.2. En anden metode til dataoverførsel fra regneark, som formentlig er bedre ved meget store datasæt, er beskrevet i afsnit 10.3.

10.1.1 Forsøgsdata i mellemrumssepareret format

Figur 21 (a) viser en tekstfil med data opstillet i mellemrumssepareret format. I eksemplet står overskrifter og data nydeligt opstillet under hinanden fordi der er tilpas mange mellemrumstegn på hver linje, men det vigtige er at søjlerne er adskilt med mindst ét mellemrum.

```
 Tid Kontrol  Vaekst
  30    6,45    6,07
  45    6,32    5,94
  60    6,17    5,82
  75    6,06    5,72
  90    6,02    5,62
 105    5,98    5,54
 120    5,94    5,47
 150    5,85    5,38
 180    5,80    5,37
 210    5,76    5,34
 240    5,71    5,34
 270    5,63    5,35
 300    5,59    5,31
 330    5,52    5,31
 360    5,49    5,32
 390    5,47    5,32
 420    5,45    5,33
 450    5,44    5,33
 480    5,43    5,34
1440    5,43    5,34
```

(a) Mellemrumssepareret `Grise2Fast.txt`

```
Tid;Kontrol;Vaekst
30;6,45;6,07
45;6,32;5,94
60;6,17;5,82
75;6,06;5,72
90;6,02;5,62
105;5,98;5,54
120;5,94;5,47
150;5,85;5,38
180;5,8;5,37
210;5,76;5,34
240;5,71;5,34
270;5,63;5,35
300;5,59;5,31
330;5,52;5,31
360;5,49;5,32
390;5,47;5,32
420;5,45;5,33
450;5,44;5,33
480;5,43;5,34
1440;5,43;5,34
```

(b) Kommasepareret, DK `Grise2KommaDK.csv`

```
Tid,Kontrol,Vaekst
30,6.45,6.07
45,6.32,5.94
60,6.17,5.82
75,6.06,5.72
90,6.02,5.62
105,5.98,5.54
120,5.94,5.47
150,5.85,5.38
180,5.8,5.37
210,5.76,5.34
240,5.71,5.34
270,5.63,5.35
300,5.59,5.31
330,5.52,5.31
360,5.49,5.32
390,5.47,5.32
420,5.45,5.33
450,5.44,5.33
480,5.43,5.34
1440,5.43,5.34
```

(c) Kommasepareret, US `Grise2KommaUS.csv`

Figur 21: Tre tekstfiler i (a) mellemrumssepareret format, (b) dansk kommasepareret format og (c) amerikansk kommasepareret format. Dansk OpenOffice og Excel eksporterer normalt i format (b), mens engelsksprogede versioner eksporterer i format (c). Filtypen `.csv` betyder "comma-separated variables".

En tekstfil i mellemrumssepareret format og med decimalkomma (a) kan indlæses i R med funktionen `read.table`:

```
d <- read.table("Grise2Fast.txt", header=TRUE, dec=",")
```

Tekstfilen `Grise2Fast.txt` skal ligge i den aktuelle filmappe, ellers skal man angive filnavnet inklusiv stien til mappen hvor filen findes. Parameteren `header=TRUE` angiver at første linje af tekstfilen indeholder datasættets variabelnavne (søjlenavne), og `dec=","` angiver at denne fil benytter decimalkomma i tal `6,45` (som på dansk, tysk og fransk) snarere end decimalpunktum `6.45` (som på amerikansk og engelsk).

Ved ovenstående ordre indlæses data fra filen til et R-datasæt `d`. Som med alle andre variable kan man få vist værdien af `d` bare ved at skrive navnet:

```
> d
    Tid Kontrol Vaekst
1    30    6.45   6.07
2    45    6.32   5.94
...
19  480    5.43   5.34
20 1440    5.43   5.34
```

R viser alle 20 rækker, men i disse noter er de midterste rækker udeladt af pladshensyn. Som det ses bruger R altid decimalpunktum i tal, uanset hvad der stod i den oprindelige tekstfil. Den venstre søjle indeholder R's automatiske nummerering af observationerne.

Pas på, typisk fejl: Havde vi glemt `dec=","` i `read.table` ovenfor havde R alligevel uden at kny indlæst datasættet men opfattet de to søjler med kommatal som *tekster* i stedet for som tal. Hvis man bare viser datasættet ser det hele meget tilforladeligt ud:

```
> fejl <- read.table("Grise2Fast.txt", header=TRUE)
> fejl
    Tid Kontrol Vaekst
1    30    6,45   6,07
2    45    6,32   5,94
...
19  480    5,43   5,34
20 1440    5,43   5,34
```

Man skal være meget vågen for at opdage at der her står kommaer hvor der burde være decimalpunktummer! Når man så begynder at ville lave en grafisk fremstilling af datasættet eller regne på værdierne vil man opleve det som om R opfører sig meget mystisk. Det sikreste er at bruge funktionen `summary` som beskrives i afsnit 10.6 til at kontrollere at datasættet er korrekt indlæst.

Pas også på: Havde vi glemt `header=TRUE` i `read.table` havde R også her alligevel uden at kny indlæst datasættet men så opfattet alle søjler som tekster og selv opfundet nogle overskrifter. Hvis man bare viser datasættet ser det hele igen meget tilforladeligt ud, omend der er flere forskelle til det korrekt indlæste datasæt end før:

```
> fejl2 <- read.table("Grise2Fast.txt", dec=",")
> fejl2
   V1      V2      V3
1   Tid Kontrol Vaekst
2    30    6,45   6,07
3    45    6,32   5,94
...
20  480    5,43   5,34
21 1440    5,43   5,34
```

Bemærk at der nu er en observation for meget (21 i stedet for 20) og at overskrifterne er V1 V2 V3. Igen vil `summary` gøre det lettere at afsløre fejlen end hvis man bare viser selve datasættet.

10.1.2 Forsøgsdata i kommasepareret format

Forsøgsdata i såkaldt kommasepareret format ligger i en tekstfil hvor hver linje svarer til en observation og hvor variablene (søjlerne) er adskilt af semikolon (`;`) eller komma (`,`) afhængig af om filen er i dansk eller amerikansk format. Tekstfiler i kommasepareret format har typisk et navn der ender på `.csv` for "comma-separated variables", men navnet kan også ende på `.txt`. Figur 21 (b) og (c) viser samme tekstfil i henholdsvis dansk og amerikansk kommasepareret format.

Tekstfiler i dansk kommasepareret format (b), med semikolon som skilletegn og med decimalkomma i tal, kan indlæses i R med funktionen `read.csv2`:

```
d <- read.csv2("Grise2KommaDK.csv")
```

Ovenstående er synonym for:

```
d <- read.table("Grise2KommaDK.csv", sep=";", dec=",", header=TRUE)
```

Tekstfiler i amerikansk kommasepareret format (c), med komma som skilletegn og med decimalpunktum i tal, kan indlæses i R med funktionen `read.csv`:

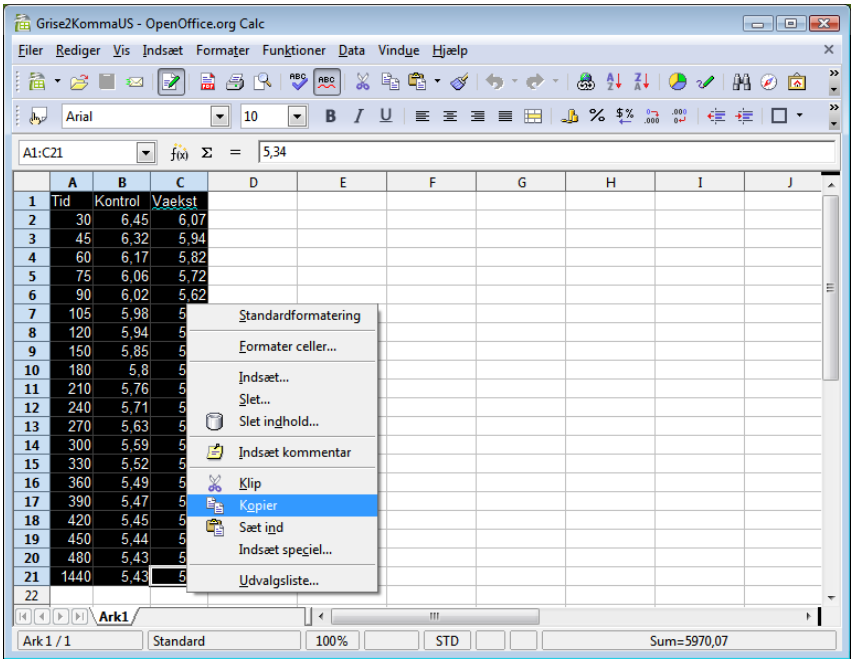
```
d <- read.csv("Grise2KommaUS.csv")
```

Ovenstående er synonym for:

```
d <- read.table("Grise2KommaUS.csv", sep=",", header=TRUE)
```

10.2 Data fra regneark til R med klippe-klistre

Den nemmeste måde at flytte data fra et regneark til R er måske med kopiering via udklipsholderen. Start med at markere dataene i regnearket og derefter enten taste Ctrl+C eller højreklikke og vælge Kopier (på Mac: ⌘+C) som vist i figur 22. I begge tilfælde anbringes en kopi af dataene i udklipsholderen (clipboard).



Figur 22: Markering og kopiering af data i regneark.

Skift derefter til R Console, hvor man kan læse fra udklipsholderen `clipboard`. I Windows foregår det således:

```
> d <- read.table("clipboard", header=TRUE, dec=",")
```

Under MacOS således:

```
> d <- read.table(pipe("pbpaste"), header=TRUE, dec=",")
```

Under Linux således:

```
> d <- read.table("X11_clipboard", header=TRUE, dec=",")
```

Når man overfører data fra et engelsksproget regneark skal argumentet `dec=","` selvfølgelig udelades, fx:

```
> d <- read.table("clipboard", header=TRUE)
```

Bemærk at når man redigerer i R Console eller i et andet program, så kan den "klippede" tekst forsvinde fra udklipsholderen, og `read.table("clipboard", ...)` eller tilsvarende vil ikke virke før man igen har kopieret data til udklipsholderen.

I stedet for at læse direkte fra regneark til R med klippe-klistre kan man dog med fordel først eksportere data fra regneark til en tekstfil i kommasepareret format (`.csv`), og dernæst importere fra tekstfil til R med funktionen `read.csv2` eller `read.csv`. Fordelen ved dette er at man kan gemme tekstfilen og senere tjekke (med NotePad eller WordPad) at det var de rigtige tal man regnede videre på — betryggende hvis man bruger resultaterne i sit bachelor- eller specialeprojekt eller en anden videnskabelig publikation.

10.3 Direkte indlæsning af data fra regneark til R

Med R-systemets databaseinterface `RODBC` kan man under Windows indlæse data direkte fra regneark i Excel-format (`.xls`).

For at læse data direkte fra et Excel-regneark til R skal R-pakken `RODBC` være installeret, som forklaret i appendiks A.2. Desuden er det formentlig nødvendigt at have Microsoft Office installeret. Når det er på plads kan man læse `Ark1` fra regnearksmappen `Grise2.xls` på denne måde:

```
> library(RODBC)
> channel <- odbcConnectExcel("Grise2.xls")
> d <- sqlFetch(channel, "Ark1")
> odbcClose(channel)
```

Herved indlæses dataene til datasættet `d` som i afsnit 10.1. I ordren `sqlFetch` betyder argumentet `"Ark1"` at der skal læses data fra arket med navn `Ark1` i regnearksmappen `"Grise2.xls"` — en regnearksmappe indeholder jo gerne mere end ét regneark. I dansk OpenOffice og Excel hedder arkene normalt `Ark1`, `Ark2`, osv; mens de i amerikansk OpenOffice og Excel normalt hedder `Sheet1`, `Sheet2`, osv.

10.4 Indlejring af forsøgsdata i R-scripts

Ved hjælp af funktionen `textConnection` kan man indskrive datasæt direkte i et R-script. Dette er især nyttigt hvis man vil sende et komplet R-eksempel til andre eller vil gemme præcis de R-ordrer der blev brugt til at lave figurer til en rapport. Hvis man kender til statistiksystemet `SAS` svarer det til at bruge `CARDS`-kommandoer til at indlægge data i programmer.

For eksempel kan man indlægge grise-datasættet fra afsnit 10.1.1 i et R-script og "indlæse" det til en data frame `d` med `read.table` sådan her:

```
d <- read.table(header=TRUE, dec=",", textConnection(
"    Tid Kontrol  Vaekst
      30    6,45    6,07
      45    6,32    5,94
     ...     ...     ...
     480    5,43    5,34
    1440    5,43    5,34"))
```

Helt tilsvarende kan data opstillet i (dansk) kommasepareret format "indlæses" med `read.csv2` fra en `textConnection`:

```
d <- read.csv2(textConnection(
"Tid;Kontrol;Vaekst
30;6,45;6,07
45;6,32;5,94
...
480;5,43;5,34
1440;5,43;5,34"))
```

Hvis et af ovenstående R-scripts er gemt i en fil `gris.R`, så kan det indlæses og udføres i R-systemet med funktionen `source("gris.R")` eller fra menuen med File || Source R code.

10.5 Eksport af data fra R

Datasæt som `d` kan eksporteres fra R til tekstfil — og dermed fx til regneark — med funktionen `write.table`.

Som standard skriver `write.table` i det mellemrumsseparerede format med overskrifter der er standard for `read.table`. Skal man således blot senere kunne læse datasættet ind i R igen er det nemmest bare at skrive:

```
write.table(d, "minedata.txt")
```

Den resulterende tekstfil `minedata.txt` lægges i den aktuelle filmappe og kan indlæses i R igen med:

```
d <- read.table("minedata.txt")
```

For eksport til regneark er det bedst at skrive en af de gængse kommaseparerede filformater (dansk eller engelsk). For at skrive i dansk kommasepareret format som i figur 21 (b), skal man angive semikolon (`;`) som separator og bruge decimalkomma (`,`) i tal:

```
write.table(d, "minedata.csv", sep=";", dec=",", quote=FALSE,
            row.names=FALSE)
```

Parameteren `quote=FALSE` betyder at der ikke kommer anførselstegn om tekster (fx søjleoverskrifter) i tekstfilen, og parameteren `row.names=FALSE` betyder at der ikke kommer rækkenumre på observationerne. Ovenstående svarer til at skrive:

```
write.csv2(d, "minedata.csv", row.names=FALSE)
```

Den resulterende tekstfil `minedata.csv` kan importeres i dansk OpenOffice eller Excel med Filer || Åben || Filtype alle filer.

For at skrive i amerikansk kommasepareret format som i figur 21 (c), skal man angive komma (`,`) som separator; decimalpunktum (`.`) i tal er standard:

```
write.table(d, "output.csv", sep=",", quote=FALSE, row.names=FALSE)
```

hvilket svarer til:

```
write.csv(d, "output.csv", row.names=FALSE)
```

Den resulterende tekstfil `output.csv` kan åbnes direkte i (dansk) Excel ved at dobbeltklikke på den, og den kan importeres i amerikansk OpenOffice.

Bemærk, at hvis man udelader `row.names=FALSE` til de to funktioner `write.csv` og `write.csv2` vil der komme en ekstra kolonne med rækkenumrene, hvilket fint kan indlæses i regnearket men måske er lidt redundant.

10.6 Overblik over et datasæt

Funktionen `summary` beregner mindste værdi, største værdi, middelværdi, kvartiler og andre oplysninger om et datasæt. Hvis `d` er datasættet indlæst i afsnit 10.1.1, giver `summary` dette resultat:

```
> summary(d)
      Tid            Kontrol         Vaekst
 Min.   :  30.0   Min.   :5.430   Min.   :5.310
 1st Qu.: 101.2   1st Qu.:5.485   1st Qu.:5.330
 Median : 225.0   Median :5.735   Median :5.345
 Mean   : 287.2   Mean   :5.776   Mean   :5.478
 3rd Qu.: 367.5   3rd Qu.:5.990   3rd Qu.:5.560
 Max.   :1440.0   Max.   :6.450   Max.   :6.070
```

Når du har indlæst et datasæt er det en god idé at bruge `summary` til at undersøge om det ser rimeligt ud. Desuden kan du bruge oplysningerne om mindste og største værdi til at vælge aksegrænser ud fra når du plotter datasættet; se afsnit 5.2.

Hvis væksthormonforsøgsdatasættet havde været indlæst forkert således at søjlerne `Kontrol` og `Vaekst` var blevet opfattet som tekst i stedet for tal (se afsnit 10.1.1) ville `summary` afsløre det med det samme:

```
> summary(fejl)
      Tid            Kontrol       Vaekst
 Min.   :  30.0   5,43   : 2   5,34   :4
 1st Qu.: 101.2   5,44   : 1   5,31   :2
 Median : 225.0   5,45   : 1   5,32   :2
 Mean   : 287.2   5,47   : 1   5,33   :2
 3rd Qu.: 367.5   5,49   : 1   5,35   :1
 Max.   :1440.0   5,52   : 1   5,37   :1
                  (Other):13   (Other):8
```

Bemærk at de to forkerte søjler ikke vises med de sædvanlige kvartiler.

Summary fortæller ikke hvor mange rækker, der er i det indlæste datasæt. For at få denne oplysning kan man bruge funktionen `nrow`:

```
> nrow(d)
[1] 20
```

Tilsvarende kunne man bruge funktionen `ncol` til at få oplyst antal søjler. Både `nrow` og `ncol` virker også for matricer (afsnit 16).

Som supplement til at kontrollere en indlæsning med `summary` kan man ønske at se de første eller sidste rækker i datasættet. Funktionerne `head` og `tail` kan bruges til dette. Som standard viser funktionerne de første hhv. sidste 6 rækker, men dette antal kan ændres:

```
> head(d)
  Tid Kontrol Vaekst
1  30    6.45   6.07
2  45    6.32   5.94
3  60    6.17   5.82
4  75    6.06   5.72
5  90    6.02   5.62
6 105    5.98   5.54
> tail(d, 3)
    Tid Kontrol Vaekst
18  450    5.44   5.33
19  480    5.43   5.34
20 1440    5.43   5.34
```

Det skal bemærkes at såvel `summary` som `head` og `tail` også virker på vektorer (afsnit 14) og matricer (afsnit 16).

10.7 Regning på værdier fra datasæt

De enkelte søjler i et datasæt fås ved at skrive datasættets og variablens navn adskilt af dollartegn (**$**):

```
> d$Kontrol
 [1] 6.45 6.32 6.17 6.06 6.02 5.98 5.94 5.85 5.80 5.76 5.71 5.63 5.59
[14] 5.52 5.49 5.47 5.45 5.44 5.43 5.43
```

Klammerne `[1]` og `[14]` i starten af første henholdsvis anden linje angiver at linjens første tal er observation nummer 1 henholdsvis 14.

En søjle i et datasæt er et eksempel på en *vektor* i R. Meget mere om vektorer og hvordan man kan udtage elementer fra dem og regne på dem følger i afsnit 14; her skal blot nævnes nogle få muligheder.

Man kan tage et enkelt tal ud fra rækken af observationer for en variabel, for eksempel værdi nummer 16 af variablen `Kontrol`:

```
> d$Kontrol[16]
[1] 5.47
```

Man kan bruge simple statistiske funktioner som `min` og `max` fra afsnit 11:

```
> min(d$Kontrol)
[1] 5.43
> max(d$Kontrol)
[1] 6.45
```

Man kan bruge matematiske funktioner til at *transformere* variable, for eksempel tage logaritmen:

```
> log(d$Kontrol)
 [1] 1.864080 1.843719 1.819699 1.801710 1.795087 1.788421 1.781709
 [8] 1.766442 1.757858 1.750937 1.742219 1.728109 1.720979 1.708378
[15] 1.702928 1.699279 1.695616 1.693779 1.691939 1.691939
```

Man kan tilføje en transformeret variabel som ny søjle i datasættet:

```
> d$LogK <- log(d$Kontrol)
> d$LogV <- log(d$Vaekst)
> d
    Tid Kontrol Vaekst     LogK     LogV
1    30    6.45   6.07 1.864080 1.803359
2    45    6.32   5.94 1.843719 1.781709
...
19  480    5.43   5.34 1.691939 1.675226
20 1440    5.43   5.34 1.691939 1.675226
```

Hvis man skal bruge en søjle fra et datasæt ofte kan man naturligvis binde den til en variabel for at undgå at skulle skrive lange udtryk som **`d$Kontrol`** hele tiden:

```
> K <- d$Kontrol
> K
 [1] 6.45 6.32 6.17 6.06 6.02 5.98 5.94 5.85 5.80 5.76 5.71 5.63 5.59
[14] 5.52 5.49 5.47 5.45 5.44 5.43 5.43
> K[16]
[1] 5.47
> min(K)
[1] 5.43
```

Man kan faktisk oprette variable med kopier af alle søjler fra et datasæt med funktionen `attach`. Efter en `attach` af datasættet `d` kan vi således referere til `Tid`, `Kontrol`, `Vaekst`, `LogK` og `LogV` uden at skulle skrive **`d$`** foran:

```
> attach(d)
> Kontrol
 [1] 6.45 6.32 6.17 6.06 6.02 5.98 5.94 5.85 5.80 5.76 5.71 5.63 5.59
[14] 5.52 5.49 5.47 5.45 5.44 5.43 5.43
```

Vi kan fjerne variablene igen med funktionen `detach`:

```
> detach(d)
> Kontrol
Error: object 'Kontrol' not found
```

OBS: Variable tilføjet med `attach` kan ikke ændres med den normale tildelingspil `<-`; hvis man prøver bliver der i stedet oprettet en ny variabel med samme navn (og den forsvinder ikke når man bruger `detach`). Man kan dog bruge den specielle tildelingsoperator `<<-` hvis man *vil* ændre en variabel tilføjet med `attach`. Variable, der er oprettet med almindelig tildeling vil "skygge" for variable tilføjet med `attach`, uanset om variablen oprettes før eller efter man bruger `attach`. Man vil dog få en advarsel hvis nogle af de variable der tilføjes med `attach` allerede er definerede og de tidligere definitioner dermed "skygger" for variablene fra datasættet:

```
> Kontrol <- 17
> attach(d)

        The following object(s) are masked _by_ .GlobalEnv :

         Kontrol

> Kontrol
[1] 17
```

Endelig skal det bemærkes at variablene oprettet med `attach` som sagt er *kopier* af søjlerne fra datasættet – ingen ændringer af dem (fx med `<<-`) vil ændre i det oprindelige datasæt.

11 Simple statistiske funktioner

Figur 23 er en oversigt over nogle simple statistiske funktioner; mere avancerede statistiske funktioner i R forklares i fakultetets statistikkurser. Mange af funktionerne, fx `max`, `mean`, `min`, `sd`, og `var`, tager en *vektor* som argument. Vektorer beskrives nærmere i afsnit 14, men som allerede nævnt er en søjle fra et datasæt en vektor, så for eksempel kan man finde middelværdien (gennemsnittet) af de målte pH værdier for kontrolgruppen i væksthormonforsøgsdatasættet som følger:

```
> mean(d$Kontrol)
[1] 5.7755
```

De fleste af funktionerne i tabellen, til og med `prod`, virker faktisk ikke alene med en vektor som argument, men også med en matrix (afsnit 16) eller med et datasæt. I sidste tilfælde får man endda for nogle af funktionerne (`mean`, `sd`, `summary`) en værdi for hver søjle:

```
> mean(d)
     Tid  Kontrol   Vaekst
287.2500   5.7755   5.4780
```

Funktionerne `max`, `min`, `range`, `sum` og `prod` kan man også kalde med flere vektorer (eller matricer eller datasæt) som argumenter; det virker som om alle tallene blev kombineret i én vektor, der så bruges som argument. For eksempel kan vi finde den maksimale målte pH-værdi i væksthormonforsøget som følger:

```
> max(d$Kontrol, d$Vaekst)
[1] 6.45
```

Hvis vi i stedet bruger `range` får vi både minimum og maksimum af de målte pH-værdier:

```
> range(d$Kontrol, d$Vaekst)
[1] 5.31 6.45
```

(se afsnit 12.1 for et eksempel på udnyttelse af dette).

Funktionerne `runif` og `rnorm`, der står for "random uniform" (ligefordeling) og "random normal" (normalfordeling), bruges til at generere pseudotilfældige tal. Dette er især nyttigt til simulering af biologiske eller økonomiske processer, se fx afsnit 15.2.

Funktion	Resultat
`mean(v)`	Middelværdi af elementerne i `v`
`median(v)`	Medianen af elementerne i `v`
`sd(v)`	Standardafvigelse af elementerne i `v`
`var(v)`	Varians af elementerne i `v`
`summary(d)`	Min, max, middelværdi og kvartiler i `d` (afsnit 10.6)
`max(v)`	Maksimum af elementerne i `v`
`min(v)`	Minimum af elementerne i `v`
`range(v)`	Svarer til `c(min(v), max(v))`
`sum(v)`	Summen af elementerne i `v`
`prod(v)`	Produktet af elementerne i `v`
`runif(n)`	Vektor med `n` ligefordelte tilfældige tal i $]0,1[$
`runif(n,min=1,max=4)`	Vektor med `n` ligefordelte tilfældige tal i $[1,4[$
`rnorm(n)`	Vektor med `n` tilfældige tal, normalfordelt $N(0,1)$
`rnorm(n,m,s)`	Vektor med `n` tilfældige tal, normalfordelt N(`m`,`s`)
`lm(`*formel*`, data=`*datasæt*`)`	Tilpasning af lineær model, lineær regression (afsnit 13)

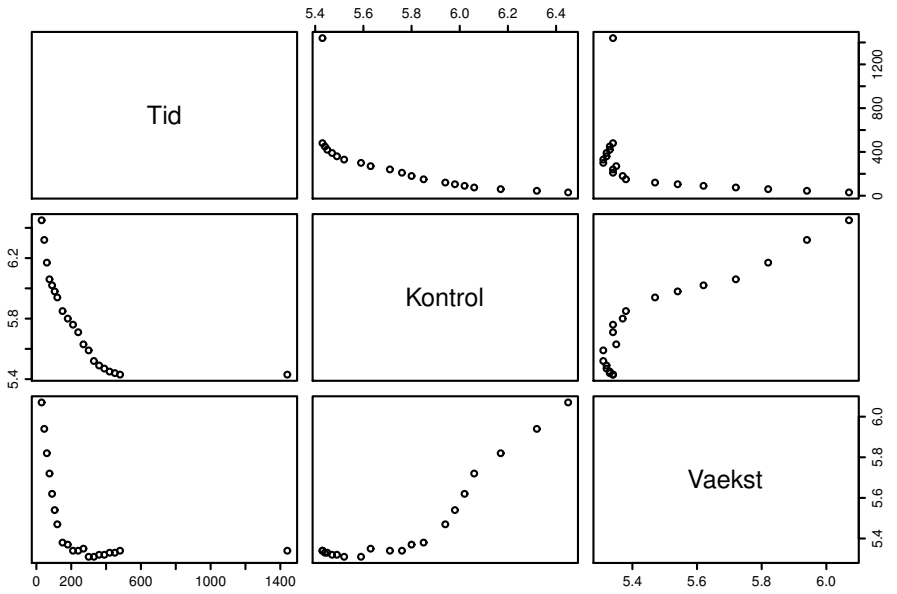
Figur 23: Nogle simple statistiske funktioner i R. Notationen N(`m`,`s`) står for normalfordelingen med middelværdi `m` og standardafvigelse `s`.

12 XY-plot

Man kan plotte observationerne fra datasættet `d` indlæst ovenfor på talrige måder. Først kan man danne sig et overblik over sammenhænge i datasættet ved at plotte alle variable mod hinanden på én gang. Det gøres ved simpelthen at skrive:

```
plot(d)
```

I dette tilfælde er der tre variable `Tid`, `Kontrol`, `Vaekst` hvilket giver $3 \cdot 2 = 6$ små XY-plots, som vist i figur 24.



Figur 24: Overbliksplot af datasæt med tre variable `Tid`, `Kontrol` og `Vaekst`.

Lige i dette tilfælde er det ikke så nyttigt, men generelt er det en god måde at få en idé om nogle af variablene for eksempel er lineært korrelerede, hvilket vil vise sig som en tilnærmelsesvis ret linje.

12.1 Plot af dataserier

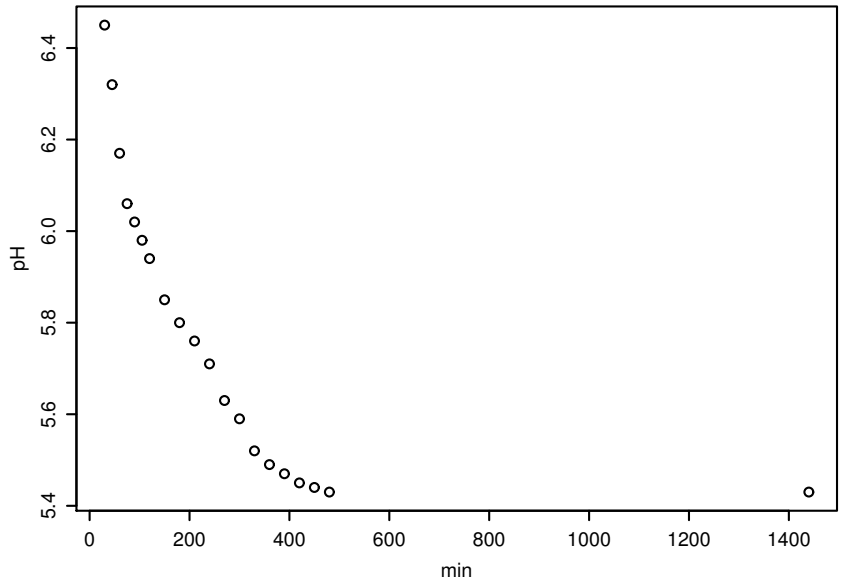
I det foreliggende datasæt er det mere nærliggende at plotte både kontrolgruppens pH `d$Kontrol` og væksthormongruppens pH `d$Vaekst` som funktion af tiden, så man kan sammenligne de to forløb.

Til at begynde med plotter vi kontrolgruppens pH som funktion af tiden med funktionen `plot`:

```
plot(d$Tid, d$Kontrol, xlab="min", ylab="pH")
```

Som standard vises datapunkterne i plottet uden at der trækkes linjer mellem dem; andre plottyper kan man få ved at angive parameteren `type`, se figur 29. Ovenfor er der brugt `xlab` og `ylab` (se afsnit 5.3) til at bestemme aksetitler. Det resulterende plot ses i figur 25.

Hvis man vil plotte både `Kontrol` og `Vaekst` som funktion af `Tid`, så skal man plotte den første dataserie med funktionen `plot` (som ovenfor), og derefter tilføje den anden dataserie med funktionen `points`. Når man skal tilføje nye datapunkter eller XY-punktkurver til et eksisterende plot skal man nemlig bruge funktionerne `points` og/eller `lines`. Man kan *ikke* bruge `plot(..., add=TRUE)`; det duer kun når man skal tilføje en *funktionsgraf* som i afsnit 5.1.



Figur 25: XY-plot af forsøgsdata: én dataserie tegnet med `plot`.

Funktionen `points` tager en vektor af x-værdier og en vektor af y-værdier og tilføjer datapunkterne (x, y) til et eksisterende plot. For eksempel kan vi tilføje et plot af `d$Vaekst` til et eksisterende plot af `d$Kontrol` sådan her:

```
plot(d$Tid, d$Kontrol, ylim=c(5,7), xlab="min", ylab="pH")
points(d$Tid, d$Vaekst, pch=4)
```

Her betyder parameteren `ylim=c(5,7)` at y-aksen skal gå fra 5 til 7, så begge kurver kan være der (se afsnit 5.2), mens `pch=4` bruges til at vælge plotsymbol (se afsnit 12.2). Resultatet ses i figur 26.

Hvis vi i stedet ville have andenaksen til kun at omfatte præcis samme interval som målingernes pH-værdier, kan vi som `ylim` angive det præcise interval, fundet med `range` funktionen fra afsnit 11:

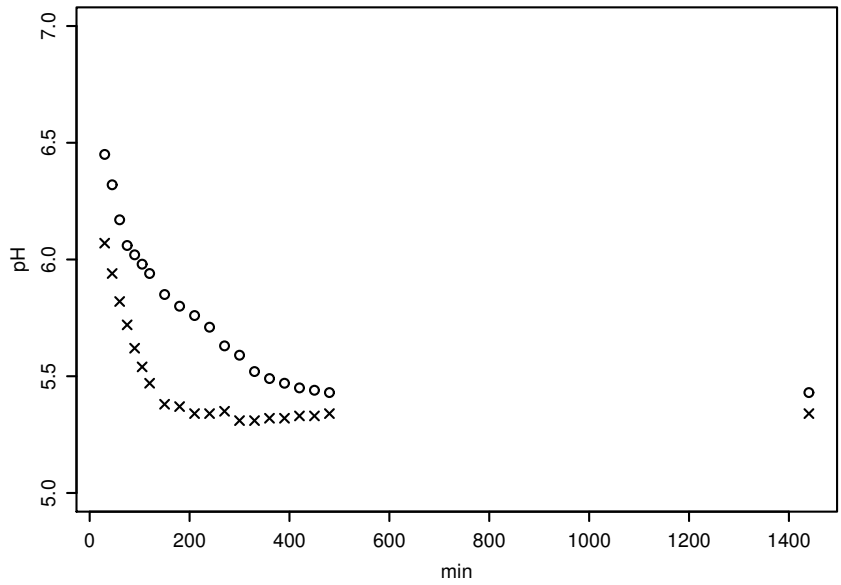
```
plot(d$Tid, d$Kontrol, ylim=range(d$Kontrol, d$Vaekst),
     xlab="min", ylab="pH")
points(d$Tid, d$Vaekst, pch=4)
```

Resultatet ses i figur 27. Hvis vi ikke var interesserede i de yderste punkter til højre (hvor `min` er 1440) kan vi specificere førsteaksens udstrækning med `xlim`:

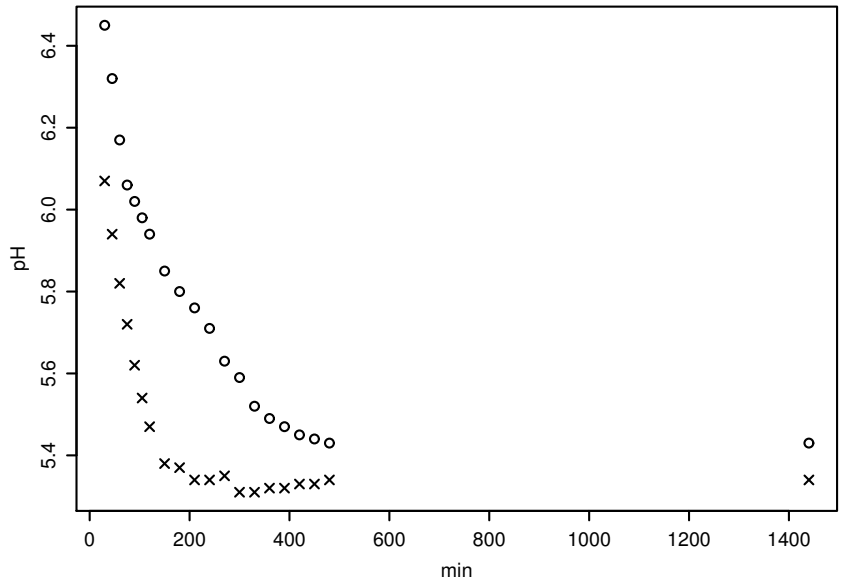
```
plot(d$Tid, d$Kontrol, xlim=c(0,500), ylim=range(d$Kontrol, d$Vaekst),
     xlab="min", ylab="pH")
points(d$Tid, d$Vaekst, pch=4)
```

Resultatet ses i figur 28.

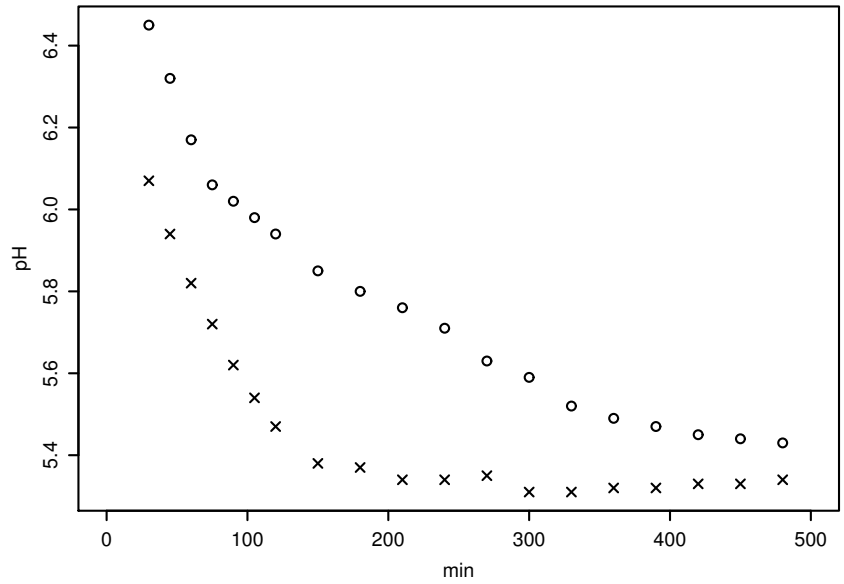
Funktionen `points(xs, ys, ...)` virker grundlæggende som `plot(xs, ys, ...)`, bortset fra at den ikke starter et nyt plot, men tegner videre på det aktuelle plot. De samme parametre bruges til at styre udseende af plotsymboler og eventuelle forbindelseslinjer i både `plot` og `points`. Parameteren `type` vælger plottype, parameteren `pch`, der står for "plotting character" kan bruges til at vælge plotsymbol, parameteren `col` kan bruges til at vælge plotsymbolernes (og linjernes) farve, og parameteren `cex` bruges til at vælge plotsymbolernes størrelse; for eksempel betyder `cex=3` tredobling af standardstørrelsen, og `cex=0.5` betyder halvering. Hvis plottypen har linjer mellem punkterne bestemmer parametrene `lty` og `lwd` henholdsvis linjernes type og tykkelse ganske som for funktionsgrafer, se figur 10.



Figur 26: XY-plot af forsøgsdata: to dataserier tegnet med `plot` og `points`, med `ylim=c(5,7)`.



Figur 27: Som figur 26, men med `ylim=range(d$Kontrol, d$Vaekst)`.



Figur 28: Som figur 27, men med `xlim=c(0, 500)`.

Til forskel fra `plot` kan man ikke i `points` regulere titler, akser og udstrækningen af koordinatsystemet, ganske som man ikke kan det for funktionsgrafer man tilføjer til et eksisterende plot med `add=TRUE`. Det vil sige at man lige som for funktionsgrafer skal sørge for at koordinatområde osv. sættes korrekt op i det første kald af `plot`, så datapunkter der efterfølgende tilføjes med `points` ikke falder udenfor.

Den generelle procedure for at plotte flere dataserier sammen er altså:

1. Plot den første dataserie med `plot`. Sørg for at sætte koordinatområde, aksetitler, akser osv. korrekt.

2. Tilføj følgende dataserier med `points`.

3. Tilføj eventuelle funktionsgrafer med `plot(...,add=TRUE)` og regressionslinjer med `abline`, se afsnit 13.

4. Tilføj eventuelle forklarende tekster og signaturforklaring.

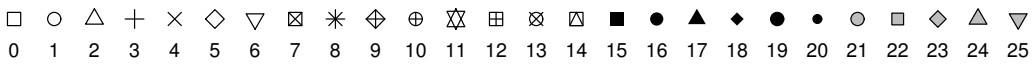
Funktionen `lines` er fuldstændig analog med `points` bortset fra at standardværdien for plottypen er `type="l"`, således at der (med mindre man angiver en anden `type`) tegnes uden plotsymboler og med linjer mellem datapunkterne. Hvis man angiver parameteren `type` er der ingen forskel på `points` og `lines`.

12.2 Plottype og plotsymboler

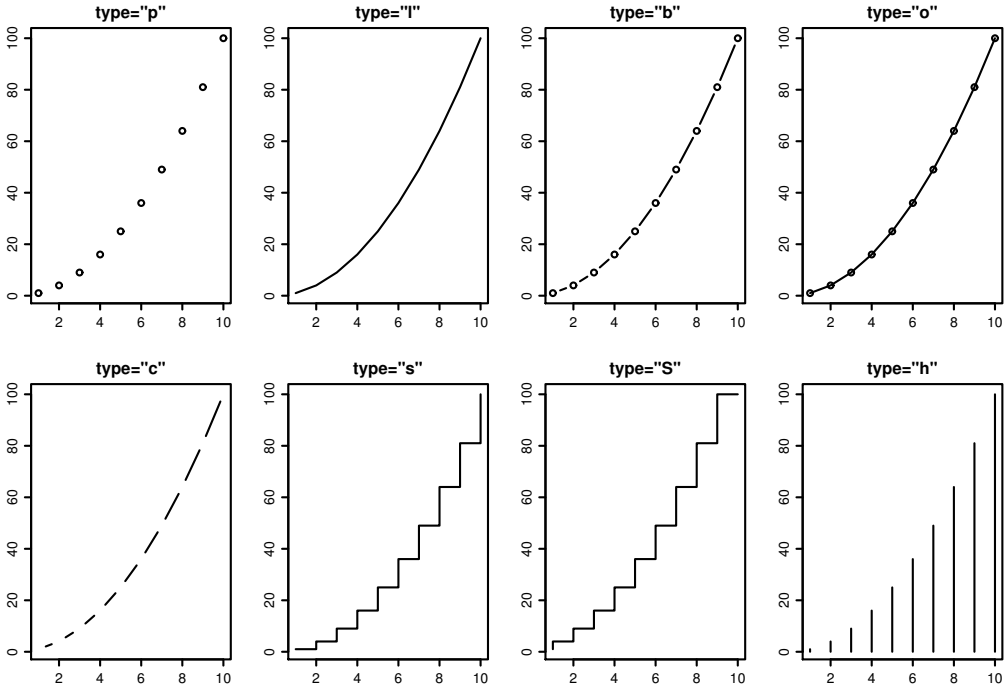
De fleste af `plot`-parametrene beskrevet i afsnit 5 kan anvendes også på XY-plot. Desuden kan man styre XY-plottets type (punkter, linjer, begge dele, trinplot, histogram) med parameteren `type`, og man kan styre hvilket symbol der bruges til at plotte datapunkter med parameteren `pch`. Se tabellen i figur 29. Endvidere kan funktionen `locator` (se afsnit 5.8) bruges til at identificere datapunkter i et XY-plot ligesom i et funktionsplot.

Parameter	Effekt	Mulige værdier
`cex`	symbolskalering	Tal, fx giver `cex=2` fordobling, `cex=0.5` halvering
`pch`	plotsymbol	`pch=0` eller `1`, `2`, ..., `25`; se figur 30 på side 48.
`type`	plottype	`type="p"` eller `"l"`, `"b"`, `"o"`, `"c"`, `"s"`, `"S"`, `"h"`; se figur 31

Figur 29: Nogle grafikparametre til brug i XY-`plot`, `points` og `lines`.



Figur 30: Plotsymboler til `plot` og `points`. For eksempel giver `pch=2` en åben trekant. "Fyldfarven" i symbolerne nummer 21 til 25 sættes med parameteren `bg`, fx som her `bg="grey"`.



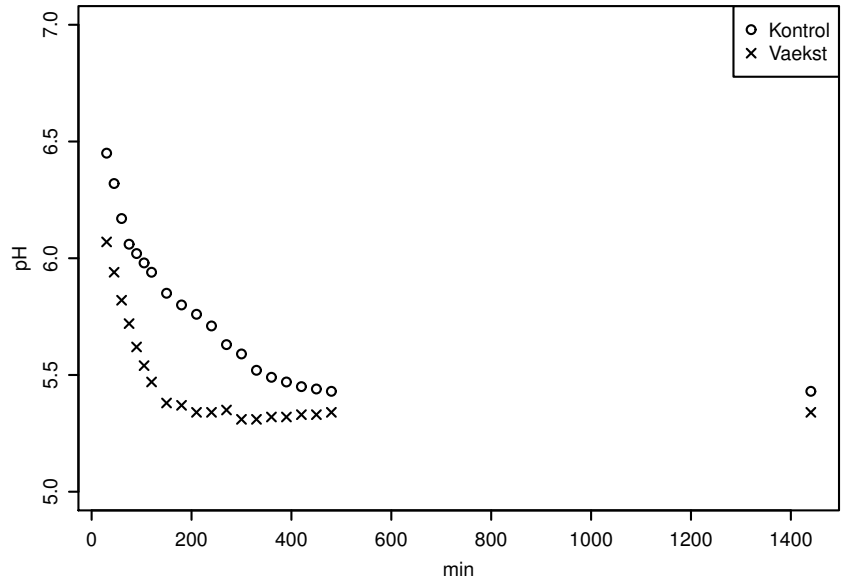
Figur 31: Plottyper: punkter, linjer, to kombinationer, linjestykker, to slags trinplot, og histogram.

12.3 Signaturforklaring i XY-plot

Funktionen `legend` som blev beskrevet i afsnit 5.7 kan også tegne signaturforklaringer med plotsymboler. Dette gøres ved at angive parameter `pch` og en liste af plotsymboler pakket ind i `c(...)`, analogt med måden man i `legend` angiver farver, linjetyper og linjetykkelser. Eksempel:

```
plot(d$Tid, d$Kontrol, ylim=c(5,7), xlab="min", ylab="pH")
points(d$Tid, d$Vaekst, pch=4)
legend("topright", c("Kontrol","Vaekst"), pch=c(1,4))
```

Resultatet ses i figur 32. Bemærk i eksemplet at der ikke er angivet nogen værdi for `pch` til `plot`; den første dataserie tegner R derfor med standard plotsymbol svarende til `pch=1` og derfor skal den første `pch`-værdi i `legend` være 1.



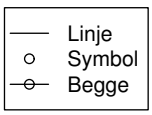
Figur 32: XY-plot af forsøgsdata med signaturforklaring tegnet med `legend`.

Hvis man har skaleret plotsymboler i XY-plottet med parameteren `cex` bør man foretage den samme skalering i signaturforklaringen. Her hedder den tilsvarende parameter imidlertid ikke `cex` men derimod `pt.cex`.

Man kan have brug for at tegne en signaturforklaring med både plotsymboler og linjer og dette gøres ved at angive både `pch` og `lty`/`lwd` parametre til `legend`. Hvis en signatur kun skal have plotsymbol (og ingen linje) angives `"blank"` på den tilsvarende plads i `lty` listen og hvis omvendt en signatur alene er en linje angives -1 på den tilsvarende plads i `pch` listen. Eksempel:

```
legend(x, y, c("Linje","Symbol","Begge"),
       lty=c("solid","blank","solid"), pch=c(-1,1,1))
```

Her har vi delt den lange indtastning over to linjer. Dette giver denne signaturforklaring:



12.4 Eksempel på et matematisk XY-plot

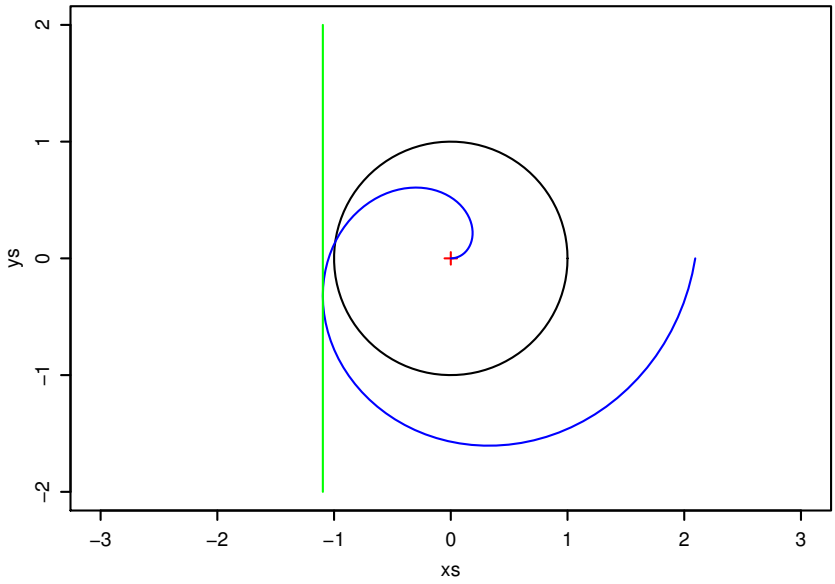
Dette afsnit bruger funktioner og begreber fra afsnit 14 nedenfor til at lave et "matematisk" plot ved hjælp af funktionerne `plot`, `lines` og `points`. For eksempel kan man tegne enhedscirklen, en spiral, og en lodret tangent til spiralen som vist i figur 33. Vi begynder med at lave en vektor `t` med 100 vinkler fra 0 til 2π med funktionen `seq` fra afsnit 14.1:

```
t <- seq(0, 2*pi, len=100)
```

Dernæst beregner man vektorer af enhedscirklens x- og y-koordinater ud fra vinklerne, og plotter med `type="l"` de linjer der forbinder disse (x, y)-koordinater:

```
x <- cos(t)
y <- sin(t)
plot(x, y, type="l", asp=1, xlim=c(-2,2), ylim=c(-2,2))
```

Resultatet er den sorte cirkel midt i figur 33. Parameteren `asp=1`, der står for "aspect ratio" betyder at en y-akseenhed skal være lige så lang som en x-akseenhed i plottet. Hvis ikke de er det, så ligner cirklen en ellipse i stedet.



Figur 33: Enhedscirkel, nulpunkt (rødt kryds), spiral (blå) og lodret tangentlinje til spiralen (grøn).

Man kan bruge `points` til at indsætte enkeltsymboler i plottet, fx til at sætte plotsymbol 3, dvs. et kryds, i origo $(0, 0)$ (med rødt):

```
points(0, 0, pch=3, col="red")
```

Hvis man ganger x- og y-koordinaterne for cirklen med en stigende sekvens af tal, fx vinklerne `t`, så får man en spiral. Vi kan bruge funktionen `lines` til at tilføje en blå spiral til det eksisterende plot, idet vi dividerer begge koordinater med konstanten 3 for at spiralen bliver inden for plottet:

```
lines(x * t/3, y * t/3, col="blue")
```

Nu kunne vi ønske at tegne den lodrette tangent til spiralen gennem det punkt hvis koordinater er omtrent $(-1.1, -0.3)$. For at finde en mere præcis værdi for tangentens x-koordinat skal vi bruge at spiralens x-koordinat som funktion af vinklen t er $x(t) = t\cos(t)/3$ så dens afledede er $x'(t) = cos(t)/3 - t\sin(t)/3$. Til at finde den værdi t_0 mellem 1 og 5 hvor spiralens tangent er lodret, kan vi bruge `uniroot` fra

afsnit 7 til at finde det t_0 der giver $x'(t_0) = 0$:

```
> dx <- function(t) { cos(t)/3 - t*sin(t)/3 }
> t0 <- uniroot(dx, c(1,5)) $ root
> t0
[1] 3.425601
```

Så kan vi ud fra t_0 beregne x-koordinaten x_0 for den lodrette tangent til spiralen og tegne tangentlinjen med grønt, som et linjestykke fra $(x_0, -2)$ til $(x_0, 2)$:

```
> x0 <- cos(t0)*t0/3
> x0
[1] -1.096124
> lines(c(x0, x0), c(-2, 2), col="green")
```

13 Lineær regression og regressionskurve

Betragt et datasæt bestående af fem sammenhørende værdier af t og y:

t	1.0	1.5	2.0	2.5	3.0
y	1.4	0.3	−1.5	−3.1	−4.8

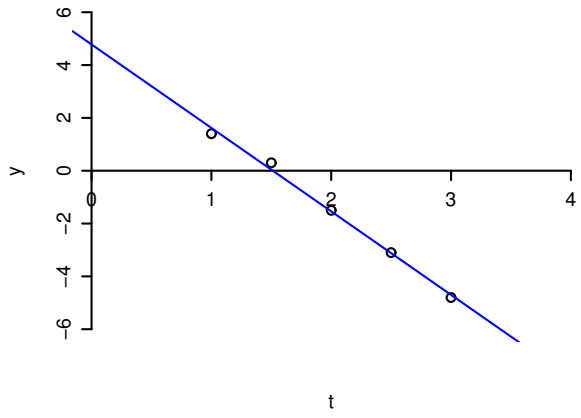
Hvis vi plotter disse fem punkter i et koordinatsystem falder de omtrent på en ret linje; se figur 34.

Lineær regression finder ligningen for den rette linje der bedst passer til punkterne. Hertil benyttes R-funktionen `lm`, der står for "linear model". Først definerer vi vektorerne `t` og `y`, og derefter kalder vi `lm` med en såkaldt R-formel `y ~ t` for at angive at vi vil modellere `y` som funktion af `t`.[9] Funktionen `lm` finder så koefficienterne b og a i modellen $y = b + at$:

```
> t <- c(1.0, 1.5, 2.0, 2.5, 3.0)
> y <- c(1.4, 0.3, -1.5, -3.1, -4.8)
> lreg <- lm(y ~ t)
> lreg$coefficients
(Intercept)           t
       4.78       -3.16
```

Den beregnede regressionslinje er $y = 4.78 - 3.16t$. Resultatet af **`lm(y ~ t)`**, gemt i variablen **`lreg`**, er en associationsliste (afsnit 20.1) hvis komponent **`coefficients`** er en vektor der indeholder koefficienterne b og a. Ved at skrive **`summary(lreg)`** kan man få mange flere oplysninger om regressionen.

Nu kan man først plotte datasættets observationer med funktionen `plot` og dernæst tilføje regressionslinjen til plottet med funktionen `abline`. Det resulterende plot ses i figur 34.



Figur 34: Datapunkter med regressionslinje.

```
plot(t, y, ylim=c(-6,6), xlim=c(0,4), axes=FALSE)
axis(1, pos=0)
axis(2, pos=0)
abline(lreg, col="blue")
```

Man kan også lave lineær regression på eksisterende datasæt, som for eksempel vækshormonforsøgsdatasættet `d` fra afsnit 10.1. Lad os beregne den lineære regressionskurve for kontrolgruppens pH

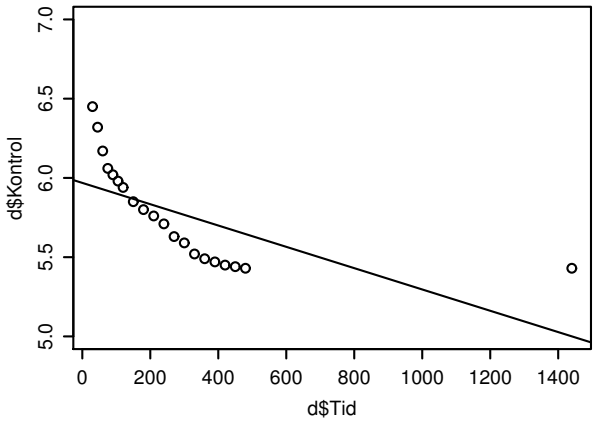
[9] Tegnet "~" får man på såvel PC som Mac med tasten umiddelbart til højre for Å, hvor også "hatten" ^ sidder. På PC taster man AltGr sammen med tasten, på Mac ⌥. I begge tilfælde skal man taste mellemrum umiddelbart bagefter da ~ er en såkaldt "død tast".

(variablen `Kontrol`) som funktion af tiden (variablen `Tid`). Vi kalder derfor funktionen `lm` med formlen `Kontrol ~ Tid` og skriver `data=d` fordi variablene skal tages fra datasættet `d`:

```
> model <- lm(Kontrol ~ Tid, data=d)
> model$coefficients
(Intercept)         Tid
  5.9686648   -0.0006725
```

Den beregnede regressionslinje er $y = 5.9686648 - 0.0006725x$. Nu kan man først plotte datasættets observationer med funktionen `plot` og dernæst tilføje regressionslinjen til plottet med funktionen `abline`; det giver datapunkterne og den rette linje i figur 35:

```
plot(d$Tid, d$Kontrol, ylim=c(5, 7))
abline(model)
```



Figur 35: Forsøgsdata med regressionslinje.

13.1 Lineær regression på transformerede variable

Som det ses i figur 35 at observationerne dårligt forklares af en lineære model. I stedet kunne man forsøge med en anden type model, fx en potensmodel. Vi antager her, at kontrolgruppens pH, pH, (variablen `Kontrol`) er en potensfunktion af tiden, t, (variablen `Tid`), altså at sammenhængen er $pH = bt^a$, hvor a og b er konstanter. I denne model afhænger $\ln pH$ lineært af $\ln t$ idet vi har $\ln pH = \ln b + a \ln t$. Vi kan finde $\ln b$ og a ved en lineær regression på de *transformerede variable* $\ln pH$ og $\ln t$.

Når man skal regne på transformerede variable fra et datasæt er det nemmest at tilføje dem til datasættet som nye variable (søjler). Vi opretter de transformerede udgaver af `Kontrol` og `Tid` som følger:

```
d$logKontrol <- log(d$Kontrol)
d$logTid <- log(d$Tid)
```

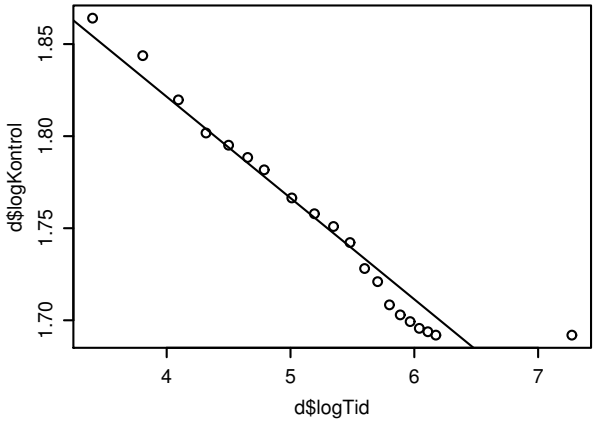
Vi kan nu bruge de transformerede søjler i en lineær model:

```
> llmodel <- lm(logKontrol ~ logTid, data=d)
> llmodel$coefficients
(Intercept)       logTid
    2.04147     -0.05502
```

Herefter kan vi lave et plot med de transformerede variable og den fundne regressionslinje:

```
plot(d$logTid, d$logKontrol)
abline(llmodel)
```

Resultatet ses i figur 36 (og det ses at heller ikke denne model stemmer særlig godt).



Figur 36: Log-transformerede forsøgsdata med regressionslinje.

Ud fra regressionslinjens skæring med andenaksen, dvs. tallet i `llmodel$coefficients[1]`, kan vi beregne konstanten b fra potensmodellen:

```
> exp(llmodel$coefficients[1])
(Intercept)
   7.701908
```

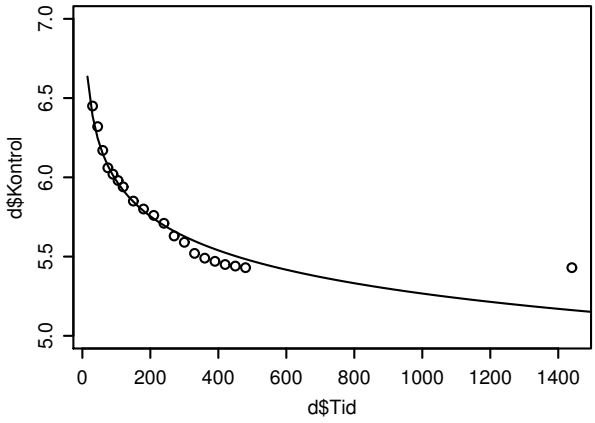
Konstanten a har vi direkte som linjens hældning, dvs. `llmodel$coefficients[2]`. Vi kan dermed som følger definere en funktion, der giver pH-værdien som fuktion af tiden efter den fundne model:

```
a <- llmodel$coefficients[2]
b <- exp(llmodel$coefficients[1])
pH <- function(t) { b*t^a }
```

Endelig kan vi lave et plot med de oprindelige, utransformerede forsøgsdata og den fundne model:

```
plot(d$Tid, d$Kontrol, ylim=c(5, 7))
plot(pH, 0, 1500, add=TRUE)
```

Resultatet kan ses i figur 37.



Figur 37: Forsøgsdata med den fundne potensmodel.

I appendiks C vises, hvordan man kan lave polynomiel regression på de samme data.

14 Vektorer

En *vektor* i R er en liste af elementer af samme type (tal, tekster, sandhedsværdier). Vi har allerede brugt vektorer adskillige steder, fx i afsnit 5.2 hvor vi angav `ylim` i et plot med vektoren `c(-16, 200)`. I det følgende vi blive gennemgået hvordan man opretter og regner med vektorer i R.

14.1 Oprettelse af vektorer

Man kan definere vektoren `v` til at være (3, 4), altså til have de to elementer 3 og 4, således:

```
> v <- c(3, 4)
```

Funktionen `c` betyder "combine" og bruges til at konstruere vektorværdier. Som altid kan man vise værdien af variablen `v` bare ved at skrive den:

```
> v
[1] 3 4
```

Med `c` kan man sammensætte både enkelte tal og vektorer (faktisk *er* et enkelt tal en vektor med kun ét element):

```
> c(v, v, 42)
[1]  3  4  3  4 42
```

Der er specielle funktioner til at lave regelmæssige talrækker hvilket man ofte har brug for. Man kan fx lave en vektor med elementerne 11 12 13 14 15 ved brug af funktionen `seq`, der er en forkortelse af "sequence":

```
> seq(11,15)
[1] 11 12 13 14 15
```

Hvis man bytter om på grænserne får man sekvensen i omvendt rækkefølge:

```
> seq(15,11)
[1] 15 14 13 12 11
```

Kolon-operatoren ":" er en kortere notation for den simple brug af `seq` som ovenfor:

```
> 11:15
[1] 11 12 13 14 15
> 15:11
[1] 15 14 13 12 11
```

Funktionen `seq` kan også bruges til at lave sekvenser med spring forskellige fra ± 1. Med parameteren `by` kan man angive springenes størrelse:

```
> seq(0, 1, by=0.1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(20, 0, by=-4)
[1] 20 16 12  8  4  0
```

Nogle gange kan det være besværligt at regne springenes størrelse ud, fx hvis man vil dele intervallet $[0, 2\pi]$ i lige store delintervaller og få en vektor med 10 elementer startende med 0 sluttende med 2π . Her kan man bruge parameteren `len`:

```
> s <- seq(0, 2*pi, len=10)
> s
[1] 0.0000000 0.6981317 1.3962634 2.0943951 2.7925268 3.4906585
[7] 4.1887902 4.8869219 5.5850536 6.2831853
```

Man kan også udelade endepunktet og kombinere `len` og `by`:

```
> seq(0, by=0.1, len=14)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3
```

Til at gentage elementer eller delsekvenser bruger man funktionen `rep` (for "repeat"). Man kan fx lave en vektor med 10 kopier af tallet 1:

```
> rep(1, 10)
[1] 1 1 1 1 1 1 1 1 1 1
```

Mere avanceret kan man bruge `rep` til at gentage hele vektorer på forskellig måde. Simpel gentagelse får man som følger:

```
> rep(c(3, 4), 5)
[1] 3 4 3 4 3 4 3 4 3 4
```

Man kan specificere gentagelse af de enkelte elementer frem for hele vektoren med parameteren `each`

```
> rep(c(3, 4), each=5)
 [1] 3 3 3 3 3 4 4 4 4 4
> rep(c(3, 4), 5, each=2)
 [1] 3 3 4 4 3 3 4 4 3 3 4 4 3 3 4 4 3 3 4 4
```

Endelig kan man specificere hvor mange gange hvert element skal gentages:

```
> rep(c(3, 4), c(5, 2))
[1] 3 3 3 3 3 4 4
```

Man kan kombinere `c`, `seq`/kolon-operatoren og `rep` for at skabe mere komplicerede sekvenser:

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

14.2 Indeksering i vektorer

Et indeksudtryk `v[i]` giver det `i`'te element af vektoren `v`. Lad os for eksempel definere en vektor `u` og og udtage dens tredje element:

```
> u <- c(7, 9, 13, 107, 109, 113)
> u
[1]   7   9  13 107 109 113
> u[3]
[1] 13
```

Man kan også indeksere en vektor med en vektor af indekser. På den måde kan vektor-elementer dubleres eller omordnes:

```
> u[c(3,4,4,1)]
[1]  13 107 107   7
```

Det er således nemt med en sekvens af indeksværdier at udtage en del-vektor, for eksempel de første 4 elementer:

```
> u[1:4]
[1]   7   9  13 107
```

Det har en speciel betydning hvis man anvender *negative* tal som indeks, det betyder at disse elementer skal *udelades*:

```
> u[-2]
[1]   7  13 107 109 113
> u[c(-3,-5)]
[1]   7   9 107 113
> u[-2:-4]
[1]   7 109 113
```

Man kan ændre elementer i en vektor ved tildeling, enten for et enkelt eller flere elementer ad gangen:

```
> w <- 1:10
> w
 [1]  1  2  3  4  5  6  7  8  9 10
> w[3] <- 17
> w
 [1]  1  2 17  4  5  6  7  8  9 10
> w[c(2,6,8)] <- 101:103
> w
 [1]   1 101  17   4   5 102   7 103   9  10
```

14.3 Regning med vektorer

Funktionerne `sum`, `mean`, `min` og `max` fra afsnit 11 beregner sum, gennemsnit, minimum og maksimum af elementerne i en vektor, her `v = c(3, 4)`:

```
> sum(v)
[1] 7
> mean(v)
[1] 3.5
> min(v)
[1] 3
> max(v)
[1] 4
```

Funktionen `summary` fra afsnit 10.6 kan også anvendes på vektorer:

```
> summary(v)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.00    3.25    3.50    3.50    3.75    4.00
```

Almindelige regneoperatorer kan også bruges på vektorer. Når man regner på to vektorer der har samme antal elementer anvendes regneoperatoren *elementvis*, og resultatet er en vektor med samme antal elementer:

```
> c(1,2,3) + c(7,9,13)
[1]  8 11 16
> c(1,2,3) * c(7,9,13)
[1]  7 18 39
```

Bemærk at ovenstående produkt (`*`) ikke beregner prikproduktet. Prikproduktet eller indre produkt eller af to vektorer med samme antal elementer beregnes som summen af de to vektorers elementvise produkter: [10]

```
> sum(c(1,2,3) * c(7,9,13))
[1] 64
```

[10] Alternativt kan prikproduktet beregnes som matrixprodukt, se afsnit 16.4.

Længden $|v|$ af en vektor kan beregnes som kvadratroden af prikproduktet af vektoren med sig selv, altså fx med `v = c(3, 4)` fra før:

```
> sqrt(sum(v * v))
[1] 5
```

Pas på: Udtrykket `length(v)` giver antallet af elementer i vektoren, *ikke* vektorens geometriske længde $|v|$:

```
> length(v)
[1] 2
```

To vektorer med forskelligt antal elementer kan bruges i samme regneudtryk:

```
> c(3, 4) + c(11, 13, 17, 19)
[1] 14 17 20 23
```

I R-systemet gælder der følgende *genbrugsregel* / *gentageregel*: Når to vektorer med forskelligt antal elementer bruges i samme regneudtryk, så genbruges elementerne fra vektoren med færrest elementer så antallet passer med den vektor der har flest elementer. Udtrykket ovenfor beregnes som om elementerne fra `c(3, 4)` var blevet gentaget, altså som dette udtryk:

```
> c(3, 4, 3, 4) + c(11, 13, 17, 19)
[1] 14 17 20 23
```

Bemærk at dette er helt anderledes end i matematik, hvor det ikke giver nogen mening at lægge en vektor der har to elementer sammen med en vektor der har fire elementer.

Et simpelt tal og en vektor kan også bruges i samme regneudtryk idet R opfatter et simpelt tal som en vektor af længde én. I overensstemmelse med genbrugsreglen vil det simple tal blive genbrugt for hvert element i vektoren. For eksempel kan man fordoble elementerne i en vektor sådan her:

```
> c(2, 3, 5, 7, 11) * 2
[1]  4  6 10 14 22
```

Dette regneudtryk betyder det samme som

```
> c(2, 3, 5, 7, 11) * c(2, 2, 2, 2, 2)
```

Svarende til de simple regneoperationer gælder at hvis man anvender en indbygget matematisk funktion af én variabel på en vektor, så anvendes funktionen elementvis:

```
> sqrt(c(49, 81, 169))
[1]  7  9 13
```

Det fungerer fordi funktionen `sqrt` forventer et enkelt tal som argument, og den eneste fornuftige måde at anvende den på en vektor af tal er at anvende den på hvert tal for sig. Det samme gælder de funktioner man selv definerer, hvis de kun anvender de indbyggede matematiske funktioner og almindelige regneoperatorer:

```
> cbrt <- function(x) { x^(1/3) }
> cbrt(c(8, 27, 64))
[1] 2 3 4
```

Hvis der er tvivl om hvordan funktionen virker på en vektor er det bedre udtrykkeligt at angive at funktionen skal anvendes på hvert element for sig. Hertil bruges R-funktionen `sapply`, der tager en vektor som første argument og en funktion som andet argument, anvender funktionen på hvert element af vektoren og producerer en vektor af resultaterne:

```
> sqrt(c(1, 4, 9))
[1] 1 2 3
> sapply(c(1, 4, 9), sqrt)
[1] 1 2 3
> sum(c(1, 4, 9))
[1] 14
> sapply(c(1, 4, 9), sum)
[1] 1 4 9
```

14.4 Vektorer af tekster

Nogle gange har man i R brug for at angive vektorer hvis elementer er tekster i stedet for tal. En del eksempler på dette så vi i forbindelse med signaturforklaringer i afsnit 5.7 og afsnit 12.3.

Vektorer af tekster angives med `c` som man forventer. Man kan også bruge `rep` med tekster:

```
> c("red", "green", "blue")
[1] "red"   "green" "blue"
> rep(c("red", "green", "blue"), 2)
[1] "red"   "green" "blue"  "red"   "green" "blue"
```

Det giver naturligvis ingen mening at regne på tekst-vektorer, men indeksering virker som forventet.

14.5 Vektorelementer med navne

Man kan knytte navne eller "overskrifter" til de enkelte elementer i en vektor. Nogle gange tildeler R endda automatisk navne til vektorelementer hvis vektoren for eksempel tages ud af en matrix med navne på søjler og rækker eller opbygges med funktioner som `cbind` og `rbind` (afsnit 16.3).

Oprettelse af en vektor `v` med to elementer med navnene "a" hhv. "b":

```
> v <- c(a=1,b=7)
> v
a b
1 7
```

Navnene vises altså som overskrifter. De følger med hvis vi gør noget med vektoren:

```
> 2*v
 a  b
 2 14
> c(v,5)
a b
1 7 5
```

Man kan få navnene ud som en tekstvektor med funktionen `names`:

```
> names(v)
[1] "a" "b"
```

Man kan endda ændre alle navnene eller bare et enkelt navn med brug af tildeling sammen med `names`:

```
> names(v) <- c("x1","x2")
> v
x1 x2
 1  7
> names(v)[1] <- "A"
> v
 A x2
 1  7
```

Man kan bruge navnene til indeksering i vektoren:

```
> v["x2"]
x2
 7
```

Endelig kan man fjerne navnene ved at sætte dem til `NULL`, der betyder "ingenting":

```
> names(v) <- NULL
> v
[1] 1 7
```

15 `For`-løkker

En `for`-løkke udfører den samme beregning én gang for hvert element i en vektor[11] efter tur. Dette er forskelligt fra fx `sapply` (afsnit 14.3) som udfører den samme operation på hvert element i en vektor "på en gang" og giver en vektor med resultaterne.

En `for`-løkke har følgende form:

`for (` *variabel* `in` *udtryk*$_1$ `)` *udtryk*$_2$

For-løkken evalueres ved at hvert element i vektoren *udtryk*$_1$ bindes efter tur til variablen *variabel* og *udtryk*$_2$ evalueres med den værdi af *variabel*. Efter løkken beholder variablen *variabel* den sidste værdi, den havde i løkken. Udtrykket *udtryk*$_2$ kaldes løkkens *krop*. Et udtryk i R kan, som vi skal se nærmere på i afsnit 17, være mange forskellige ting, for eksempel en sekvens af udtryk eller tildelinger til variable, så man kan lave vilkårligt komplekse ting i kroppen af en `for`-løkke. For eksempel er en `for`-løkke i sig selv et udtryk så man kan endda have flere løkker inde i hinanden. I eksemplerne i disse noter vil vi konsekvent skrive løkkekroppe med krøllede parenteser `{ ... }` omkring (forklaring på denne notation kommer i afsnit 17.3).

Den mest almindelige brug af en `for`-løkke er til trinvis beregning. Som et simpelt eksempel kan vi bruge en `for`-løkke til at udregne 6 fakultet, altså produktet $6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6$:

```
> y <- 1
> for (x in 2:6) { y <- y*x }
> y
[1] 720
> x
[1] 6
```

Ovenstående virker som følger: Vi vil gemme resultatet af beregningen i variablen `y`. Derfor giver vi først `y` værdien 1 inden løkken. Så bruger vi `for`-løkken til at løbe gennem `x`-værdierne 2, 3, ..., 6 og for hver af disse værdier af `x` ændre værdien af `y` til at være den "gamle" værdi af `y` multipliceret med `x`. Hvis vi skrev de samme operationer uden en `for`-løkke ville det se således ud:

```
> y <- 1
> x <- 2
> y <- y*x
> x <- 3
> y <- y*x
> x <- 4
> y <- y*x
> x <- 5
> y <- y*x
> x <- 6
> y <- y*x
> y
[1] 720
> x
[1] 6
```

Det er ikke altid at man overhovedet bruger selve værdierne fra vektoren til noget – det kan være vi bare ønsker at gentage en operation et vist antal gange.

[11] `For`-løkker virker faktisk ikke kun for vektorer men også for associationslister (og dermed datasæt), se afsnit 20.

15.1 Fejlfinding i `for`-løkker

Da man ikke kan køre en løkke "trin for trin" og inspicere indholdet af variable undervejs kan det være svært at gennemskue, hvad der er galt hvis løkken ikke giver det forventede resultat. Man kan (og bør i første omgang) forsøge at *håndkøre* løkken i det mindste for de første få gennemløb (dvs. for de første få udførelser af løkkekroppen). En håndkørsel vil sige at man sætter sig med papir og blyant og "leger R", dvs. eftergør hvordan R udfører de enkelte instruktioner og opskriver hvordan variablenes værdier ændres undervejs. For løkken der beregner 6 fakultet kunne det se sådan ud:

Selve koden:

```
y <- 1
for (x in 2:6) { y <- y*x }
```

Håndkørsel:

`y<-1`	før løkken er $y = 1$
Første gennemløb af `for(x in 2:6)`	nu er $x = 2$
`y<-y*x`	nu bliver $y = 1 \cdot 2 = 2$
Andet gennemløb af `for(x in 2:6)`	nu er $x = 3$
`y<-y*x`	nu bliver $y = 2 \cdot 3 = 6$
Tredje gennemløb af `for(x in 2:6)`	nu er $x = 4$
`y<-y*x`	nu bliver $y = 6 \cdot 4 = 24$
Fjerde gennemløb af `for(x in 2:6)`	nu er $x = 5$
`y<-y*x`	nu bliver $y = 24 \cdot 5 = 120$
Femte gennemløb af `for(x in 2:6)`	nu er $x = 6$
`y<-y*x`	nu bliver $y = 120 \cdot 6 = 720$

Ved at håndkøre sikrer man at man virkelig forstår, hvad der sker i løkken, og dermed finder man forhåbentlig ud af hvad der er galt.

En anden teknik til fejlfinding er at indsætte kald af funktionen `print` i løkken. Denne funktion skriver sit argument ud i konsollen:

```
> print(19)
[1] 19
```

Hvis man indsætter `print` af passende mellemregninger og/eller variable i løkkekroppen får man skrevet værdierne af disse ud for hvert gennemløb af løkken. I løkken der beregner 6 fakultet kan vi ønske at få udskrevet såvel `x` som `y`:

```
> y <- 1
> for (x in 2:6) { y <- y*x ; print(x) ; print(y) }
[1] 2
[1] 2
[1] 3
[1] 6
[1] 4
[1] 24
[1] 5
[1] 120
[1] 6
[1] 720
```

I løkkekroppen har vi nu tre udtryk (en tildeling og to `print`-kald) adskilt af semikoloner. Denne notation (som forklares nærmere i afsnit 17.2–17.3) betyder at de tre udtryk udføres efter hinanden i rækkefølge fra venstre til højre, dvs. først tildelingen til `y`, derefter udskriften af `x` og endelig udskriften af `y`. Hvis vi ser på de i alt 10 tal, der skrives ud, kan vi se at de svarer til hinanden parvis: de første to tal er

fra første gennemløb hvor både `x` og `y` er 2; de næste to tal er fra andet gennemløb hvor `x` er 3 og `y` er 6; osv.

Blandingen af `x` og `y` værdier kan måske være svært at overskue. Det er bedre at udskrive vektoren `c(x,y)` i stedet:

```
> y <- 1
> for (x in 2:6) { y <- y*x ; print(c(x,y)) }
[1] 2 2
[1] 3 6
[1]  4 24
[1]   5 120
[1]   6 720
```

Her kan man tydelig se udviklingen af `y` for hvert gennemløb.

15.2 Eksempel: Brownsk bevægelse

Vi vil som eksempel på brug af `for`-løkker simulere Brownsk bevægelse[12] i én dimension. Vi betragter en partikel hvis position (i én dimension – tænk på det som om den kun kan bevæge sig op og ned) til tiden t er et tal x_t. Mellem tid t og tid $t+1$ bevæger partiklen sig et lille stykke ϵ_t, hvor ϵ_t er et ligefordelt[13] tilfældigt tal mellem -1 og 1 uafhængigt af x_t og af senere og tidligere værdier af ϵ. Modellen er altså:

$$x_{t+1} = x_t + \epsilon_t,$$

med ϵ_t et tilfældigt tal mellem -1 og 1.

Vi vil nu starte med $x_0 = 0$ og simulere bevægelsen frem til $t = 100$, altså 100 trin:

```
> x <- 0
> for ( t in 1:100 ) { x <- x + runif(1, -1, 1) }
> x
[1] -3.419144
```

Funktionskaldet `runif(1, -1, 1)` giver et pseudotilfældigt tal mellem -1 og 1, se afsnit 11.

Efter 100 trin har partiklen altså flyttet sig cirka 3.42 enheder i lige netop denne simulation – hvis vi kører simulationen igen vil partiklen sandsynligvis flytte sig anderledes.

Men hvad hvis vi er interesseret i de mellemliggende positioner og ikke kun x_{100}? Så må vi gemme de mellemliggende x-værdier som følger.

Først opretter vi en vektor `X` med 100 pladser (hver med den vilkårlige startværdi 0) til at indeholde de 100 værdier x_1 til x_{100}. Derefter gentager vi eksperimentet, men gemmer nu i løkken hver værdi x_t i `X[t]`:

```
X <- rep(0, 100)
x <- 0
for ( t in 1:100 ) { x <- x + runif(1, -1, 1) ; X[t] <- x }
```

I løkkekroppen har vi nu to tildelinger, adskilt med semikolon, hvilket vil sige at først beregnes den nye værdi af `x` (første tildeling) og derefter gemmes den i `X[t]` (anden tildeling). I første gennemløb af

[12] Opkaldt efter den skotske botaniker *Robert Brown* (1773-1858), som i 1827 da han kiggede på pollen i vand under mikroskop observerede, at små partikler i en væske bevægede sig rundt tilfældigt i små ryk (hvilket skyldes at væskens molekyler støder ind i partiklerne). Brown påviste at dette var et fysisk frem for biologisk fænomen ved at gentage eksperimentet for mineralsk støv. Albert Einstein (1879-1955) præsenterede i 1905 Brownsk bevægelse som en indirekte måde at bevise eksistensen af atomer og molekyler og teorien blev eksperimentelt eftervist af Jean Baptiste Perrin (1870-1942). Brownsk bevægelse anvendes blandt andet også til matematisk at modellere aktiemarkedets udvikling, først af Louis Bachelier (1870-1946) i år 1900 (Bachelier var den første til matematisk at modellere Brownsk bevægelse).

[13] I den gængse matematiske model for Brownsk bevægelse, *Wiener processen*, er ϵ ikke ligefordelt men derimod normalfordelt, men vi ønsker ikke her at komme ind på normalfordelingen eller stokastiske processer.

løkkekroppen er `t=1` og x_1 gemmes derfor i `X[1]`. I andet gennemløb er `t=2` og x_2 gemmes derfor i `X[2]`, osv. Skrevet ud uden en `for`-løkke ville det se således ud:

```
X <- rep(0, 100)
x <- 0
x <- x + runif(1, -1, 1) ; X[1] <- x
x <- x + runif(1, -1, 1) ; X[2] <- x
 ... (i alt 100 linjer af denne type) ...
x <- x + runif(1, -1, 1) ; X[99] <- x
x <- x + runif(1, -1, 1) ; X[100] <- x
```

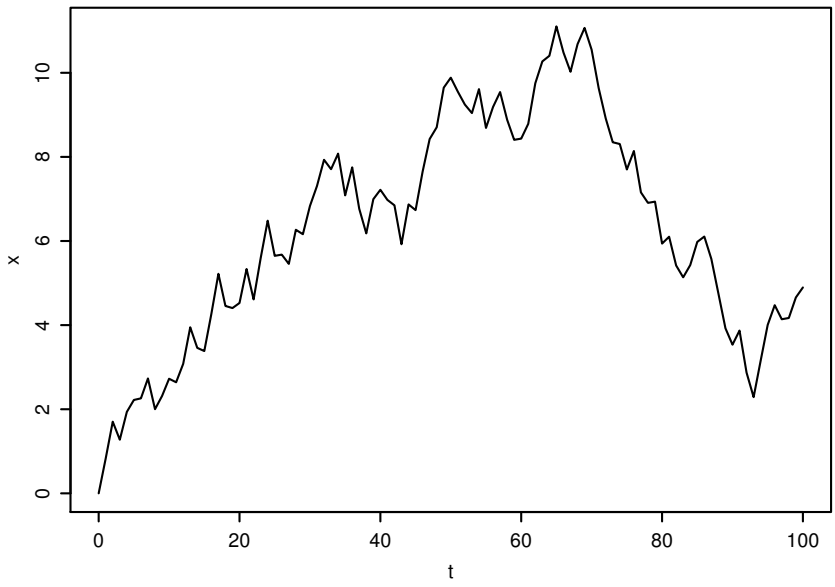
Vi har altså efter løkken værdierne x_1 til x_{100} gemt i X:

```
> X
  [1]  0.8296121  1.7037629  1.2760420  1.9369372  2.2204283  2.2586202
  [7]  2.7317968  2.0011300  2.3151146  2.7252441  2.6407277  3.0789522
   ...
 [91]  3.8700470  2.8705248  2.2876647  3.1537329  4.0050224  4.4732110
 [97]  4.1393550  4.1694817  4.6574310  4.8957494
```

Vi kan grafisk vise partiklens bevægelse som funktion af tiden:

```
plot(0:100, c(0,X), type="l", xlab="t", ylab="x")
```

Bemærk at vi for at få (0,0) med som første punkt i grafen lod t-koordinaterne være sekvensen `0:100` og klistrede et ekstra nul på i starten af `X` med `c(0,X)`. Den resulterende graf ses i figur 38. Hvis vi gentog eksperimentet ville vi sandsynligvis få en anden graf.



Figur 38: Graf fra simulation af Brownsk bevægelse i én dimension.

Opgave 32 udvider dette eksempel til at simulere en partikels bevægelse i to dimensioner mens opgave 47 omhandler et tilfælde hvor en partikels bevægelse er begrænset af to plader.

15.3 Eksempel: Fibonacci-tal

Fibonacci-tallene[14] er den uendelige talrække

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

hvor Fibonacci-tal nummer n er defineret som følger:

$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{hvis } n > 1 \end{cases}$$

Vi vil nu med en `for`-løkke udregne de første 30 Fibonacci-tal og gemme dem i en vektor `fib` således at `fib[1]` er Fibonacci-tal nummer 1, `fib[2]` er Fibonacci-tal nummer 2, og så videre:

```
> fib <- rep(1,30)
> for(i in 3:30) { fib[i] <- fib[i-1] + fib[i-2] }
> fib
 [1]      1      1      2      3      5      8     13     21     34
[10]     55     89    144    233    377    610    987   1597   2584
[19]   4181   6765  10946  17711  28657  46368  75025 121393 196418
[28] 317811 514229 832040
```

Virkemåden i detaljer: Først sættes `fib` til at være en vektor med 30 elementer, alle med værdien 1. Da Fibonacci-tal nummer 1 og 2 begge er 1 er `fib[1]` og `fib[2]` hermed allerede korrekte mens `fib[3]` til `fib[30]` skal beregnes i `for`-løkken. Løkkekroppen udføres for hvert tal `i` mellem 3 og 30, og beregner `fib[i]` som summen af de to foregående (og allerede beregnede) tal `fib[i-1]` og `fib[i-2]`. Til sidst vises indholdet af `fib`.

Dette eksempel udbygges i afsnit 19.2.3.

[14] Fibonacci: Leonardo af Pisa, kaldet Fibonacci (= søn af Bonaccio), omkr. 1180-1250, Italiensk matematiker man tilskriver den første renæssance for matematikken på kristen jord. Udgav i 1202 *Liber Abaci* hvori bl.a. Fibonacci-talsekvensen introduceres (i Vesteuropa - den var allerede kendt i indisk matematik). Fibonnaci-sekvensen har forbindelse til *det gyldne snit* i billedkunst og dukker op i naturen fx i fyrrekogler, ananas og solsikker.

16 Matricer

En matrix i R er ligesom i matematik et rektangulært skema af tal. I det følgende beskrives hvordan man i R opretter og regner med matricer.

16.1 Oprettelse af matricer

Man kan definere matricen

$$A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

sådan her:

```
> A <- matrix(c(1,2,3,4), 2)
```

Det sidste 2-tal betyder at matricen skal have to rækker. Det kan også skrives `nrow=2`, og tilsvarende kan man skrive `ncol=2` for at angive at der skal være to søjler. Som sædvanlig vises værdien af `A` ved at skrive navnet:

```
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Bemærk at `A` oprettes søjlevis: de to første elementer går til søjle 1, de to sidste til søjle 2. Hvis man i stedet vil bruge elementerne til rækkevis oprettelse af matricen kan man give argumentet `byrow=TRUE`, så de to første elementer går til række 1 og de to sidste til række 2:

```
> A2 <- matrix(c(1,2,3,4), 2, byrow=TRUE)
> A2
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

I udskrifterne betyder klammen `[,1]` søjle 1, og klammen `[1,]` betyder række 1.

Matricer kan have lige så mange rækker og søjler man ønsker, og behøver ikke være kvadratiske:

```
> M1 <- matrix(1:12, 3)
> M1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> M2 <- matrix(1:8, 4)
> M2
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

Man kan definere enhedsmatricen

$$E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

med 2 rækker og 2 søjler ved brug af funktionen `diag`:

```
> E <- diag(2)
> E
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Man kan nogle gange have brug for at *transponere* en matrix, dvs. bytte om på rækker og søjler. Dette gøres i R med funktionen `t`:

```
> t(M2)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

16.2 Indeksering i matricer

Hvis `A` er en matrix som ovenfor, så er `A[1,2]` elementet i første rækkes anden søjle, og `A[,1]` henholdsvis `A[,2]` er første henholdsvis anden søjle, og tilsvarende er `A[1,]` henholdsvis `A[2,]` første henholdsvis anden række:

```
> A[1,2]
[1] 3
> A[,1]
[1] 1 2
> A[1,]
[1] 1 3
```

Bemærk, at når vi udtager en hel række eller søjle er resultatet en vektor.

Analogt med indeksering i vektorer (afsnit 14.2) kan man anvende en vektor som indeks til at udvælge rækker og/eller søjler. Herved kan man frit udtage delmatricer og bytte om på søjler eller rækker. Med `M1` defineret som ovenfor kan man udtrække hele række 3 og hele række 1 og få en ny matrix:

```
> M1[c(3,1),]
     [,1] [,2] [,3] [,4]
[1,]    3    6    9   12
[2,]    1    4    7   10
```

Eller man kan udtrække hele søjle 2 og hele søjle 4 og få en ny matrix:

```
> M1[,c(2,4)]
     [,1] [,2]
[1,]    4   10
[2,]    5   11
[3,]    6   12
```

Tilsvarende kan man fx udtrække række 1, søjle 2 til 4; eller række 1 og 2, søjle 3; eller række 1 og 2, søjle 3 og 4. Hvor der kun udtages fra en enkelt søjle eller række bliver resultatet en vektor, ellers bliver resultatet en ny matrix:

```
> M1[1, 2:4]
[1]  4  7 10
> M1[1:2, 3]
[1] 7 8
> M1[1:2, 3:4]
     [,1] [,2]
[1,]    7   10
[2,]    8   11
```

Med negative indeksværdier kan man (analogt med indeksering i vektorer) angive søjler og/eller rækker, der *ikke* skal med:

```
> M1[,-2]
     [,1] [,2] [,3]
[1,]    1    7   10
[2,]    2    8   11
[3,]    3    9   12
> M1[-1,]
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
[2,]    3    6    9   12
> M1[2, -3]
[1]  2  5 11
```

16.3 Opbygning af matricer med `cbind` og `rbind`

Funktionen `cbind` kombinerer vektorer eller matricer søjlevis (c for "column") ved at "sætte dem ved siden af hinanden". Resultatet af `cbind(...)` er en matrix:

```
> A <- matrix(1:4, 2)
> v <- c(5,6)
> w <- c(7,8)
> cbind(v, w)
     v w
[1,] 5 7
[2,] 6 8
> cbind(A, w)
          w
[1,] 1 3 7
[2,] 2 4 8
```

(Vi bemærker at `cbind` automatisk indfører søjlenavne for de søjler, der kommer fra variable. Se afsnit 16.8 om dette.)

Funktionen `rbind` kombinerer vektorer eller matricer rækkevis (r for "row") ved at "stable dem oven på hinanden". Resultatet af `rbind(...)` er en matrix:

```
> rbind(v, w)
  [,1] [,2]
v    5    6
w    7    8
> rbind(A,w)
  [,1] [,2]
     1    3
     2    4
w    7    8
```

(Vi bemærker at `rbind` automatisk indfører rækkenavne for de rækker, der kommer fra variable. Se afsnit 16.8 om dette.)

Det er instruktivt at sammenligne med funktionen `c` fra afsnit 14.1, der sætter vektorer i forlængelse af hinanden. Resultatet af `c(...)` er en vektor:

```
> c(v, w)
[1] 5 6 7 8
```

Funktion i R	Matematik	Betydning
`A + B`	$A+B$	`A` plus `B`, elementvis addition
`A - B`	$A-B$	`A` minus `B`, elementvis subtraktion
`A * B`	**PAS PÅ**	`A` gange `B`, elementvis multiplikation
`A %*% B`	AB	`A` gange `B`, matrixprodukt
`A / B`	**PAS PÅ**	`A` divideret med `B`, elementvis division
`A ^ y`	**PAS PÅ**	`A` opløftet i `y`, elementvis
`t(A)`	A'	den transponerede til `A`
`det(A)`	$\det(A)$, $\lvert A\rvert$	determinanten af `A`
`solve(A)`	A^{-1}	den inverse til `A`
`solve(A, v)`		find `u` der løser ligningen `A %*% u = v`
`eigen(A)`		egenvektorer og egenværdier for `A`
`A[i,j]`	A_{ij}	indeksering: `i`'te række, `j`'te søjle af A
`A[i,]`	$A_{i\cdot}$	indeksering: `i`'te række af A
`A[,j]`	$A_{\cdot j}$	indeksering: `j`'te søjle af A

Figur 39: Operatorer og funktioner på matricer og vektorer i R. Inden du bruger en operator mærket "**PAS PÅ**", så overvej om ikke der snarere er brug for fx `%*%` eller `solve(A)` eller matrixpotensopløftning.

16.4 Regning med matricer

Alle sædvanlige regneoperatorer på matricer findes også i R, men nogle af dem har uventede navne. Især kan det overraske at `A * B` ikke er matrixprodukt; det skal i stedet skrives `A %*% B`. Læg også mærke til at `A^-1` ikke beregner den inverse til matrix `A`; den skal beregnes med `solve(A)`. Nedenfor følger nogle eksempler på matrixoperationer.

I det følgende eksempler benytter vi to matricer `A` og `B` defineret som følger:

```
> A <- matrix(1:4, 2)
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> B <- matrix(9:6, 2)
> B
     [,1] [,2]
[1,]    9    7
[2,]    8    6
```

Beregn resultatet af at addere matrix A og B:

```
> A + B
     [,1] [,2]
[1,]   10   10
[2,]   10   10
```

Beregn resultatet af at gange matrix A med matrix B, som matrixprodukt, dvs:

$$AB = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} 9 & 7 \\ 8 & 6 \end{pmatrix} = \begin{pmatrix} 33 & 25 \\ 50 & 38 \end{pmatrix}$$

```
> A %*% B
     [,1] [,2]
[1,]   33   25
[2,]   50   38
```

Det er *ikke* det samme som `A * B`, der ganger A med B *elementvis*. Dette giver som regel kun mening hvis en matrix multipliceres med et *tal*, hvor R så anvender genbrugsreglen fra afsnit 14.3 for at få de to operander til at have lige mange elementer:

```
> 7 * A
     [,1] [,2]
[1,]    7   21
[2,]   14   28
```

Når en matrix $M1$ der har r rækker og k søjler (en $r \times k$ matrix) multipliceres med en matrix $M2$ der har k rækker og s søjler (en $k \times s$ matrix), så har resultatmatricen $M1 \cdot M2$ r rækker og s søjler (en $r \times s$ matrix). Ved matrixmultiplikationen `M1 %*% M2` med `M1` og `M2` som defineret i afsnit 16.1 ovenfor er $r = 3$, $k = 4$ og $s = 2$ og resultatet derfor en matrix med 3 rækker og 2 søjler:

```
> M1 %*% M2
     [,1] [,2]
[1,]   70  158
[2,]   80  184
[3,]   90  210
```

Anvender man matrixmultiplikation `%*%` med vektorer betragtes en vektor til venstre som en rækkevektor og en vektor til højre som en søjlevektor:

```
> A %*% c(5,6)
     [,1]
[1,]   23
[2,]   34
> c(5,6) %*% A
     [,1] [,2]
[1,]   17   39
> c(1,2,3) %*% c(7,9,13)
     [,1]
[1,]   64
```

Sidste eksempel ovenfor beregner prikproduktet eller indre produkt af vektorerne `c(1,2,3)` og `c(7,9,13)`.

16.4.1 Potensopløftning af matricer

Hvis A er en matrix og n er et naturligt tal, så betyder A^n opløftning af A til n'te potens, hvilket er det samme som A matrixmultipliceret med sig selv n gange:

$$A^n = \underbrace{A \cdots A}_{n}$$

Der er ikke nogen indbygget funktion til dette formål i R; udtrykket `A^n` beregner noget helt andet som forklaret ovenfor. For mindre potenser, fx $n = 4$, kan man nemt beregne $R = A^n$ manuelt:

```
> R <- A
> R <- R %*% A
> R <- R %*% A
> R <- R %*% A
> R
     [,1] [,2]
[1,]  199  435
[2,]  290  634
```

Man kan alternativt bruge en `for`-løkke:

```
> R <- A
> for (n in 2:4) { R <- R %*% A }
> R
     [,1] [,2]
[1,]  199  435
[2,]  290  634
```

Dette er nemt også for større værdier af n. Bemærk, at løkken går fra 2 til n og ikke starter ved 1 – det er fordi `R` sættes til $A^1 = A$ allerede før løkken.

Endnu mere bekvemt er det dog selv én gang for alle at definere en funktion `matpow(A, n)` til at beregne A^n som vist i afsnit 19.5.5, og derefter bruge den som enhver anden funktion:

```
> matpow(A, 4)
     [,1] [,2]
[1,]  199  435
[2,]  290  634
```

16.5 Determinant og invers, lineære ligningssystemer

Determinanten af en matrix beregnes med funktionen `det`:

```
> det(A)
[1] -2
```

Den inverse til en matrix A, der i matematik ofte betegnes A^{-1}, beregnes i R med funktionen `solve`:

```
> solve(A)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Tjek af at A gange sin egen inverse (og omvendt) giver enhedsmatricen:

```
> A %*% solve(A)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> solve(A) %*% A
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Funktionen hedder `solve` fordi den også bruges til at løse lineære ligningssystemer (som man har udtrykt på matrixform). Lad os for eksempel løse ligningssystemet

$$\begin{aligned} x_1 &+ 3x_2 &= 3, \\ 2x_1 &+ 4x_2 &= 5 \end{aligned}$$

hvilket på matrixform er

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}.$$

Udregnet i R med `solve`:

```
> A <- matrix(c(1,2,3,4), 2)
> x <- solve(A, c(3,5))
> x
[1] 1.5 0.5
```

Det vil sige at $x_1 = 0.5$, $x_2 = 1.5$ er løsning til ligningssystemet. Vi kan tjekke at `x` faktisk er en løsning ved at gange `A` med `x`, resultatet skal være `c(3,5)`:

```
> A %*% x
     [,1]
[1,]    3
[2,]    5
```

16.6 Egenværdier og -vektorer

Egenværdier og egenvektorer kan beregnes med funktionen `eigen`. Lige som fx `lm` eller `integrate` giver `eigen` et resultat med flere komponenter (i en associationsliste, se afsnit 20.1). Der er to komponenter, nemlig `$values`, som er egenværdierne ordnet efter faldende numerisk værdi[15], og `$vectors`, som er en matrix med tilhørende (enheds)egenvektorer som søjler.

I dette eksempel udregnes egenværdier og -vektorer for A og løsningen gemmes i en variabel `evA`. Den første og anden egenvektor fås derefter ved at udtrække første hhv. anden søjle af `ev$vectors`:

```
> evA <- eigen(A)
> evA
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
> ev1 <- evA$vectors[,1]
> ev2 <- evA$vectors[,2]
```

Eftersom `ev1` er en egenvektor for `A` og `evA$values[1]` er den tilhørende egenværdi, skal nedenstående to udregninger give samme resultat:

```
> A %*% ev1
          [,1]
[1,] -3.039462
[2,] -4.429794
> evA$values[1] * ev1
[1] -3.039462 -4.429794
```

I nogle tilfælde vil en eller flere egenværdier for en matrix være komplekse tal[16], hvilket gør at `$values` og `$vectors` i resultatet fra `eigen` bliver komplekse:

[15] Den dominerende egenværdi er således altid den, der står først.

[16] Se fodnote 7 på side 30 om komplekse tal.

```
> M <-matrix(c(2.0,0.7,0,1.6,0,0.6,1.3,0,0.3),3)
> M
     [,1] [,2] [,3]
[1,]  2.0  1.6  1.3
[2,]  0.7  0.0  0.0
[3,]  0.0  0.6  0.3
> evM <- eigen(M)
> evM$values
[1]  2.5375365+0.0000000i -0.1187682+0.2620144i -0.1187682-0.2620144i
> evM$vectors
              [,1]                   [,2]                   [,3]
[1,] 0.96155200+0i  0.0336399+0.2504825i  0.0336399-0.2504825i
[2,] 0.26525191+0i  0.5213340-0.3261876i  0.5213340+0.3261876i
[3,] 0.07112785+0i -0.7469535+0.0000000i -0.7469535+0.0000000i
```

I ovenstående eksempel er den første egenværdi et reelt tal mens de to andre er komplekse tal. At egenværdierne er ordnet efter faldende numerisk værdi kan man verificere med `abs`:[17]

```
> abs(evM$values)
[1] 2.5375365 0.2876759 0.2876759
```

16.7 Antal rækker og søjler

Man kan bestemme antal rækker i en matrix med funktionen `nrow` og tilsvarende kan man bestemme antal søjler med `ncol`:

```
> M<-matrix(1:12,3)
> M
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> nrow(M)
[1] 3
> ncol(M)
[1] 4
```

Der er også en funktion `dim` som giver både antal rækker og søjler:

```
> dim(M)
[1] 3 4
```

16.8 Rækkenavne og søjlenavne

Ligesom elementerne i vektorer kan have navne (afsnit 14.5) kan rækker og søjler i matricer navngives. Funktionen `rownames` svarer til `names` men manipulerer rækkenavne i matricer:

[17] Den numeriske værdi (eller modulus) af et komplekst tal $a+bi$ defineres som $|a+bi| = \sqrt{a^2+b^2}$ og er således et reelt tal. Hvis man tænker de komplekse tal repræsenteret i et koordinatsystem, hvor den relle del afsættes ud af x-aksen og den imaginære del ud af y-aksen, svarer $a+bi$ til punktet (a,b) og $|a+bi|$ til afstanden fra $(0,0)$ til punktet. Reelle tal vil ligge på x-aksen i dette system og vores almindelige opfattelse af "numerisk værdi" af et reelt tal x harmonerer med at afstanden fra $(0,0)$ til $(x,0)$ er $|x|$.

```
> rownames(A)=c("r1","r2")
> A
   [,1] [,2]
r1    1    3
r2    2    4
> rownames(A)[2]="rk2"
> A
    [,1] [,2]
r1     1    3
rk2    2    4
> rownames(A)
[1] "r1"  "rk2"
```

Tilsvarende gælder med funktionen `colnames` og søjlenavne:

```
> colnames(A)=c("s1","s2")
> A
    s1 s2
r1   1  3
rk2  2  4
```

Man kan bruge navnene til indeksering:

```
> A["r1","s2"]
[1] 3
```

Endelig kan man fjerne navnene ved at sætte dem til `NULL`, der betyder "ingenting":

```
> colnames(A) <- NULL
> rownames(A) <- NULL
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Søjler eller rækker får ofte automatisk navne når man bruger `cbind` og `rbind` til at opbygge en matrix:

```
> a<-1:2
> b<-3:4
> cbind(a,b)
     a b
[1,] 1 3
[2,] 2 4
> rbind(a,b)
  [,1] [,2]
a    1    2
b    3    4
```

Her arves søjle- eller rækkenavnene altså fra variabelnavnene! Det er måske mere forventeligt at navnene videregives hvis man har navngivet vektorelementerne:

```
> names(a)<-c("x1","x2")
> cbind(a,b)
   a b
x1 1 3
x2 2 4
```

16.9 Eksempel: Fremskrivning med lineær afbildning (kaninpopulation)

Som eksempel vil vi lave fremskrivninger i modellen for en kaninpopulation givet i Anvendelseseksempel B.1 i *Noter om matematik*. Modellen angiver hvordan en population af kaniner opdelt i unge individer x og gamle individer y udvikler sig fra år til år. Fra år t til år $t+1$ er udviklingen som følger:

$$\begin{aligned} x_{t+1} &= 2.0 \cdot x_t + 1.5 \cdot y_t \\ y_{t+1} &= 0.7 \cdot x_t + 0.4 \cdot y_t \end{aligned}$$

Vi antager at i år 0 er der 100 unge og ingen gamle kaniner, dvs. at $x_0 = 100$ og $y_0 = 0$. Vi vil gerne vide hvordan populationen ser ud efter 10 år. Vi opskriver modellen på matrixform således:

$$\mathbf{v}_t = \begin{pmatrix} x_t \\ y_t \end{pmatrix}, \quad \mathbf{M} = \begin{pmatrix} 2.0 & 1.5 \\ 0.7 & 0.4 \end{pmatrix}, \quad \mathbf{v}_{t+1} = \mathbf{M}\mathbf{v}_t, \quad \mathbf{v}_0 = \begin{pmatrix} 100 \\ 0 \end{pmatrix}.$$

I R opskriver vi det som følger. Først oprettes matricen `M` (og vi kontrollerer at vi ikke har fået byttet om på rækker og søjler):

```
> M <- matrix(c(2.0, 0.7, 1.5, 0.4), 2)
> M
     [,1] [,2]
[1,]  2.0  1.5
[2,]  0.7  0.4
```

Herefter oprettes `v0` (svarende til $\mathbf{v}_0$) samt en variabel `v` som vi vil fremskrive i hvert trin og som indledningsvis er lig `v0`.

```
v0 <- c(100,0)
v <- v0
```

Nu er vi klar til at fremskrive og vi benytter en simpel `for`-løkke (afsnit 15) til dette:

```
for(t in 1:10) { v <- M %*% v }
```

Efter løkken er resultatet ($\mathbf{v}_{10}$) i `v`:

```
> v
          [,1]
[1,] 770275.4
[2,] 256758.5
```

Efter 10 år er der `sum(v) = 1027034` kaniner, altså over en million ...

Men hvad hvis vi også ønskede kaninpopulationen i de mellemliggende år og ikke kun $\mathbf{v}_{10}$? Så må vi gemme resultatet i en matrix hvor første søjle er populationen til $t = 0$ (altså $\mathbf{v}_0$), anden søjle populationen efter et år ($\mathbf{v}_1$) osv. indtil sidste søjle som er populationen efter ti år ($\mathbf{v}_{10}$). Resultatet er dermed en 2×11 matrix som vi vil kalde $\mathbf{V}$:

$$\mathbf{V} = (\mathbf{v}_0 \mathbf{v}_1 \cdots \mathbf{v}_{20}) = \begin{pmatrix} x_0 & x_1 & \cdots & x_{10} \\ y_0 & y_1 & \cdots & y_{10} \end{pmatrix}.$$

I R gør vi nu indledningsvis fuldstændig som før og opretter matricen `M`, vektoren `v0` og variablen `v`. Endvidere oprettes matricen $\mathbf{V}$ med kun én søjle; resten af `V` vil blive klistret på, søjle for søjle, efterhånden som vi fremskriver.

```
M <- matrix(c(2.0, 0.7, 1.5, 0.4), 2)
v0 <- c(100,0)
v <- v0
V <- matrix(v0,2)
```

Nu bruger vi en lidt udvidet udgave af `for`-løkken fra før til selve fremskrivningerne:

```
for(t in 1:10) { v <- M %*% v ; V <- cbind(V,v) }
```

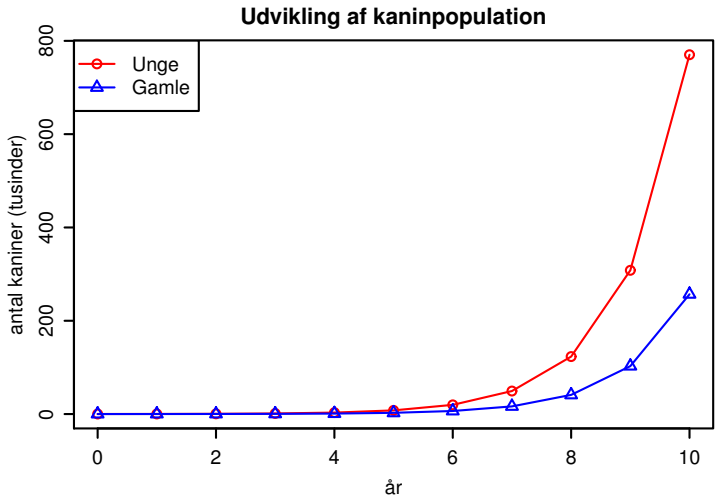
Løkken gennemløbes 10 gange og hver gang beregnes først den nye population med `v <- M %*% v` og derefter klistres den på `V` som sidste søjle med `V <- cbind(V,v)`. De to tildelinger i løkkekroppen udføres i rækkefølge når de er adskilt med semikolon, se afsnit 17.2–17.3. Når løkken er færdig er `v` kaninpopulationen efter 10 år og `V` har fået tilføjet 10 søjler svarende til populationerne i hver af de 10 år. Vi kan nu vise resultatet:

```
> v
         [,1]
[1,] 770275.4
[2,] 256758.5
> V
     [,1] [,2] [,3]   [,4]    [,5]     [,6]      [,7]     [,8]
[1,]  100  200  505 1262.0 3155.05 7887.620 19719.051 49297.63
[2,]    0   70  168  420.7 1051.68 2629.207  6573.017 16432.54
          [,9]    [,10]    [,11]
[1,] 123244.07 308110.2 770275.4
[2,]  41081.36 102703.4 256758.5
```

Vi kan nu nemt udtage alle x'erne henholdsvis y'erne som to vektorer, for eksempel for at tegne en graf der viser udviklingen. Den første række af `V` er jo $x_0,\ x_1,\ \ldots,\ x_{10}$ og kan udtages med udtrykket `V[1,]`, og tilsvarende kan anden række $(y_0,\ y_1,\ \ldots,\ y_{10})$ udtages med `V[2,]`. Dermed kan vi få tegnet en graf som følger (antal kaniner deles med 1000 for at få pænere enheder på akserne):

```
plot(0:10, V[1,]/1000, type="o", ylim=c(0, max(V)/1000),
     col="red", xlab="år", ylab="antal kaniner (tusinder)",
     main="Udvikling af kaninpopulation")
points(0:10, V[2,]/1000, type="o", pch=2, col="blue")
legend("topleft", c("Unge","Gamle"), pch=c(1,2), lty="solid",
       col=c("red", "blue"))
```

Dette giver grafen i figur 40. Eksemplet med kaninpopulationen bruges også i afsnit 18 og afsnit 19.5.2.



Figur 40: Graf fra beregning af en kaninpopulations udvikling.

17 Noget om udtryk i R

I de følgende afsnit vil blive gennemgået hvordan man kan anvende R på en mere avanceret måde, herunder hvordan man kan gøre det lettere for sig selv når man skal løse større opgaver. Men først er det nødvendigt at tage et nærmere kig på den helt basale struktur af R sproget. Forhåbentlig vil det give en dybere forståelse af hvordan de forskellige elementer i R spiller sammen og dermed også gøre det nemmere at finde årsagen når noget, man prøver, måske ikke helt virker som man havde tænkt sig.

17.1 Udtryk, værdier og sideeffekter

De grundlæggende byggesten i R sproget er *udtryk*. Alt man kan taste ind er udtryk og ethvert udtryk giver en *værdi* når det beregnes. Vi har set talrige eksempler på udtryk i det foregående, især regneudtryk og funktionskald, men også funktionsdefinitioner og binding af værdier til variable og sågar `for`-løkker er faktisk udtryk. Altså har hver eneste linje i eksemplerne på R sproget indtil nu været udtryk.

Ud over at et udtryk ved beregning giver en værdi kan beregningen (og dermed udtrykket) have en *sideeffekt*. En sideeffekt er noget der ændrer tilstanden af "verden" uden for udtrykket, for eksempel ved at binde en værdi til en variabel (hvilket har betydning for beregning af senere udtryk hvor variablen indgår) eller ved at tegne en graf (hvilket ændrer indholdet af grafvinduet eller skriver grafen i en fil).

17.1.1 Tildelingsudtryk og "usynlige" værdier

En binding af en værdi til en variabel, såsom `x <- 2`, kaldes en *tildeling* og er et udtryk. Værdien af et tildelingsudtryk er variablens nye værdi. Således kunne man bruge værdien direkte i en videre beregning, altså ved at have tildelingen som et deludtryk i et større udtryk. Dette kan dog absolut ikke anbefales, da det gør udtrykket meget svært at forstå. I en enkelt type udtryk kan det dog være i orden at bruge tildeling som et deludtryk, nemlig ved tildeling af samme værdi til flere variable:

```
> y <- x <- 2
```

Dette binder værdien 2 til både `x` og `y`.

Værdien af et tildelingsudtryk vises ikke i R Console. For eksempel giver nedenstående ikke noget direkte "svar" når det tastes ind i R Console:

```
> x <- 2
```

At en tildeling vitterlig er et udtryk som alle andre og har en værdi kan man overbevise sig om ved at sætte en parentes rundt om den:

```
> (x <- 2)
[1] 2
```

Nu bliver værdien vist i R Console. Et tildelingsudtryk er et eksempel på et udtryk, hvis værdi er gjort "usynlig" i R Console (fordi man som hovedregel ikke er interesseret i at se den). Et andet eksempel på et udtryk med en usynlig værdi er en `for`-løkke, hvis værdi er værdien af den sidste evaluering af løkkens krop. Mange funktioner i R skjuler deres værdi på denne måde, nemlig de funktioner der ikke er funktioner i matematisk forstand men snarere er *procedurer* der "gør noget" og altså kaldes for deres sideeffekters skyld. Et eksempel er funktionen `plot`, der tegner en graf. Hvis vi gerne vil se værdien af et kald af `plot` kan vi lige som med en tildeling pakke udtrykket ind i en parentes:

```
> (plot(sin,0,6*pi))
NULL
```

Dette giver værdien `NULL` (som er en særlig værdi der betyder "ingenting") og tegner samtidig en graf. Det er ikke kun funktioner som giver `NULL` der skjuler deres værdi. Funktionen `persp` (afsnit 21.1) giver

for eksempel en 4×4 matrix der kan bruges til at projicere 3D punkter ind i det tegnede koordinatsystem (se hjælpeteksten for `?persp` hvis dette har interesse).

17.2 Sekvenser af udtryk

Hidtil har (næsten) alle indtastninger i disse noter været af en form hvor én linje svarer til ét udtryk. Vi har således afsluttet alle udtryk med linjeskift. Man kan imidlertid afslutte et udtryk med semikolon (`;`) i stedet og dermed have flere udtryk på samme linje:

```
> 1+2; 2^5; sqrt(2)
[1] 3
[1] 32
[1] 1.414214
```

Dette svarer fuldstændig til at indtaste de tre regneudtryk på hver sin linje; R viser de tre resultater. Man kan altid afslutte et udtryk med et semikolon, uanset om det også står sidst på linjen, så nedenstående er for eksempel også en lovlig indtastning:

```
> 1+2;
[1] 3
```

17.2.1 Åbne og lukkede udtryk

Semikolon og linjeskift er en lille smule forskellige i den måde, de afslutter et udtryk på. Et semikolon afslutter *altid* et udtryk, og udtrykket før et semikolon skal derfor altid være "lukket", dvs. det skal være et komplet udtryk. Hvis et udtryk foran et semikolon er "åbent" (uafsluttet) er det en fejl (for eksempel hvis man har glemt en slutparentes). Et linjeskift afslutter derimod kun et udtryk hvis udtrykket *er* lukket. Hvis udtrykket er åbent ved slutningen af linjen fortsætter udtrykket simpelthen på næste linje. Ved indtastning i R Console skifter prompten fra > til + hvis linjen fortsætter et åbent udtryk fra den foregående linje. Et eksempel, hvor udtrykket `(1+2-3)` deles over to linjer:

```
> (1+2
+ -3)
[1] 0
```

Et andet eksempel på deling af samme udtryk:

```
> 1+2-
+ 3
[1] 0
```

I begge tilfælde er udtrykket åbent efter første linje. I første eksempel er det åbent fordi parentesen ikke er afsluttet, i andet eksempel fordi der er nødt til at følge en operand efter minustegnet.

17.3 Blokudtryk

Man kan samle en sekvens af udtryk i en *blok* ved at omslutte den med krølleparenteser (`{`) og (`}`):

```
> { 1+2; 2^5; sqrt(2) }
[1] 1.414214
```

Som det ses er værdien af sådan et *blokudtryk* simpelthen værdien af sidste udtryk. I ovenstående beregnes alle de tre udtryk, men værdierne af de to første udtryk smides væk, hvorfor det lige i dette eksempel er meningsløst overhovedet at have dem med. Det giver imidlertid mening at have flere udtryk i en blok hvis de (eventuelt på nær det sidste) har sideeffekter som for eksempel at binde værdier til variable eller tegne grafer:

```
> { x<-2; y<-3; x+y }
[1] 5
> x
[1] 2
```

Dette tildeler værdier til `x` og `y` og giver endelig en værdi hvor de to variable indgår i beregningen. Bemærk at de to variable har deres nye værdier (2 og 3) også bagefter, uden for blokken.

At samle en sekvens af udtryk til et enkelt blokudtryk på denne måde benyttede vi os af allerede i `for`-løkke eksemplerne i afsnit 15.2 og 16.9. Vi har i det hele taget konsekvent angivet `for`-løkkers kroppe som blokudtryk, også selv om blokken kun indeholdt et enkelt udtryk og de krøllede parenteser derfor kunne have været udeladt. Det samme har vi gjort med funktionskroppe. Der er to grunde til dette: For det første giver konsekvent brug af blokudtryk rent typografisk en god markering af, hvad der er "kroppen" af løkken eller funktionen. For det andet fanger den konsekvente brug af blokudtryk nogle typer fejl som fx glemte slutparenteser.

En blok kan lige som andre udtryk fortsættes over flere linjer. For eksempel:

```
> { x<-2
+   y<-3
+   x+y }
[1] 5
```

Bemærk at de første to linjeskift afslutter de to tildelingsudtryk men ikke selve blokken, der jo skal afsluttes af med højre krølleparentes (`}`). Dermed kan semikolonerne mellem udtrykkene udelades. Eller sagt på en anden måde: et lukket (del)udtryk afsluttes af et linjeskift, også inden i en blok[18].

17.4 Funktionsdefinitioner er også udtryk

En funktionsdefinition `function(...) {...}` er et udtryk hvis værdi er en funktion. Vi har hidtil kun set eksempler på definition af *navngivne funktioner*, hvor funktionen bindes til en variabel og derefter bruges ved navn. Nogle gange hvor vi kun bruger en funktion en enkelt gang, for eksempel for at tegne en kurve i en graf, kunne vi ligeså godt have anvendt funktionen uden at navngive den (som *anonym funktion*):

```
plot(function(x) { x^3 - 5*x^2 }, -2, 6)
```

Ovenstående tegner en graf for $y = x^3 - 5x^2$ for $x \in [-2, 8]$.

Hidtil har vi for læselighedens skyld konsekvent angivet kroppen af en funktion som et blokudtryk, dvs. i en krølle-parentes. En funktionskrop kan imidlertid være et vilkårligt udtryk, og således er alle nedenstående definitioner ækvivalente:

```
f <- function(x) x^3 - 5*x^2
f <- function(x) ( x^3 - 5*x^2 )
f <- function(x) { x^3 - 5*x^2 }
```

17.5 Kommentarer

Man kan angive kommentarer i sin R kode for at gøre den lettere at læse og forstå. Alt hvad der følger efter et #-tegn på en linje er en kommentar og ignoreres af R. For eksempel:

```
> 1:40   # giver mig tallene fra 1 til 40
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
[23] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
> # Denne linje er kun en kommentar.
```

[18] Det gælder dog ikke `if`-udtryk hvis de efterfølges af `else`, se afsnit 19.2.

18 Fra scripts til funktioner

Når den samme type beregning skal udføres mange gange, eller man for eksempel skal lave mange beslægtede grafer, kan man gøre det ved i R Editor at kopiere det "script" (de linjer R-kode) som udfører én udgave af beregningen og så lave de nødvendige ændringer for at det skal udføre en anden (og tredje, fjerde, ...) udgave af beregningen. Dermed får man et stort script sammensat af mange små næsten identiske scripts. Man kan imidlertid gøre det lettere og mere overskueligt for sig selv ved i stedet at definere en funktion med det grundlæggende script der udfører én beregning og så kalde den funktion hver gang, man har brug for beregningen. Dermed bliver hvad der ellers ville være et stort uoverskueligt script reduceret til én funktionsdefinition og et antal linjer med kald af funktionen.

Ofte vil man arbejde på den måde at man i første omgang løser én udgave af problemet med helt konkrete tal og derefter med udgangspunkt i det R-script, man har skrevet, skriver en funktion der løser problemet "smartere". De konkrete tal, man brugte i første omgang, ændrer man til *parametre* således at man kan variere dem når man kalder funktionen. At indkapsle beregningen i en funktion gør således at man kan *parametrisere* sin kode så den kan bruges til at løse mange forskellige udgaver af problemet.

Som eksempel på en beregning man med fordel kan indkapsle i en funktion vil vi bruge fremskrivning med lineær afbildning som blev udviklet for en model for kaninpopulation i afsnit 16.9. Her endte vi med at skulle igennem følgende trin for at definere modellen, fremskrive 20 år og opsamle resultaterne i en matrix `V` hvor hver søjle indeholder den fremskrevne population for et givet år:

```
M <- matrix(c(2.0, 0.7, 1.5, 0.4), 2)
v0 <- c(100,0)
v <- v0
V <- matrix(v0,2)
for(t in 1:20) { v <- M %*% v ; V <- cbind(V,v) }
```

Hvis vi nu vil ændre matricen `M`, startpopulationen `v0` eller antallet af år, er vi nødt til at gentage linjerne (med nogle få ændringer). Hvis vi arbejder fornuftigt og har linjerne i et R Editor vindue, hvor vi laver ændringerne, og så kopierer de rettede linjer derfra til R Console, er det måske overkommeligt, men det bliver alligevel let uoverskueligt når vores "script" i R Editor vokser og vokser.

En bedre løsning er at indkapsle selve fremskrivningen i en funktion. De tre ting, der varierer, (`M`, `v0` og antal år) gør vi til *parametre* i funktionen. I R Editor skriver vi:

```
fremskriv <- function(M, v0, antal.trin) {
   v <- v0
   V <- matrix(v0,2)
   for(t in 1:antal.trin) { v <- M %*% v ; V <- cbind(V,v) }
   V
}
```

Alt dette kopieres så over i R Console på én gang og dermed er funktionen `fremskriv` defineret.

Funktionskroppen ovenfor afviger på tre punkter fra de oprindelige indtastninger:

1. De to første linjer fra de oprindelige indtastninger, nemlig definitionerne af `M` og `v0`, er ikke taget med. `M` og `v0` er jo netop gjort til parametre i funktionen så vi kan udføre fremskrivning med forskellige matricer og startpopulationer, så det er når vi *anvender* funktionen at vi skal angive en konkret matrix og en konkret startpopulation.

2. Tilsvarende er sekvensen `1:20` i `for`-løkken blevet ændret til `1:antal.trin`, hvor `antal.trin` er en parameter i funktionen. Dermed kan vi fremskrive et forskelligt antal trin alt efter hvilken værdi vi angiver når vi bruger funktionen.

3. Endelig er der kommet en ny linje til, nemlig sidste linje der bare er udtrykket `V`. Hvad skal det nu til for? Jo, når vi kalder funktionen er det jo for at få beregnet en resultatmatrix `V` og denne matrix

er den værdi, vi forventer at "få ud af" funktionen, dvs. funktionens *returværdi*.
En funktion returnerer den værdi som fås ved at evaluere funktionskroppen med de givne parameterværdier. Funktionskroppen i `fremskriv` et et blokudtryk, og som beskrevet i afsnit 17.3 er værdien af et blokudtryk værdien af det sidste udtryk i blokken. Så for at få en bestemt værdi ud af funktionen skal et udtryk, der beregner værdien, altså stå som det allersidste udtryk (her: sidste linje) i blokken. Her er det den opbyggede matrix `V` vi vil have ud, så det sidste udtryk skal simpelthen være `V`.

Den oprindelige fremskrivning kan vi så lave som følger:

```
M <- matrix(c(2.0, 0.7, 1.5, 0.4), 2)
v0 <- c(100,0)
V1 <- fremskriv(M, v0, 20)
```

De to første linjer er de samme som i de oprindelige indtastninger, mens selve fremskrivningen nu forgår ved at kalde `fremskriv`.

Bemærk, at vi i sidste indtastningslinje ovenfor gemmer værdien fra kaldet af `fremskriv` i en variabel `V1` da vi ellers bare ville få resultatet vist i R Console uden at kunne bruge det i videre beregninger. Den matrix `V`, der opbygges inden i funktionen `fremskriv`, kendes *ikke* uden for funktionen. Det var derfor, vi var nødt til at gøre `V` til returværdi for funktionen for "at få resultatet ud". Variable, der tildeles en værdi i en funktionskrop, er *lokale variable* i funktionen. Lokale variable kendes *ikke* uden for den funktion, de er defineret i, og tildeling til dem ændrer *ikke* ved eventuelle ikke-lokale variable, der måtte hedde det samme. Så hvis man ønsker at få et resultat ud af en funktion *skal* det ske som funktionens returværdi.

At først definere en funktion til fremskrivning og derefter kalde den for at lave en enkelt fremskrivning er selvfølgelig umiddelbart ikke lettere end før, men fordelen kommer hvis vi fx skal lave fremskrivningen over 10 år for tre forskellige startpopulationer:

```
M <- matrix(c(2.0, 0.7, 1.5, 0.4), 2)
V1 <- fremskriv(M, c(100,0), 10)
V2 <- fremskriv(M, c(50,50), 10)
V3 <- fremskriv(M, c(0,100), 10)
```

Vi har nu de tre forskellige resultater som `V1`, `V2` og `V3`. Hvis vi nu vil lave tre grafer, én for hver populations udvikling, kan vi igen med fordel definere en funktion der laver denne type graf og så blot kalde funktionen for hver population. Funktionen `plot.udvikling` nedenfor gør netop dette:

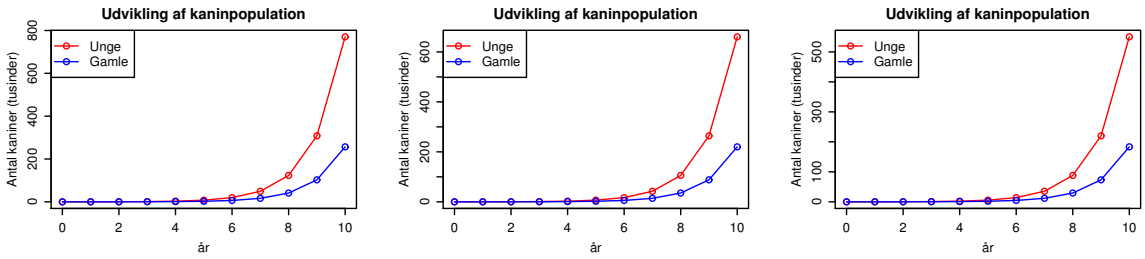
```
plot.udvikling <- function(V) {
   årnumre <- 0:(ncol(V)-1)
   plot(årnumre, V[1,]/1000, type="o", col="red",
        main="Udvikling af kaninpopulation", xlab="år",
        ylab="Antal kaniner (tusinder)")
   points(årnumre, V[2,]/1000, type="o", col="blue")
   legend("topleft", c("Unge", "Gamle"),
          lty="solid", pch=1, col=c("red", "blue"))
}
```

Her er populationsmatricen `V` eneste parameter. I modsætning til `fremskriv` har funktionen ikke nogen decideret returværdi – der skal bare tegnes en graf. Brug af denne funktion:

```
plot.udvikling(V1)
plot.udvikling(V2)
plot.udvikling(V3)
```

Dette producerer de tre grafer vist i figur 41.

Det kan nu være vi vil lave en graf der *sammenligner* udviklingen i de tre populationer `V1`, `V2` og `V3`



Figur 41: Tre grafer produceret med `plot.udvikling` funktionen. Eneste forskel for de tre startpopulationer (efter de første få år) viser sig at være en skalering af antallet af kaniner, grafisk kun synligt på de forskellige enheder på andenakserne.

i forhold til hinanden ved at vise den procentvise andel af unge kaniner i populationerne i samme graf. En vektor med den procentvise andel af unge kaniner i en population `V` over alle år kan beregnes som `100*V[1,]/(V[1,]+V[2,])`. For at undgå fejl når vi skal lave denne beregning for såvel `V1` som `V2` og `V3`, definerer vi en funktion til formålet:

```
pct.unge <- function(V) { 100*V[1,]/(V[1,]+V[2,]) }
```

Herefter kan vi lave grafen:

```
plot(0:10, pct.unge(V1), type="o", col="red", ylim=c(0,100),
     main="Udvikling af kaninpopulationer", xlab="år",
     ylab="Andel af unge i procent")
points(0:10, pct.unge(V2), type="o", col="blue")
points(0:10, pct.unge(V3), type="o", col="green")
```

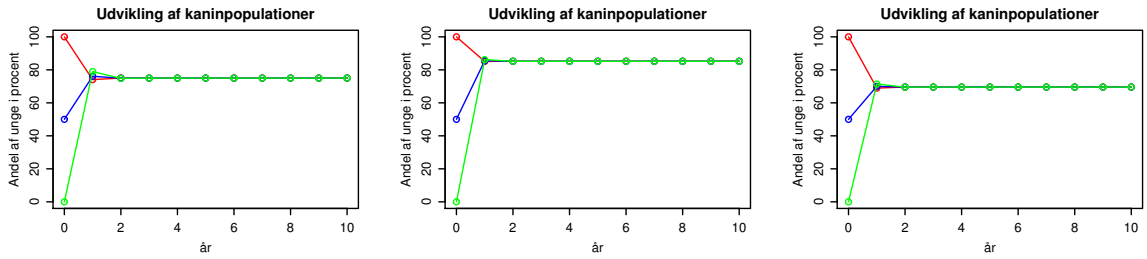
Hvis vi skulle lave mange af den type grafer, for eksempel for forskellige matricer `M`, kunne vi med fordel lave en funktion der klarer både fremskrivninger og plot:

```
kaningraf <- function(M, antal.trin) {
   V1 <- fremskriv(M, c(100,0), antal.trin)
   V2 <- fremskriv(M, c(50,50), antal.trin)
   V3 <- fremskriv(M, c(0,100), antal.trin)
   plot(0:antal.trin, pct.unge(V1), type="o", col="red", ylim=c(0,100),
        main="Udvikling af kaninpopulationer", xlab="år",
        ylab="Andel af unge i procent")
   points(0:antal.trin, pct.unge(V2), type="o", col="blue")
   points(0:antal.trin, pct.unge(V3), type="o", col="green")
}
```

Her er matricen `M` og antal trin (år) parametre. Ligesom `plot.udvikling` har funktionen ikke nogen decideret returværdi – der skal bare tegnes en graf. Brug af denne funktion:

```
M1 <- matrix(c(2.0, 0.7, 1.5, 0.4), 2)
kaningraf(M1, 10)
M2 <- matrix(c(4.0, 0.7, 2.5, 0.4), 2)
kaningraf(M2, 10)
M3 <- matrix(c(2.0, 0.9, 1.5, 0.6), 2)
kaningraf(M3, 10)
```

Dette producerer de tre grafer vist i figur 42.



Figur 42: Tre grafer produceret med `kaningraf` funktionen. Et konstant forhold mellem antal unge og gamle uanset startpopulation nås meget hurtigt for alle de tre modeller.

19 Logiske udtryk, `if` og `while`

I dette afsnit vil blive gennemgået hvordan man med *logiske udtryk* kan udtrykke betingelser og dermed får R til at "reagere" forskelligt afhængigt af om en betingelse er opfyldt eller ej. Man kan også med logiske udtryk nemt fra et datasæt udtage netop de observationer, som opfylder bestemte betingelser.

19.1 Logiske udtryk

Logiske udtryk er udtryk, der som resultat har en sandhedsværdi (altså sandt eller falsk; i R `TRUE` eller `FALSE`). Eksempler:

```
> x <- 3
> x > 5
[1] FALSE
> x < 5
[1] TRUE
```

"Mindre end" (<) og "større end" (>) er eksempler på *sammenligningsoperatorer*; de sammenligner to værdier og giver `TRUE` eller `FALSE`. Figur 43 giver en oversigt over sammenligningsoperatorerne i R.

Udtryk i R	Matematik	Betydning
$udtryk_1$ == $udtryk_2$	$udtryk_1 = udtryk_2$	$udtryk_1$ lig med $udtryk_2$
$udtryk_1$!= $udtryk_2$	$udtryk_1 \neq udtryk_2$	$udtryk_1$ forskellig fra $udtryk_2$
$udtryk_1$ < $udtryk_2$	$udtryk_1 < udtryk_2$	$udtryk_1$ mindre end $udtryk_2$
$udtryk_1$ > $udtryk_2$	$udtryk_1 > udtryk_2$	$udtryk_1$ større end $udtryk_2$
$udtryk_1$ <= $udtryk_2$	$udtryk_1 \leq udtryk_2$	$udtryk_1$ mindre end eller lig med $udtryk_2$
$udtryk_1$ >= $udtryk_2$	$udtryk_1 \geq udtryk_2$	$udtryk_1$ større end eller lig med $udtryk_2$

Figur 43: Sammenligningsoperatorer i R.

Faktisk virker sammenligningsoperatorer, lige som de aritmetiske operatorer, på vektorer af værdier, med den sædvanlige genbrugsregel hvis vektoren på den ene side er kortere end på den anden:

```
> x <- 1:10
> x > 5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x == 3
 [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

En vektor v_1 af sandhedsværdier kan bruges til indeksering af en anden vektor v_2. Det giver en ny vektor v_3 indeholdende de elementer fra v_2, hvor de tilsvarende pladser i v_1 har værdien `TRUE`. Dette kan vi

Operator i R	Matematik	Betydning
! *udtryk*	$\neg$*udtryk*	negation; "**ikke** *udtryk*"
udtryk$_1$ `&` *udtryk*$_2$	*udtryk*$_1 \wedge$ *udtryk*$_2$	*udtryk*$_1$ **og** *udtryk*$_2$
udtryk$_1$ `|` *udtryk*$_2$	*udtryk*$_1 \vee$ *udtryk*$_2$	*udtryk*$_1$ **eller** *udtryk*$_2$
udtryk$_1$ `&&` *udtryk*$_2$		*udtryk*$_1$ **og** *udtryk*$_2$, kun første element (se tekst)
udtryk$_1$ `||` *udtryk*$_2$		*udtryk*$_1$ **eller** *udtryk*$_2$, kun første element (se tekst)

Figur 44: Logiske operatorer. Den lodrette streg, der indgår i `|` og `||` operatorerne, får man på et PC-tastatur ved at taste AltGr+´, hvor ´ er anden tast til højre for 0 (nul). På en Mac får man den lodrette streg med ⌥+Ī.

bruge sammen med logiske udtryk for at udtage netop de elementer, hvorom der gælder noget bestemt:

```
> x[x>6]
[1]  7  8  9 10
```

(hvor `x` fortsat indeholder tallene fra 1 til 10).

Hvis vi ikke er interesserede i hvilke specifikke elementer fra `x`, der er større end 6, men bare vil vide *hvor mange* af tallene i `x`, der er større end 6, kan vi bruge `sum`:

```
> sum(x>6)
[1] 4
```

Dette virker fordi `sum` "regner" med sandhedsværdier og tæller hvert `TRUE` som tallet 1 og hvert `FALSE` som nul. Så ovenstående fortæller at i vektoren af sandhedsværdier, som er resultatet af `x>6`, er 4 af elementerne `TRUE` (og resten `FALSE`).

Logiske udtryk kan kombineres med *logiske operatorer*; se figur 44. For eksempel (stadig med samme `x`):

```
> x > 2 & x < 9
 [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
> !(x > 2 & x < 9)
 [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
> x <= 3 | x >= 7
 [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> x[!(x>2 & x<9) | x==5]
[1]  1  2  5  9 10
```

De to operatorer `&&` og `||` adskiller sig kun fra operatorerne `&` og `|` hvis operanderne er *vektorer* af logiske værdier. Operatorerne `&&` og `||` bruger altid kun det første element i hver operand og giver kun en enkelt sandhedsværdi som svar. For eksempel, med `x` fortsat defineret som tallene fra 1 til 10:

```
> x > 2 && x < 9
[1] FALSE
> x <= 3 || x >= 7
[1] TRUE
```

Svarene gælder det første element i `x`, dvs. tallet 1.

Derudover er der den forskel at `&&` og `||` ikke evaluerer deres anden operand hvis det samlede udtryks sandhedsværdi kan afgøres alene ud fra den første operand; fx hvis deludtrykket *udtryk*$_1$ er sandt i udtrykket *udtryk*$_1$ `||` *udtryk*$_2$ så evalueres deludtrykket *udtryk*$_2$ slet ikke. Dette kan have betydning hvis evalueringen af anden operand enten har en sideeffekt eller vil give en fejl. Dobbeltoperatorerne `&&` og `||` anvendes oftest sammen med `if ... else` udtryk som omtales i afsnit 19.2.

19.1.1 Eksempel: funktionen `heltal`

Man kan naturligvis definere en funktion, hvis krop er et logiske udtryk og som dermed giver sandhedsværdier som svar, når man kalder den. Som eksempel på dette vil vi her definere en funktion `heltal`, som givet et tal `x` vil returnere `TRUE`, hvis `x` er et heltal, og ellers vil returnere `FALSE` (vi får brug for funktionen i afsnit 19.2.2). Vi skriver `heltal` på følgende måde:

```
heltal <- function(x) { floor(x) == x }
```

Idéen er, at `floor(x)` giver `x` rundet ned til nærmeste heltal og netop hvis dette er lig `x` selv, er `x` et heltal, og sammenligningen giver da `TRUE`.

Afprøvning af den nye funktion:

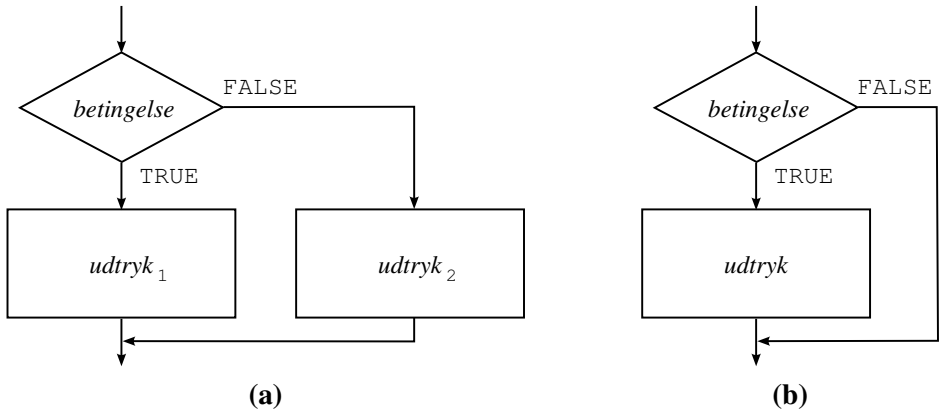
```
> heltal(42)
[1] TRUE
> heltal(4.2)
[1] FALSE
```

19.2 Betingede udtryk: `if ... else`

Et `if ... else` udtryk har følgende form:

`if (` *betingelse* `)` *udtryk*$_1$ `else` *udtryk*$_2$

Effekt: Udtrykket *betingelse* evalueres.[19] Hvis det giver `TRUE` evalueres *udtryk*$_1$ (kaldet `if`-grenen), ellers evalueres *udtryk*$_2$ (kaldet `else`-grenen). Værdien af det samlede `if ... else` udtryk er værdien af den af de to grene, der evalueres. Figur 45 **(a)** er et flowdiagram, der viser princippet.



Figur 45: Principper for **(a)** `if ... else` udtryk og **(b)** `if` udtryk. Beregningen af udtrykkene følger pilene, startende fra oven, og flowdiagrammerne læses på samme måde ved at følge pilene rundt.

Eksempler:

```
> x <- 5
> if (x<2) 2 else x
[1] 5
> if (x>2) 2 else x
[1] 2
```

Man kan udelade `else`-grenen, så har man et `if` udtryk (figur 45 **(b)**):

[19] Hvis betingelsen i et `if` eller `if ... else` udtryk giver en vektor af sandhedsværdier bruges kun det første element i vektoren. Hvis der er brug for en konstruktion der virker på vektorer af sandhedsværdier skal man bruge funktionen `ifelse` beskrevet i afsnit 19.3.

```
> if (x<2) 2
> if (x>2) 2
[1] 2
```

Man skal passe på hvis man (uden for en blok) deler et `if ... else` udtryk over flere linjer, idet udtrykket uden `else` i sig selv er et gyldigt lukket udtryk:

```
> if (x<2) 2
> else x
Error: syntax error in "else"
> if (x<2) 2 else
+ x
[1] 5
```

Det er ofte en god idé at lade de to grenudtryk være blokke i stedet for simple udtryk, så skal man bare være sikker på at have `else` på samme linje som den afsluttende krøllede parentes fra `if`-grenens blok:

```
> if (x<2) {
+   2
+ } else {
+   x
+ }
[1] 5
```

Bemærk at ovenstående kun gælder uden for blokke. Inden i en blok afslutter linjeskift *ikke* et `if`-udtryk hvis der følger en `else`-gren umiddelbart efter:

```
> { if (x<2) 2
+   else x }
[1] 5
```

19.2.1 Eksempler med `if ... else` udtryk

Et `if ... else` udtryk kan anvendes til at skrive matematiske funktioner, hvor der skal anvendes to forskellige udtryk afhængig af funktionsvariablens værdi, fx følgende funktion f (se venstre graf i figur 46):

$$f(x) = \begin{cases} 0, & \text{hvis } x < 1, \\ \frac{x-1}{3} & \text{ellers.} \end{cases}$$

Dette kan i R skrives således:

```
f <- function(x) { if (x<1) 0 else (x-1)/3 }
```

Hvis man har en definition med mere end to "grene" må man bruge flere `if ... else` udtryk indlejret i hinanden. Betragt for eksempel følgende funktion g (se højre graf i figur 46):

$$g(x) = \begin{cases} 0, & \text{hvis } x < 1, \\ \frac{x-1}{3}, & \text{hvis } 1 \leq x < 4, \\ 1 & \text{ellers.} \end{cases}$$

Dette kan i R skrives således:

```
g <- function(x) { if (x<1) 0 else if (x<4) (x-1)/3 else 1 }
```

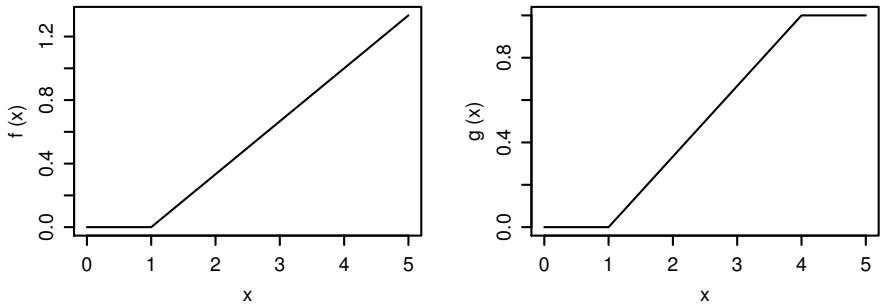
Ovenstående kan måske skrives mere læseligt ved at dele udtrykket over flere linjer:

```
g <- function(x) {
   if (x<1) 0
   else if (x<4) (x-1)/3
   else 1
}
```

For ikke at komme i tvivl om, hvilket `else`, der hører til hvilket `if`, er det bedst at pakke indlejrede `if ... else` udtryk ind i blokke:

```
g <- function(x) {
   if (x<0) 0
   else {
      if (x<4) (x-1)/3
      else 1
   }
}
```

Bemærk, at hvis man skal kunne tegne en graf for en funktion eller integrere den med `integrate` skal den kunne benyttes på *vektorer* af tal, så man fx kan skrive **`f(0:5)`** ligesom med R's indbyggede funktioner. Det kan man ikke med `if` udtryk i funktionen som i definitionerne af `f` og `g` oven for, man skal i stedet bruge funktionen `ifelse`, se afsnit 19.3.



Figur 46: Plot af funktionerne f (til venstre) og g (til højre) fra afsnit 19.2.1.

19.2.2 Eksempel med `if` udtryk: Primtal

I dette eksempel benyttes to `if`-udtryk (begge uden `else`).

Et naturligt tal n er et *primtal*, hvis det kun er deleligt (uden rest) med 1 og med n selv. Pr. definition er 1 dog ikke et primtal. De første ti primtal er 2, 3, 5, 7, 11, 13, 17, 19, 23 og 29.

En måde hvorpå man kan afgøre om et givet tal n er et primtal er ved at forsøge med hvert heltal m fra 2 og op til kvadratroden af n om det går op i n. Hvis der findes sådan et tal m er n *ikke* er et primtal.[20] Følgende funktion skrevet i R vil givet et naturligt tal `n` (større end 1) som argument returnere `TRUE`, hvis tallet er et primtal, og `FALSE` ellers (linjenumrene er ikke en del af funktionen – de er til brug for den efterfølgende beskrivelse):

```
1 primtal <- function(n) {
2    er.primtal <- TRUE
3    max.m <- floor(sqrt(n))
4    for(m in 2:max.m) {
5       if ( heltal(n/m) ) er.primtal <- FALSE
6    }
7    er.primtal
8 }
```

Forklaring:
Funktionen benytter en variabel `er.primtal` til at holde rede på, om der er fundet et tal der går op i

[20] Dette er langt fra den mest effektive måde hvorpå man kan afgøre om et tal n er et primtal, specielt ikke for store tal n. Men det er en simpel og intuitiv test og nem at implementere i fx R.

tallet `n`. I starten af funktionen, **linje 2**, sættes `er.primtal` til `TRUE`, og hvis `er.primtal` ikke er sat til `FALSE` inden vi når ned i bunden i linje 7 er `n` et primtal.
I **linje 3** beregnes det største tal, vi behøver at prøve med (kvadratroden af `n` rundet ned til nærmeste heltal med `floor`) og gemmes i `max.m`.
Løkken i **linje 4–6** afprøver for hvert tal `m` fra 2 til `max.m` om det skulle gå op i `n`.
Et tal `m` går op i `n` hvis `n/m` giver et heltal. Vi bruger funktionen `heltal`, som vi definerede i afsnit 19.1.1, og som vil give `TRUE`, hvis resultatet af divisionen `n/m` er et heltal, og `FALSE`, hvis det ikke er. Så. hvis betingelsen `heltal(n/m)` er *sand* er `n` *ikke* et primtal (da `m` går op) og i dette tilfælde ændres værdien af `er.primtal` så til `FALSE`.
Linje 7 er det sidste udtryk i funktionen og dermed dens *returværdi*. Så værdien i `er.primtal` efter løkken (enten `TRUE` eller `FALSE`) bliver det resultat, funktionen giver.

Eksempler på anvendelse af funktionen:

```
> primtal(21)
[1] FALSE
> primtal(23)
[1] TRUE
```

Et problem med denne funktion er, at sekvensen af tal `2:max.m` i linje 4 kun er korrekt, hvis `max.m` er mindst 2, dvs. hvis `n` er mindst 4. Hvis `n` er 3 eller 2 (begge primtal) bliver `max.m` 1 og sekvensen dermed `2:1`. Dvs. at vi også kommer til at udføre løkkekroppen for `m` med værdien 1, og da `n/m` dermed er et heltal kommer vi fejlagtigt til at sætte `er.primtal` til `FALSE`.

For at reparere dette kan vi pakke løkken ind i et `if`-udtryk og kun udføre den hvis `n` er større end 3, som følger:

```
primtal <- function(n) {
   er.primtal <- TRUE
   if (n > 3) {
      max.m <- floor(sqrt(n))
      for(m in 2:max.m) {
         if ( heltal(n/m) ) er.primtal <- FALSE
      }
   }
   er.primtal
}
```

Da 2 og 3 begge er primtal er det ikke noget problem at springe løkken over for de to tal, da værdien `TRUE` som `er.primtal` får i linje 2 da er korrekt. Det betyder så til gengæld at funktionen nu også giver `TRUE` hvis `n` er 1, men vi antager at man kun vil benytte funktionen til at afprøve om heltal *større* end 1 er primtal. Ellers kunne dette problem klares med et `if ... else` udtryk, eller mere elegant (men også mere kryptisk) ved at ændre linje 2 til:
`er.primtal <- n>1`

19.2.3 Eksempel med `if` udtryk: Fibonacci-tal

Som endnu et eksempel på anvendelse af `if` udtryk vil vi konstruere en funktion der kan give os Fibonacci-tal nummer n for heltal $n \geq 1$. Til beregningen vil vi benyttes os af fremgangsmåden fra afsnit 15.3 hvor vi beregnede Fibonacci-tal ved hjælp af en `for`-løkke; dér er det også forklaret hvad Fibonacci-tal overhovedet er.

For at funktionen skal svare i brugen til de indbyggede R-funktioner skal den også kunne bruges når n er en vektor, dvs. at for eksempel `fibonacci(c(1,5,7))` skal give en vektor med det første, femte og syvende Fibonacci-tal.

Her er så en R-funktion `fibonacci`, der for heltal $n \geq 1$ giver Fibonacci-tal nummer n (linjenumrene er ikke en del af funktionen men blot med for at lette beskrivelsen):

```
1  fibonacci <- function(n) {  # Fibonacci-tal for n>0.
2    max.n <- max(n)           # Hvor mange fibonacci-tal skal bruges?
3    fib <- rep(1, max.n)      # Lav en vektor med plads til max.n tal.
4    if ( max.n > 2 ) {
5      for ( i in 3:max.n ) {
6         fib[i] <- fib[i-1] + fib[i-2]   # Udregn Fibonacci-tal nummer i.
7      }
8    }
9    fib[n]                    # Returner de(t) n'te Fibonacci-tal.
10 }
```

Princippet i funktionen er at udregne Fibonacci-tallene fra nummer 1 og op til de(t) største Fibonacci-tal, der skal beregnes. Undervejs skal Fibonacci-tallene gemmes i en vektor `fib` således at `fib[1]` er Fibonacci-tal nummer 1, `fib[2]` er Fibonacci-tal nummer 2, og så videre. Resultatet af funktionskaldet er så `fib[n]`, som giver elementerne fra de(n) plads(er) i `fib` der svarer til numrene i `n`.

Virkemåden i detaljer: Nummeret på det største Fibonacci-tal, der skal beregnes, findes i **linje 2** som `max(n)`, dvs. det største tal i vektoren `n` (husk at et tal i R repræsenteres som en vektor med ét element, så dette virker også hvis `n` bare er et enkelt tal). Dette største tal gemmes i variablen `max.n`. Vektoren `fib` oprettes i **linje 3** med `max.n` pladser, alle sat til 1. Da Fibonacci-tal nummer 1 og 2 begge er 1 er `fib[1]` og `fib[2]` hermed allerede korrekte. Kun hvis Fibonacci-tal nummer 3 eller højere skal udregnes udføres `for`-løkken i **linje 5–7** for at beregne tallene. Det er `if`-udtrykket i **linje 4** som sørger for at løkken ikke udføres ellers (dette ville nemlig give fejl). Løkkekroppen i **linje 6** udføres for hvert tal `i` mellem 3 og `max.n` og beregner `fib[i]` som summen af de to foregående tal i `fib`. Efter løkken indeholder `fib` de første `max.n` Fibonacci-tal. I **linje 9** udtages de(n) ønskede værdi(er) fra `fib` med indekseringen `fib[n]` (jvf. afsnit 14.2). Dette sidste udtryk i funktionens krop specificerer funktionens *returværdi*, dvs. den værdi man "får ud" når man anvender funktionen — se afsnit 18 for en nærmere forklaring.

Eksempler på brug:

```
> fibonacci(6)
[1] 8
> fibonacci(1:10)
[1]  1  1  2  3  5  8 13 21 34 55
```

Ovenstående beregner først det sjette Fibonacci-tal og derefter de 10 første Fibonacci-tal.

19.3 Betingede udtryk: `ifelse`

Et `if ... else` udtryk som beskrevet i det foregående er beregnet til at operere på kun en enkelt sandhedsværdi i betingelsen. Hvis betingelsen giver en vektor med mere end én sandhedsværdi vil kun det første element blive brugt og R vil ydermere give en advarsel. Dette betyder at funktionerne `f` og `g` fra afsnit 19.2.1 i modsætning til R's indbyggede funktioner *ikke* virker på en vektor med mere end ét tal. Her er funktionerne gengivet igen:

```
f <- function(x) { if (x<1) 0 else (x-1)/3 }

g <- function(x) {
   if (x<1) 0
   else if (x<4) (x-1)/3
   else 1
}
```

Hvis nu funktionerne kaldes med en vektor som argumkent vil der dels blive genereret en masse advarsler fordi en betingelse som `(x<1)` giver et vektorresultat når `x` er en vektor, fx `c(TRUE, FALSE, FALSE)` hvis `x` er talsekvensen `0:2`. Dels vil det alene være det første tal i argumentvektoren der vil komme til at styre hvilken gren og dermed hvilket resultat, der vælges. Det betyder fx at `f(0:2)` simpelthen giver tallet `0`, da den første gren i `if` udtrykket vælges, mens `g(c(1,7,-2))` giver vektoren `c(0,2,-1)` (hvilket kun er korrekt i første element), da grenen `(x-1)/3` vælges.

For at lave implementationer af f og g, der virker på vektorer af tal lige som R's interne funktioner, kan man i stedet benytte funktionen `ifelse`. Det er en funktion der tager tre (vektor) argumenter:

`ifelse(` v_1 `,` v_2 `,` v_3 `)`

Det første argument v_1 skal være vektor af logiske værdier og det andet og tredie to vektorer af samme længde som v_1 (omend den sædvanlige genbrugsregel gælder hvis den ene er for kort). Resultatet er en vektor af samme længde som v_1 sammensat af elementer fra v_2 og v_3 således at hvor v_1 er `TRUE` tages elementer fra v_2 og hvor v_1 er `FALSE` tages elementer fra v_3. Eksempel:

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> 10-x
 [1] 9 8 7 6 5 4 3 2 1 0
> ifelse(x>5, x, 10-x)
 [1]  9  8  7  6  5  6  7  8  9 10
```

Vi kan nu implementere f og g på en bedre måde end før:

```
f <- function(x) { ifelse(x<1, 0, (x-1)/3) }
g <- function(x) { ifelse(x<1, 0, ifelse(x<4, (x-1)/3, 1)) }
```

Nu kan man kalde funktionerne med vektorer som argument og få det rigtige resultat:

```
> f(0:2)
[1] 0.0000000 0.0000000 0.3333333
> g(0:5)
[1] 0.0000000 0.0000000 0.3333333 0.6666667 1.0000000 1.0000000
```

Man kan nu også bruge funktionerne i `plot` eller `integrate`:

```
> integrate(g,0,5)
2.5 with absolute error < 2.8e-14
> plot(g, 0, 5)
```

Se det resulterende plot af g i figur 46 (højre plot).

19.4 Rekursion

En vigtig anvendelse af `if ... else` udtryk er til at skrive *rekursive funktioner*. Rekursiv problemløsning er en "del og hersk" strategi. "Del og hersk" går ud på at opdele et stort problem i mindre delproblemer, løse de mindre delproblemer og kombinere løsningerne herfra til en løsning på det oprindelige, store problem. *Rekursion* kan benyttes når de mindre delproblemer er af samme slags som det overordnede problem. For at kunne løse et problem rekursivt skal vi bruge to "opskrifter":

1. En opskrift på hvordan den mest simple "grundudgave" af problemet kan løses direkte.

2. En opskrift på hvordan en større udgave af problemet i det generelle tilfælde kan reduceres til en eller flere mindre udgaver af problemet, hver især tættere på grundudgaven, og hvordan løsningerne fra disse kan sættes sammen til en løsning på det større problem.

Som eksempel der intet har med R at gøre kan man opstille en rekursiv strategi til at finde ud af hvem ud af en given gruppe mennesker, der er bedst til at spille tennis. Vi deler op i et grundtilfælde og det generelle tilfælde som følger:

1. Grundtilfældet er når der kun er én person i gruppen. Denne person er da trivielt den bedste i gruppen til at spille tennis.

2. Hvis der er mere end én person, opdeles gruppen i to mindre grupper, A og B. Vi lader så den person fra gruppe A, der er bedst til tennis, spille en kamp mod den person fra gruppe B, der er bedst til tennis. Vinderen af kampen vil være den person der er bedst til tennis ud af hele den oprindelige gruppe.

Uanset antallet af personer i gruppen vil vi kunne anvende et af tilfældene 1 eller 2 ovenfor. "Del og hersk" princippet og rekursionen kommer ind i tilfælde 2 når vi skal finde den person fra gruppe A henholdsvis den person fra gruppe B der er bedst til tennis. Da antallet af personer i såvel gruppe A som gruppe B er mindre end antallet i den oprindelige gruppe har vi reduceret problemet til to delproblemer af mindre omfang end det oprindelige problem, som begge kan løses efter opskrifterne ovenfor. Da vi ved videre opdeling af grupperne i tilfælde 2 altid til sidst vil nå ned til grundtilfældet med grupper á én person er vi sikret at strategien vil finde en løsning. Så cup-systemet (som det foregående faktisk er en beskrivelse af) kan altså betragtes som en rekursiv løsning af problemet at finde den bedste spiller eller det bedste hold i en sportsturnering.

Hvor kommer så anvendelsen af `if ... else` udtryk ind i dette? Som vi skal se i det følgende eksempel kommer de ind når vi i en rekursiv defineret funktion i R skal skelne mellem grundtilfældet og det generelle tilfælde.

19.4.1 Eksempel: Fakultetsfunktionen

Som eksempel på anvendelse af et `if ... else` udtryk i en rekursiv funktion vil vi implementere fakultetsfunktionen $x!$, defineret for naturlige tal x som

$$x! = \begin{cases} 1, & \text{hvis } x = 1 \text{ (grundtilfældet)}, \\ x \cdot (x-1)! & \text{ellers (generelt tilfælde)}. \end{cases}$$

Bemærk at definitionen er *rekursiv*: $x!$ er defineret ved hjælp af $(x-1)!$, så fakultetsfunktionen er defineret ved hjælp af sig selv. Vi følger definitionen nøje og implementerer funktionen `fak` rekursivt:

```
> fak <- function(x) { if (x==1) 1 else x*fak(x-1) }
> fak(4)
[1] 24
```

Forgreningen i grundtilfældet $x = 1$ og det generelle tilfælde $x > 1$ implementerede vi altså med et `if ... else` udtryk.

For til fulde at forstå hvordan en rekursiv funktion virker kan man med fordel *håndkøre* den for en passende lille værdi af argumentet/argumenterne (for at få et lille antal rekursive kald). Håndkørsel vil sige at bruge blyant og papir og "lege R" ved at skrive beregningerne ud i hånden. Håndkørsel af **`fak(4)`** kan se således ud:

Håndkørsel:

`fak(4) =`	$x = 4 \neq 1$, dvs. else-grenen vælges:
`4*fak(3) =`	$x = 3 \neq 1$, dvs. else-grenen vælges igen:
`4*3*fak(2) =`	$x = 2 \neq 1$, dvs. else-grenen vælges igen:
`4*3*2*fak(1) =`	$x = 1$, dvs. if-grenen vælges:
`4*3*2*1 =`	herfra skal der bare ganges ud:
`24`	

Lige som med løkker kan det nogle gange være en fordel at sætte `print`-kald ind i en rekursiv funktion for at finde fejl eller bare forstå hvad der foregår. Her er fakultetsfunktionen igen, denne gang med `print` af x og af hvilken gren, der vælges:

```
fak <- function(x) {
   print(x)
   if (x==1) {
      print("if-gren valgt")
      1
   } else {
      print("else-gren valgt")
      x*fak(x-1)
   }
}
```

Bemærk at det kræver lidt omtanke at placere `print`-kald i en funktion som denne: kaldene skal stå *før* de udtryk, der beregner selve funktionsværdien (da værdien af en blok er værdien af det *sidste* udtryk i blokken). Nu får vi en masse at vide når vi skriver **`fak(4)`**:

```
> fak(4)
[1] 4
[1] "else-gren valgt"
[1] 3
[1] "else-gren valgt"
[1] 2
[1] "else-gren valgt"
[1] 1
[1] "if-gren valgt"
[1] 24
```

Det sidste `24` kommer ikke fra `print` men er resultatet fra funktionskaldet **`fak(4)`**, der vises som sædvanligt.

Implementationen af `fak` ved hjælp af et `if ... else` udtryk har den styrke at det er en meget direkte oversættelse af den matematiske notation men den svaghed at funktionen ikke som R's indbyggede funktioner kan bruges på vektorer af tal. For at opnå dette må vi skifte `if ... else` udtrykket ud med et `ifelse`, lige som i afsnit 19.3. En direkte oversættelse giver følgende funktionsdefinition:

```
fak <- function(x) { ifelse(x==1, 1, x*fak(x-1)) }
```

Dette vil imidlertid *ikke* fungere hvis argumentet til funktionen består af mere end ét tal. Hvorfor kan man indse ved at forsøge at håndkøre fx **`fak(1:5)`**. For at beregne det tredje argument i `ifelse` kaldet vil `fak` blive kaldt igen rekursivt, hvilket resulterer i en uendelig følge af kald af `fak`:

```
fak(c(1,2,3,4,5))
  fak(c(0,1,2,3,4))
    fak(c(-1,0,1,2,3))
      fak(c(-2,-1,0,1,2))
        fak(c(-3,-2,-1,0,1))
          ⋱
```

Denne opførsel ville generelt umuliggøre at man overhovedet implementerer en rekursiv funktion med `ifelse`, men heldigvis gælder følgende regel: For udtrykket

`ifelse(` v_1 `,` v_2 `,` v_3 `)`

gælder at hvis betingelsen v_1 er `TRUE` i alle elementer evalueres v_3 slet ikke. Tilsvarende vil v_2 ikke blive evalueret hvis v_1 er `FALSE` i alle elementer. Kun hvis v_1 har en blanding af sande og falske elementer evalueres både v_2 og v_3.

Så for at sikre at rekursionen stopper i `fak` skal betingelsen `x==1`, der i eksemplet ovenfor aldrig kan opfyldes for alle elementer samtidig og dermed bevirker at `fak` kaldes igen og igen, skiftes ud. En brugbar definition af `fak` er følgende:

```
fak <- function(x) { ifelse(x<=1, 1, x*fak(x-1)) }
```

Betingelsen `x<=1` vil på et tidspunkt blive opfyldt af alle elementerne i argumentvektoren samtidig og dermed stopper de rekursive kald af `fak`, idet det første argument til `ifelse` så er `TRUE` i alle elementer og tredje argument ikke behøver at blive evalueret. Funktionen kan nu bruges også på vektorer:

```
> fak(1:5)
[1]   1   2   6  24 120
```

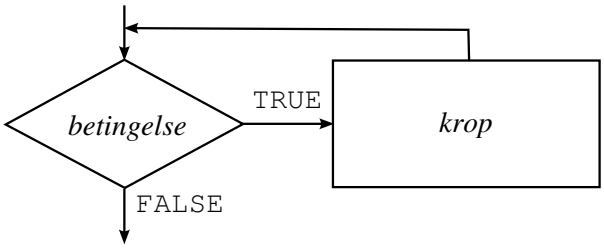
For en god ordens skyld skal nævnes at R allerede har en indbygget fakultetsfunktion: den hedder `factorial`.

19.5 `while`-løkker

I afsnit 15 beskrev vi iteration over elementerne i en vektor med `for`-løkker. `For`-løkker er nemme at bruge når noget skal gentages et bestemt antal gange, man kender på forhånd. En mere generel form for løkke er `while`-løkken. Den kan bruges til at foretage samme operation gentagne gange så længe en given betingelse er opfyldt – dvs. antallet af gentagelser behøver ikke at være kendt på forhånd. Et `while` udtryk har følgende form:

`while` (*betingelse*) *krop*

Effekt: Udtrykket *betingelse* evalueres. Hvis det giver `TRUE` evalueres udtrykket *krop* (kaldet løkkekroppen). Dette gentages, så længe *betingelse* giver `TRUE`. Når *betingelse* giver `FALSE` stopper evalueringen af `while`-løkken. Værdien af et `while` udtryk er værdien af den sidste evaluering af kroppen (og `NULL` hvis kroppen aldrig evalueres – altså hvis betingelsen var falsk fra start). Figur 47 viser et flowdiagram for en `while`-løkke.



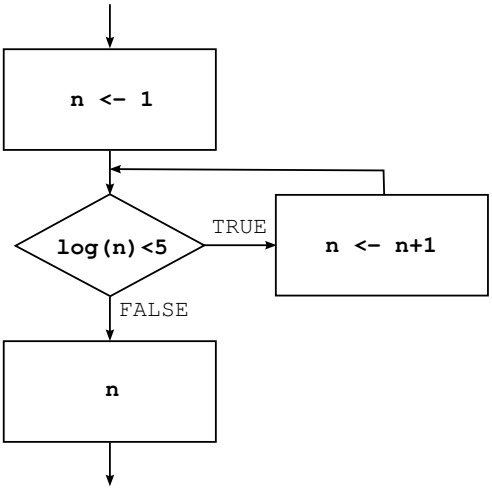
Figur 47: Princip for en `while`-løkke. Følg pilene, startende fra oven.

Eksempel: Find det mindste heltal n så $\ln(n)$ er mindst 5:

```
> n <- 1
> while(log(n)<5) { n <- n+1 }
> n
[1] 149
```

Her starter `n` med at være 1 og tælles op med 1 i hvert gennemløb af løkken. Når løkken stopper må `log(n)` være større end eller lig 5 (idet løkken skal fortsætte med at køre så længe `log(n)<5`). Da har vi fundet det søgte n, der viser sig at være 149. Figur 48 viser eksemplet i diagramform.

Kontrol af løsningen:



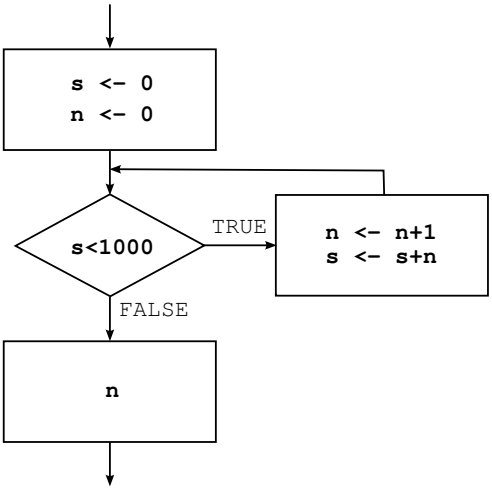
Figur 48: Diagram for `while`-løkke der finder det mindste heltal n således at $\ln(n)$ er mindst 5. Følg pilene, startende fra oven.

```
> log(148)
[1] 4.997212
> log(149)
[1] 5.003946
```

Eksempel: Find det mindste heltal n så summen af tallene fra 1 til n giver mindst 1000:

```
> s <- 0 ; n <- 0
> while(s<1000) { n <- n+1; s <- s+n }
> n
[1] 45
```

Her tælles `n` op med 1 i hvert gennemløb af løkken og i `s` akkumuleres løbende summen af `n`'erne. Når løkken stopper må `s` være mindst 1000, idet løkken skal fortsætte så længe `s<1000`. Hermed indeholder `n` det søgte tal n, der viser sig at være 45. Figur 49 viser eksemplet i diagramform.



Figur 49: Diagram for `while`-løkke der finder det mindste heltal n således at summen af tallene fra 1 til n giver mindst 1000.

Kontrol af løsningen:

```
> sum(1:44)
[1] 990
> sum(1:45)
[1] 1035
```

OBS: Det er let med `while` at komme til at lave en løkke, der aldrig stopper (en *uendelig løkke*). Man kan altid kan stoppe den igangværende beregning i R ved at klikke på det lille stop-skilt eller taste Esc i R Console (efterfulgt af ↩ i MacOS X udgaven af R). I en Linux terminal skal man i stedet taste Ctrl+C.

19.5.1 Fejlfinding i `while`-løkker

Lige som med `for`-løkker kan det være svært at gennemskue hvorfor en given `while`-løkke ikke fungerer. Teknikkerne til at finde fejl i (eller bare forstå) `while`-løkker er de samme som blev præsenteret for `for`-løkker i afsnit 15.1: håndkørsel og indsættelse af `print`-kald.

Håndkørsel med papir og blyant hvor man "leger R" og i hånden udfører de samme beregninger som R gør når løkken køres er det bedste redskab til at forstå løkken til bunds. En sådan håndkørsel af en `while`-løkken der finder det mindste heltal n således at summen af tallene fra 1 til n giver mindst 10 kan se således ud:[21]

Selve koden:

```
s <- 0 ; n <- 0
while(s<10) { n <- n+1; s <- s+n }
```

Håndkørsel:

`s <- 0 ; n <- 0`	før løkken er $s = 0$ og $n = 0$
`while(s<10)`	Er $0 < 10$? Ja, vi udfører løkkekroppen:
`n <- n+1 ; s <- s+n`	nu bliver $n = 0+1 = 1$ og $s = 0+1 = 1$
`while(s<10)`	Er $1 < 10$? Ja, vi udfører løkkekroppen igen:
`n <- n+1 ; s <- s+n`	nu bliver $n = 1+1 = 2$ og $s = 1+2 = 3$
`while(s<10)`	Er $3 < 10$? Ja, vi udfører løkkekroppen igen:
`n <- n+1 ; s <- s+n`	nu bliver $n = 2+1 = 3$ og $s = 3+3 = 6$
`while(s<10)`	Er $6 < 10$? Ja, vi udfører løkkekroppen igen:
`n <- n+1 ; s <- s+n`	nu bliver $n = 3+1 = 4$ og $s = 6+4 = 10$
`while(s<10)`	Er $10 < 10$? Nej, løkken er færdig!

Resultatet er altså $n = 4$.

Vi kan også lade R om at køre løkken og så indsætte et passende `print`-kald så vi kan følge med i udviklingen af `n` og `s`:

```
> s <- 0 ; n <- 0
> while(s<10) { n <- n+1; s <- s+n ; print(c(n,s)) }
[1] 1 1
[1] 2 3
[1] 3 6
[1]  4 10
```

Vi udskriver her i hvert gennemløb af løkken, *efter* at `n` og `s` har fået tildelt nye værdier, vektoren `c(n,s)`. Hermed kan vi klart se udviklingen i de to variable i løbet af udførelsen af løkken.

[21] Princippet er det samme som i løkken fra før hvor summen af tallene skulle give mindst 1000, men da dette ville kræve en meget lang håndkørsel sætter vi grænsen ned til 10.

Hvis en løkke, man ønsker at følge udførelsen af v.h.a. `print`, tager lang tid, vil man opleve at alt hvad der udskrives med `print` først bliver vist samlet til sidst når løkken *er* færdig. Dette er faktisk også tilfældet hvis løkken kører hurtigt, men der lægger man bare ikke mærke til det. Forklaringen på dette er at output til R Console af effektivitetshensyn opsamles i en buffer og først vises, når enten bufferen er fuld eller når R kommer med en prompt. Man kan imidlertid tvinge R til at tømme bufferen og vise indholdet med funktionskaldet `flush.console()`. Dette kald indsætter man så lige efter `print`, fx som her:

```
s <- 0 ; n <- 0
while(s<10) { n <- n+1; s <- s+n ; print(c(n,s)) ; flush.console() }
```

19.5.2 Eksempel: Kaninpopulation

Som første lidt større eksempel på brug af `while` vil vi tage modellen for en kaninpopulation fra afsnit 16.9 og finde ud af, hvor hurtigt to kaniner bliver til tusind. Vi har modellen på matrixform som følger:

$$\mathbf{v}_t = \begin{pmatrix} x_t \\ y_t \end{pmatrix}, \quad \mathbf{M} = \begin{pmatrix} 2.0 & 1.5 \\ 0.7 & 0.4 \end{pmatrix}, \quad \mathbf{v}_{t+1} = \mathbf{M}\mathbf{v}_t, \quad \mathbf{v}_0 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

I R:

```
M <- matrix(c(2.0, 0.7, 1.5, 0.4), 2)
v0 <- c(2,0)
v <- v0
```

Det totale antal kaniner i populationen er summen `v[1]+v[2]`, eller simpelthen `sum(v)`. Vi skriver nu en `while`-løkke til at fremskrive populationen ind til denne sum er mindst 1000. Undervejs må vi holde rede på antallet af generationer; til dette benytter vi en variabel `antal` som starter med at være nul og som tælles op med én for hver generation:

```
antal <- 0
while(sum(v)<1000) { antal <- antal+1 ; v <- M%*%v  }
```

Efter udførelse af løkken ved vi at betingelsen `sum(v)<1000` må være falsk (ellers ville løkken fortsætte), så derfor må der være mindst 1000 kaniner. Vi kan nu se nærmere på antallet af generationer og den endelige kaninpopulation:

```
> antal
[1] 7
> v
          [,1]
[1,] 985.9525
[2,] 328.6508
> sum(v)
[1] 1314.603
```

Vi ser at der er over 1000 kaniner efter 7 generationer (hvor der er cirka 1315 kaniner). Hvis vi gerne ville have set hvor mange kaniner der var efter hver generation skulle vi have opsamlet værdierne af `v` undervejs, ligesom vi gjorde i afsnit 16.9.

19.5.3 Eksempel: Primtal, nu med `while`

I afsnit 19.2.2 præsenteredes en funktion, der bruger bl.a. en `for`-løkke til at afgøre om et tal er et primtal. Her gengives funktionen igen (i sin første form):

```
primtal <- function(n) {
   er.primtal <- TRUE
   max.m <- floor(sqrt(n))
   for(m in 2:max.m) {
      if ( heltal(n/m) ) er.primtal <- FALSE
   }
   er.primtal
}
```

Funktionen er ikke særlig effektiv, for selv når vi *har* fundet et tal `m`, der går op i `n` (og vi derfor ved, at `n` ikke kan være et primtal), fortsætter vi med at prøve også resten af tallene op til og med `max.m`. Det vil være bedre at stoppe så snart vi ved at `n` ikke er et primtal, og til dette skal vi bruge en `while`-løkke. I første omgang kan vi simpelt hen omskrive `for`-løkken til en `while`:

```
1  primtal <- function(n) {
2     er.primtal <- TRUE
3     max.m <- floor(sqrt(n))
4     m <- 2
5     while(m <= max.m) {
6        if ( heltal(n/m) ) er.primtal <- FALSE
7        m <- m + 1
8     }
9     er.primtal
10 }
```

Sekvensen af `m`-værdier fra 2 til `m.max` bliver nu genereret ved dels at sætte `m` til startværdien 2 *før* løkken (**linje 4**), dels at tælle `m` op med 1 i hvert gennemløb (**linje 7**) og dels at fortsætte så længe `m` er mindre end eller lig `max.m` (betingelsen i `while` udtrykket, **linje 6**). I øvrigt er løkken uforandret og den vil da også fortsætte helt op til `max.m`, nøjagtig som `for`-løkken gjorde.[22] Det er imidlertid kun en lille ændring i betingelsen, der skal til for at få løkken til at stoppe så snart `er.primtal` bliver `FALSE`:

```
1  primtal <- function(n) {
2     er.primtal <- TRUE
3     max.m <- floor(sqrt(n))
4     m <- 2
5     while(m <= max.m && er.primtal) {
6        if ( heltal(n/m) ) er.primtal <- FALSE
7        m <- m + 1
8     }
9     er.primtal
10 }
```

Nu bliver betingelsen i linje 5 falsk (og løkken stopper) så snart det første tal, der går op i `n`, er fundet.

Det kan faktisk gøres yderligere en lille smule kortere (og hurtigere). Det er nu sådan, at kun hvis løkken kører "hele vejen" og ender med at stoppe fordi `m` bliver for stor, er `n` et primtal. Derfor kan returværdien `er.primtal` i linje 9 erstattes med udtrykket `m > max.m`. Vi kan så flytte afprøvningen af, om `m` går op i `n`, op i selve betingelsen i `while`-løkken og helt undvære variablen `er.primtal`, som følger:

[22] Der ér faktisk en forskel: Når `n` er 2 eller 3 og `max.m` dermed bliver 1 vil `while`-udgaven virke, hvilket `for`-udgaven ikke gjorde. I `while`-udgaven vil betingelsen `m <= max.m` nemlig *ikke* være opfyldt fra start og løkken vil derfor blive helt sprunget over. I `for`-udgaven blev løkken kørt med `m` værdierne 2 og 1, og da 1 går op i alle heltal ville såvel 2 som 3 fejlagtigt give `FALSE` (ikke blive set at være primtal). I afsnit 19.2.2 på side 88 var det derfor nødvendigt med et ekstra `if`.

```
1 primtal <- function(n) {
2    max.m <- floor(sqrt(n))
3    m <- 2
4    while(m <= max.m && ! heltal(n/m) ) { m <- m + 1 }
5    m > max.m
6 }
```

Nu kører `while` løkken i linje 4 simpelt hen så længe at `m` er mindre end eller lig `max.m` *og* `m` *ikke* går op i `n` (bemærk negationen: `! heltal(n/m)`). Hvis `m > max.m` i linje 5 efter løkken var der intet `m` vi prøvede, der gik op, og funktionen giver `TRUE` (`n` er et primtal). Ellers gik det senest prøvede tal `m` op i `n` uden rest og funktionen giver `FALSE` (`n` er ikke et primtal).

19.5.4 Eksempel: Gæt et tal

Som endnu et eksempel på anvendelse af `while` vil vi lave et lille "gæt et tal" spil. I spillet vælger computeren et tal mellem 1 og 100 som man skal gætte i færrest mulige forsøg. Man får ved hvert gæt at vide om man har gættet for højt eller lavt, således at man kan skyde sig ind på resultatet.

Spillet kan implementeres som en funktion i R således (linjenumrene er ikke en del af funktionen – de er til brug for den efterfølgende beskrivelse):

```
1  gæt.et.tal <- function() {
2     tal <- floor(runif(1, 1, 101))
3     antal.gæt <- 0
4     gæt <- 0
5     while(gæt!=tal) {
6        indtastning <- readline("Indtast et gæt mellem 1 og 100: ")
7        gæt <- as.numeric(indtastning)
8        if(gæt < tal) { cat("Nej, tallet er større end", gæt, ".\n") }
9        if(gæt > tal) { cat("Nej, tallet er mindre end", gæt, ".\n") }
10       antal.gæt <- antal.gæt+1
11    }
12    cat("Du brugte", antal.gæt, "gæt.\n")
13 }
```

Forklaring:
I **linje 2** genereres et tilfældigt heltal mellem 1 og 100, som skal gættes. Med `runif(1,1,101)` genereres et tilfældigt (komma)tal i intervallet]1, 101[og derefter rundes det *ned* til nærmeste heltal med `floor`. Resultatet gemmes i variablen `tal`.
I **linje 3–4** oprettes to variable og de gives startværdier. Variablen `antal.gæt`, som indeholder det antal gæt, spilleren hidtil har brugt, sættes til nul fra start. Variablen `gæt`, som skal indeholde spillerens seneste gæt, sættes til nul fra start, hvilket med sikkerhed er forskellig fra det genererede tilfældige tal og derfor garanterer at betingelsen i `while`-løkken umiddelbart efter starter med at være opfyldt.
Linje 5–11 er en `while`-løkke, der kører så længe spillerens gæt er forskellig fra tallet, der skal gættes.
I **linje 6** benyttes funktionen `readline` til at "læse" en indtastning fra tastaturet. I R Console vises teksten "Indtast et tal mellem 1 og 100:" (argumentet til `readline`) og spilleren skal så indtaste et tal og afslutte ved at taste ↵. Den indtastede tekst gemmes i variablen `indtastning`.
Da `readline` indlæser en *tekst* og vi skal bruge et *tal* konverteres indtastningen i **linje 7** til et tal med funktionen `as.numeric` og gemmes i variablen `gæt`.
Linje 8–9 er to `if`-udtryk (uden `else`), der giver en meddelelse i R Console hvis det gættede tal er for lille henholdsvis for stort. R-funktionen `cat` skriver sine argumenter ud i R Console i "råt format", til forskel fra `print` der formaterer udskriften så det svarer til den måde R plejer at vise resultater på. Tegnsekvensen "`\n`", der forekommer i teksten, der skal skrives ud, betyder linjeskift. Dette er nødvendigt at

have med fordi `cat` ikke automatisk indsætter nogle linjeskift, så hvis ikke de blev angivet eksplicit ville `cat` skrive det hele ud i én køre.
Til sidst i `while`-løkkens krop (i **linje 10**) tælles antallet af benyttede gæt op med ét. Da dette sker i hvert gennemløb vil variablen `antal.gæt` hele tiden svare til antallet af tal, spilleren har indtastet.
Efter løkken, i **linje 12**, har spilleren gættet tallet og derfor udskrives (igen med `cat`) det benyttede antal gæt.

Spillet startes ved at kalde funktionen `gæt.et.tal`:

```
> gæt.et.tal()
Indtast et gæt mellem 1 og 100: 50
Nej, tallet er mindre end 50 .
Indtast et gæt mellem 1 og 100: 25
Nej, tallet er mindre end 25 .
Indtast et gæt mellem 1 og 100: 12
Nej, tallet er mindre end 12 .
Indtast et gæt mellem 1 og 100: 6
Nej, tallet er større end 6 .
Indtast et gæt mellem 1 og 100: 9
Du brugte 5 gæt.
```

19.5.5 Eksempel: En funktion til potensopløftning af matricer

Som vi så tidligere kan man ikke skrive potensopløftning af en kvadratisk matrix `A` med den naturlige hat-notation; `A^2` giver ikke A^2, altså `A` matrixmultipliceret med sig selv, men derimod en matrix hvor hvert element er kvadratet af det tilsvarende element i `A`.

Vi vil nu definere en funktion `matpow(A,n)` der beregner A^n, det vil sige `A` matrixmultipliceret med sig selv `n` gange (hvor `n` er et heltal). Vi bemærker at A^0 = `matpow(A,0)` giver en enhedsmatrix af samme dimensioner som `A`. I første omgang begrænser vi os til $n \geq 0$.

```
1  matpow <- function(A, n) { # matrix-potensopløftning for heltal n>=0
2    res <- diag(ncol(A))     # start: enhedsmatrix
3    while (n > 0) {
4      res <- res %*% A       # gang A på resultatet, n gange
5      n <- n-1
6    }
7    res                      # det færdige resultat
8  }
```

I ovenstående opbygger vi resultatet i variablen `res`. Vi starter med i **linje 2** at sætte `res` lig enhedsmatricen af samme størrelse som `A` (funktionen `ncol` giver som nævnt i afsnit 16.7 antal søjler i en matrix). Kroppen af `while`-løkken **linje 3–6** udføres `n` gange og hver gang sættes `res` i **linje 4** lig den gamle værdi af `res` matrixmultipliceret med `A`. For at holde rede på hvor mange gange løkken udføres trækkes der 1 fra `n` hver gang, i **linje 5**.

I **linje 7**, efter løkken, har vi

$$\texttt{res} = E\,\underbrace{\texttt{A}\cdots\texttt{A}}_{\texttt{n}} = \texttt{A}^{\texttt{n}}$$

og dette bliver funktionens returværdi. Bemærk, at hvis `n` er nul udføres løkkekroppen nul gange, hvorved `res` forbliver enhedsmatricen (dette er grunden til at vi ikke kan anvende en `for`-løkke).

Eksempler på brug:

```
> A <- matrix(c(1,2,3,4),2)
> matpow(A,0)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> matpow(A,1)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> matpow(A,2)
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```

Ofte bruger man notationen A^{-1} for den inverse af A. Vi kan definere $A^{-n} = (A^{-1})^n$ for negative heltal $-n$. Dermed kan vi generalisere `matpow` til også at håndtere negative potenser af matricer:

```
1  matpow <- function(A, n) { # matrix-potensopløftning for heltal n
2    if (n < 0) {
3      n <- -n                # for negativ potens, vend n's fortegn
4      A <- solve(A)          # og sæt A til den inverse af A
5    }
6    res <- diag(ncol(A))     # start: enhedsmatrix
7    while (n > 0) {
8      res <- res %*% A       # gang A på resultatet, n gange
9      n <- n-1
10   }
11   res                      # det færdige resultat
12 }
```

Ændringen i forhold til før er alene indførelse af `if`-udtrykket i **linjerne 2–5**. Hvis `n` er negativ ændres `A` til at være sin egen inverse (dvs. A^{-1}) og fortegnet på `n` vendes. Derefter er beregningen (og koden) fuldstændig den samme som før.

Eksempler på brug:

```
> matpow(A,-1)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
> matpow(A,-2)
     [,1]  [,2]
[1,]  5.5 -3.75
[2,] -2.5  1.75
```

19.5.6 Eksempel: En funktion til iteration af funktioner

Hvis f er en funktion og $n \geq 0$ er et heltal, kan en ny funktion $f^n(x)$ defineres som resultatet af n gange at bruge f på x:

$$f^n(x) = \underbrace{f(\ldots(f}_{n}(x))\ldots)$$

Specielt gælder at $f^0(x) = x$ og $f^1(x) = f(x)$ og $f^2(x) = f(f(x))$. Denne form for iteration af funktioner kan i R programmeres på samme måde som potensopløftning for matricer i foregående afsnit.

Funktionen `funpow(f, n, x)` beregner $f^n(x)$ og kan defineres sådan her:

```
funpow <- function(f, n, x) {
  res <- x
  while (n > 0) {
    res <- f(res)
    n <- n-1
  }
  res
}
```

Til forskel fra `matpow` kan man dog ikke generelt udvide `funpow(f,n,x)` til at virke for negative `n`. Det ville kræve at der var en generel måde at finde den inverse f^{-1} til en given funktion f, og det er der ikke.

Som eksempel på brug af `funpow` kan vi betragte pensionsopsparing med en årlig rente på 4% og et årligt indskud på 41000 kroner. Funktionen $f(s) = s + 0.04s + 41000$ beregner næste års saldo $f(s)$ som summen af dette års saldo s, med renten og det årlige indskud konstant. Hvis man starter med 0 kroner og sparer op på den måde i 35 år, så får man $f^{35}(0)$ kroner, godt 3 millioner kroner:

```
> f <- function(s) { s + 0.04*s + 41000 }
> funpow(f, 35, 0)
[1] 3019741
```

Hvis man vil se hvordan opsparingen forløber over de første 7 år kan man beregne $f^0(0), f^1(0), \ldots, f^7(0)$ ved at bruge funktionen `sapply` (afsnit 14.3) til at beregne $f^n(0)$ for $n = 0, \ldots, 7$ sådan her:

```
> sapply(0:7, function(n) { funpow(f, n, 0) })
[1]      0.0  41000.0  83640.0 127985.6 174105.0 222069.2 271952.0 323830.1
```

Dette resultat er nemmere at aflæse hvis man parrer saldoen med årnummeret `n`. Det kan man gøre sådan:

```
> sapply(0:7, function(n) { c(n, funpow(f, n, 0)) } )
     [,1]  [,2]  [,3]     [,4]     [,5]     [,6]  [,7]     [,8]
[1,]    0     1     2      3.0      4.0      5.0     6      7.0
[2,]    0 41000 83640 127985.6 174105.0 222069.2 271952 323830.1
```

Hvis resultatet er langt, bliver det mere overskueligt hvis man transponerer det med R-funktionen `t`:

```
> t(sapply(0:35, function(n) { c(n, funpow(f, n, 0)) } ))
      [,1]      [,2]
 [1,]    0       0.0
 [2,]    1   41000.0
  ...
[35,]   34 2864174.2
[36,]   35 3019741.2
```

Kombinationen af `sapply` og `funpow` er særdeles nyttig, og virker også i forbindelse med funktioner f der tager argumenter og producerer resultater som er matricer eller vektorer. Nedenstående er en simpel nationaløkonomisk model, hvor funktionen f som argument tager en søjlevektor der beskriver forbrug C og investeringer I i et givet år, og som resultat giver en søjlevektor der viser næste års forbrug og investeringer (se Anvendelseseksempel B.5 i *Noter om matematik*):

```
> A <- matrix(c(0.8, -0.1, 0.8, 0.4), 2)
> b <- c(4, 2)
> f <- function(v) { A %*% v + b }
> f(c(15, 0))
     [,1]
[1,] 16.0
[2,]  0.5
```

Hvis man antager at $C = 15$ og $I = 0$ ved start, kan man lave økonomiske fremskrivninger for de næste 6 år på denne måde:

```
> sapply(c(0:6), function(i) { funpow(f, i, c(15, 0)) } )
     [,1] [,2] [,3]  [,4]   [,5]    [,6]     [,7]
[1,]   15 16.0 17.2 18.24 19.008 19.5136 19.81312
[2,]    0  0.5  0.6  0.52  0.384  0.2528  0.14976
```

Det egentlige emne for dette afsnit er løkker, men vi skal da nævne at `funpow(f,n,x)` også kan defineres rekursivt, uden en løkke:

```
funpow <- function(f, n, x) {
  if (n==0) x else funpow(f, n-1, f(x))
}
```

Denne version af `funpow` giver samme resultater som den løkkebaserede. Den svarer nøje til denne rekursive matematiske definition af $f^n(x)$ for $n \geq 0$:

$$f^n(x) = \begin{cases} x, & \text{hvis } n = 0, \\ f^{n-1}(f(x)) & \text{ellers.} \end{cases}$$

I stedet for `if-else` kan man bruge `ifelse` fra afsnit 19.3 i den rekursive `funpow`-funktion. Så virker funktionen også når `n` er en vektor, og man kan undgå brugen af `sapply`. Men det virker ikke når `f` returnerer en vektor eller matrix, så vi foretrækker kombinationen af `sapply` og `funpow`, der altid virker.

20 Associationslister og datasæt

Associationslister og datasæt i R bruges til at samle forskelligartede data i ét objekt. De opstår typisk når man indlæser data fra en tekstfil (afsnit 10.1) eller som resultat af funktioner der returnerer sammensatte data, såsom de indbyggede funktioner til nulpunktsfinding, numerisk optimering og numerisk integration i afsnit 7–9, eller lineær regression i afsnit 13. Men man kan også selv konstruere associationslister og datasæt.

20.1 Associationslister, funktionen `list`

En *associationsliste* er en værdi der kan indeholde flere navngivne komponenter. En associationsliste oprettes med funktionen `list`. I dette eksempel har associationslisten `xx` to komponenter `t` og `y`, som begge er vektorer:

```
> xx <- list(t=c(1.0, 1.5, 2.0, 2.5, 3.0),
+            y=c(1.4, 0.3, -1.5, -3.1, -4.8))
> xx
$t
[1] 1.0 1.5 2.0 2.5 3.0
$y
[1]  1.4  0.3 -1.5 -3.1 -4.8
```

Komponenterne i en associationsliste behøver ikke være vektorer, og behøver ikke have samme længde:

```
> blandet <- list(A=matrix(c(1, 2, 3, 4), 2), d=42.2)
```

Man kan få fat på de enkelte komponenter af en associationsliste med operatoren `$`, og man kan få oplyst navnene på en associationslistes komponenter med funktionen `names`:

```
> xx$t
[1] 1.0 1.5 2.0 2.5 3.0
> xx$y
[1]  1.4  0.3 -1.5 -3.1 -4.8
> names(xx)
[1] "t" "y"
```

Man kan også indeksere med den sædvanlige indeksoperator og navnene på tekstform som indeksværdier, eller med talværdier som indeks:

```
> xx["y"]
$y
[1]  1.4  0.3 -1.5 -3.1 -4.8
> xx[2]
$y
[1]  1.4  0.3 -1.5 -3.1 -4.8
```

Man kan ændre og tilføje elementer med tildelinger:

```
> xx$t <- 1:8
> xx$ny <- 10
> xx
$t
[1] 1 2 3 4 5 6 7 8
$y
[1]  1.4  0.3 -1.5 -3.1 -4.8
$ny
[1] 10
```

20.2 Datasæt, funktionen `data.frame`

Et *datasæt* i R, også kaldet en *data frame*, er en associationsliste hvis komponenter er vektorer af samme længde. Operatoren `$` samt alle funktioner (fx `names`) der virker på associationslister kan også bruges på datasæt. Et datasæt kan indlæses med funktionerne `read.table` og `read.csv` som vist i afsnit 10.1.1 og 10.1.2, eller oprettes direkte med funktionen `data.frame`. I modsætning til en associationsliste vises et datasæt på tabelform med en søjle for hver komponent. Det er muligt netop fordi alle komponenterne er vektorer af samme længde:

```
> d <- data.frame(t=c(1.0, 1.5, 2.0, 2.5, 3.0),
+                 y=c(1.4, 0.3, -1.5, -3.1, -4.8))
> d
    t    y
1 1.0  1.4
2 1.5  0.3
3 2.0 -1.5
4 2.5 -3.1
5 3.0 -4.8
```

I modsætning til for en almindelig associationsliste får man en fejlmelding hvis man forsøger at indsætte en ny søjle som ikke har den rigtige længde:

```
> d$ny <- 1:8
Error in "$<-.data.frame"('*tmp*', "ny", value = c(1, 2, 3, 4, 5, 6, :
        replacement has 8 rows, data has 5
```

20.3 Delmængder af datasæt

Man kan indeksere et datasæt med række og søjle, lige som en matrix (se afsnit 16.2). For eksempel:

```
> d[1,2]
[1] 1.4
> d[2:4,2]
[1]  0.3 -1.5 -3.1
> d[3,]
  t    y
3 2 -1.5
> d[1:3,]
    t    y
1 1.0  1.4
2 1.5  0.3
3 2.0 -1.5
```

Resultatet af en indeksering der giver mere end én søjle er et nyt datasæt. Ved at bruge vektorer af sandhedsværdier til at indeksere rækkerne med (afsnit 19.1) kunne man udtrække bestemte observationer fra et datasæt. Her udtrækker vi de observationer, altså rækker, hvor $t > y$:

```
> d1 <- d[ d$t > d$y ,]
> d1
    t    y
2 1.5  0.3
3 2.0 -1.5
4 2.5 -3.1
5 3.0 -4.8
```

Der er imidlertid en funktion `subset` som tillader det samme med en noget klarere notation:

```
> d2 <- subset(d, t>y)
> d2
    t    y
2 1.5  0.3
3 2.0 -1.5
4 2.5 -3.1
5 3.0 -4.8
```

Bemærk at man med `subset` kan skrive `t>y` i betingelsen i stedet for at skulle skrive `d$t>d$y`.

21 Plot af funktioner af 2 variable

R har mulighed for grafisk fremstilling af funktioner af 2 variable. Fælles for de muligheder, der vil blive præsenteret her, er at man først er nødt til at udregne z koordinaterne (funktionsværdierne) for et net af støttepunkter for det (x, y) område, funktionen skal tegnes for. For eksempel, hvis vi ønsker at tegne funktionen $z = f(x, y) = \cos(x)\sin(2y)$ for $x \in [0, 2\pi]$ og $y \in [-\pi, \pi]$ med 50×50 støttepunkter:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
```

Ovenstående kald af funktionen `outer` genererer en matrix af værdier med lige så mange rækker, som der er elementer i `x` og lige så mange søjler som der er elementer i `y`, hvor værdien på hver plads udregnes ved at anvende funktionen `f` på de tilsvarende pladser i `x` og `y`, altså `z[i,j]=f(x[i],y[j])`. Vi antager overalt i det følgende at `x`, `y` og `z` er beregnet som oven for.

Vi skal i afsnit 22 se et eksempel på hvordan man kan definere en funktion `overflade` der indpakker ovenstående datagenereringsopskrift og kaldet af plotfunktionen, så det hele bliver lidt mere behageligt at arbejde med.

21.1 3D overfladeplot af en funktion af to variable

For at lave et simpelt 3D plot af en funktion af 2 variable kan man bruge funktionen `persp`:

```
persp(x, y, z)
```

Her er `x`, `y` og `z` er defineret på forhånd som angivet i starten af afsnit 21 ovenfor. Overfladen tegnes som et net af firkanter udspændt af støttepunkterne, hvor hvert støttepunkt angiver et "knudepunkt" i nettet. Med 50×50 støttepunkter bliver der et net af 49×49 firkanter.

Udseendet af plot genereret med `persp` kan selvfølgelig ændres. Desværre har man i R ikke mulighed for interaktivt at trække i plottet med musen for at ændre betragtningsvinkel, så det kan kræve en del eksperimenteren med gentagne kald af `persp` indtil man finder den korrekte vinkel.Betragtningsvinkel angives med parametrene `phi`, der er højden i grader over (x, y) planen, og `theta`, der er drejningsvinklen omkring z-aksen. For eksempel:

```
persp(x, y, z, phi=35, theta=30)
```

Afstanden til betragtningspunktet fra centrum af "kassen" med den tegnede overflade (og dermed graden af perspektivisk fortegning) kan reguleres med parameteren `r`. Som standard er `r`=$\sqrt{3}$; større værdier giver mindre fortegning og mindre værdier en kraftigere fortegning, `r=10` giver udmærkede resultater (se figur 50):

```
persp(x, y, z, phi=35, theta=30, r=10)
```

Plottet skaleres som standard til at være i en kasse der er lige lang på alle sider. Hvis man ønsker at bevare det korrekte forhold mellem akserne skal man angive parameteren `scale=FALSE`:

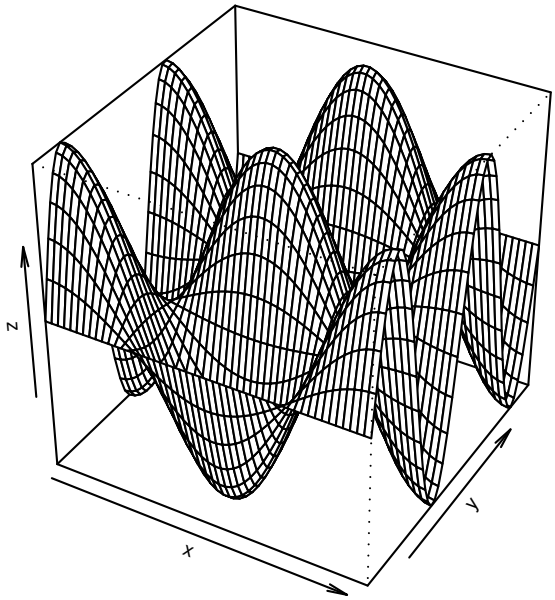
```
persp(x, y, z, phi=35, theta=30, scale=FALSE)
```

For at få enheder på akserne:

```
persp(x, y, z, phi=35, theta=30, scale=FALSE, ticktype="detailed")
```

Det er muligt at farve overfladen ved at angive parameteren `col`:

```
persp(x, y, z, phi=35, theta=30, col="red")
persp(x, y, z, phi=35, theta=30, col=rgb(1,0.5,0))
```



Figur 50: Plot af $f(x,y) = \cos(x)\sin(2y)$ med `persp(x, y, z, phi=35, theta=30, r=10)`.

Den sidste linje farver overfladen orange. Farven er angivet med dens koordinater i RGB-farverummet, hvor parametrene til funktionen `rgb` er intensiteten af rødt, grønt og blåt, hver angivet med et tal mellem 0 (ingen) og 1 (fuld intensitet). Man kan også angive værdien `NA` som farve, hvilket gør firkanterne "gennemsigtige". Man kan angive en sekvens af farver for at få en interessant flerfarvet effekt. Der er endda funktioner beregnet til at generere passende sekvenser af farver, for eksempel `rainbow`:

```
persp(x, y, z, phi=35, theta=30, col="NA")
persp(x, y, z, phi=35, theta=30, col=c("red","green","blue"))
persp(x, y, z, phi=35, theta=30, col=rainbow(50))
```

Parameteren til `rainbow` angiver hvor mange farver der skal genereres og er med vilje her valgt så antallet af farver er 1 større end antallet af firkanter langs x-aksen. Eksperimentér eventuelt selv med værdier fra 47 til 52 for at se effekten.

Funktionen `persp` kan også farvelægge overfladen som om den belyses fra en given retning. Parameteren `shade` regulerer groft sagt hvor meget der skal lægges vægt på retningsbestemt lys og hvor meget der skal lægges vægt på diffust "baggrundslys", hvor `shade=0` er standard og svarer til 100% diffust lys og `shade=1` svarer til 100% retningsbestemt lys (værdier mellem 0.5 og 0.75 svarer nogenlunde til almindeligt dagslys).

```
persp(x, y, z, phi=35, theta=30, shade=0.7)
persp(x, y, z, phi=35, theta=30, shade=0.7, ltheta=110)
```

De to parametre `ltheta` og `lphi` angiver retning til det retningsbestemte lys, efter samme system som betragtningsvinklen angives med `theta` og `phi`.

Den sidste parameter til `persp` der skal nævnes her er parameteren `border` der bestemmer farven af det "net" der tegnes mellem støttepunkterne. Hvis man har mange støttepunkter kan linjerne mellem dem næsten få overfladen til at forsvinde, så her kan man med fordel anvende `border=NA` for at slå linjerne helt fra (dette virker bedst sammen med retningsbestemt belysning).

```
persp(x, y, z, phi=35, theta=30, border="red")
persp(x, y, z, phi=35, theta=30, shade=0.7, border=NA)
```

21.1.1 Avanceret farvning af overfladeplot

Hvis man vil have farven af overfladen i et `persp` plot til at afhænge af z værdi (højde) er man nødt til at angive en farve for hver firkant mellem støttepunkterne. For vores 50×50 støttepunkter i eksemplet er der 49×49 firkanter mellem dem. Det vil sige at som `col` parameter kan vi for eksempel angive en 49×49 matrix af farver. Lad os nu prøve at lave en matrix af farver der gør at de laveste værdier farves sorte og de højeste hvide. For at lave en grå farve kan man bruge funktionen `gray`, hvor `gray(0)` giver sort og `gray(1)` giver hvid og værdier af parameteren mellem 0 og 1 giver grå nuancer. Vi skal så gøre følgende:

1. Find niveauet for de 49×49 firkanter. Som tilnærmelse tager vi her bare værdien i det ene hjørne af hver firkant ved at tage de første 49 rækker og 49 søjler af `z`.

2. Skalér værdierne således at de kommer til at ligge i intervallet 0 til 1.

3. Brug funktionen `gray` for at lave værdierne om til gråtonefarver.

```
zv <- z[1:49,1:49]
zv.skaleret <- (zv - min(zv)) / (max(zv) - min(zv))
zcol <- gray(zv.skaleret)
persp(x, y, z, phi=35, theta=30, col=zcol)
```

21.2 Plot med funktionen `image`

For at visualisere en funktion af to variable kan man i stedet for at tegne den i 3D med `persp` få tegnet den som et "kort" hvor funktionsværdierne (z-værdierne) afbildes over i en farveskala (for eksempel en gråtoneskala hvor laveste værdi svarer til sort og højeste til hvid). Lige som med `persp` kræver det en matrix af på forhånd udregnede funktionsværdier, z, samt vektorer x og y svarende til det anvendte net af støttepunkter, se forklaringen i starten af afsnit 21. Her er den indledende beregning af støttepunkter og funktionsværdier:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
```

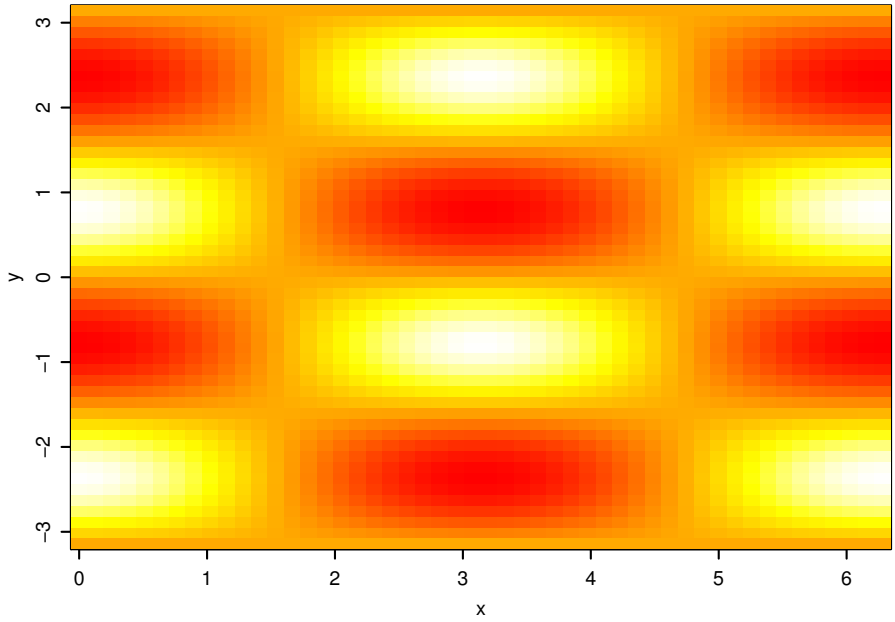
Funktionen, der tegner "kortet" hedder `image`:

```
image(x, y, z)
```

Som standard afbildes funktionsværdierne med en "varme"-farveskala hvor de lave værdier bliver røde og de høje bliver hvide, med nuancer af orange og gult indimellem. Standardskalaen består af kun 12 farver, så det første man kan gøre for at få et bedre plot er at sætte antallet af farver i skalaen op:

```
image(x, y, z, col=heat.colors(256))
```

Ovenstående anvender samme "varme"-farveskala som standardkaldet, men nu med 256 forskellige nuancer i skalaen. De 256 er valgt fordi farver i R (og i standard computer grafik i det hele taget) repræsenteres i et farverum af 256 nuancer af hver af komponenterne rød, grøn og blå, så med en lineær farveskala som den der produceres af `heat.colors` kan opdelingen ikke gøres finere.



Figur 51: Plot af $f(x, y) = \cos(x)\sin(2y)$ med `image(x, y, z, col=heat.colors(256))`.

Med parameteren `col` angives farveskalaen, der skal bruges, som en vektor af farvekoder. Sådanne vektorer er besværlige at angive manuelt, så man vil oftest anvende funktioner, der genererer vektoren af koder, som for eksempel `heat.colors`. Funktionen `rainbow` som blev brugt i nogle af eksemplerne med `persp` er et andet eksempel på en farveskalagenererende funktion, men den er begrænset nyttig til `image` da den som standard genererer en cyklisk liste af farver, hvilket bevirker at de højeste og laveste værdier får samme farve!

Man kan selv generere sin liste af farver, hvilket for eksempel er nødvendigt hvis man vil have en gråtoneskala hvor sort er lavt og hvidt er højt:

```
image(x, y, z, col=gray(seq(0,1,len=256)))
```

I ovenstående genereres farverne ved at anvende funktionen `gray` (beskrevet i afsnit 21.1.1 om `persp`) på en vektor af 256 tal gående fra 0 til 1.

Parametre til at regulere titel, koordinatakser, skalering osv. er lige som for `plot`; se afsnit 5.

21.3 Plot af niveaukurver

En tredie måde at visualisere en funktion af to variable på er med niveaukurver (som højdekurver på et kort). Funktionen, der gør dette, hedder `contour`. Lige som for `persp` og `skal` støttepunkter beregnes på forhånd, se forklaring i starten af afsnit 21:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
```

Herefter er vi klar til at lave selve niveaukurve-plottet:

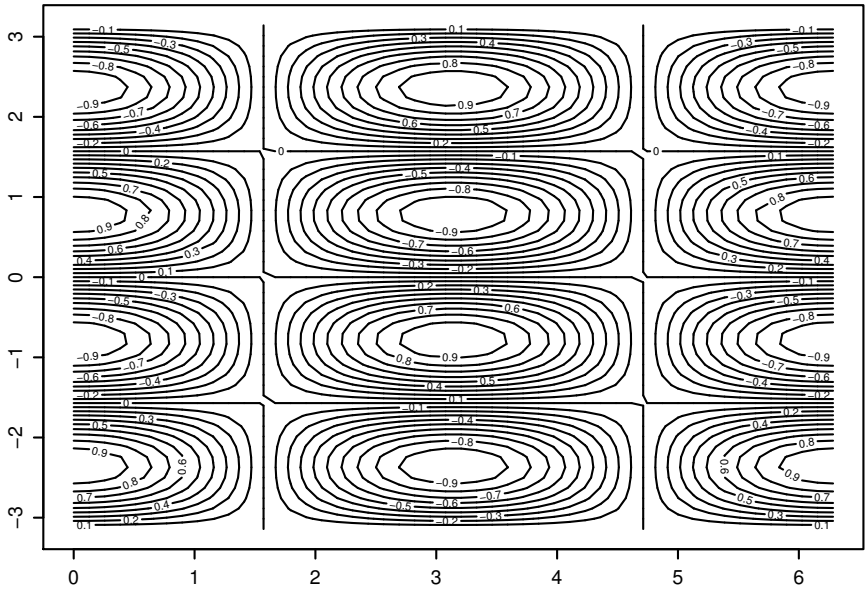
```
contour(x, y, z)
```

Som standard tegnes 10 niveaukurver, men man kan angive hvor mange man ønsker med parameteren `nlevels`:

```
contour(x, y, z, nlevels=15)
```

Man kan også eksplicit angive en vektor af de niveauer, der skal tegnes kurver for:

```
contour(x, y, z, levels=c(-0.95,-0.75,-0.5,0,0.5,0.75,0.95))
```



Figur 52: Niveaukurveplot af $f(x,y) = \cos(x)\sin(2y)$ med `contour(x, y, z, nlevels=15)`.

Det er muligt at regulere farve af kurverne med parameteren `col` og placering af niveauangivelserne på kurverne med parameteren `method`:

```
contour(x, y, z, method="edge", col="red")
```

Som `method` kan man angive `"simple"` (niveauangivelser i kanten af plottet, oven på konturerne), `"edge"` (niveauangivelser i kanten af plottet "indsat" i konturerne) eller `"flattest"` (standard, niveauangivelser indsat i den fladeste del af konturerne uden at de overlapper).

Hvis man slet ikke ønsker tekst på niveaukurverne kan man angive `drawlabels=FALSE`. Ud over det her nævnte kan man anvende mange af de samme parametre som til `plot` funktionen.

Man kan kombinere niveaukurveplot lavet af `contour` med et `image` plot som følger:

```
image(x, y, z)
contour(x, y, z, add=TRUE)
```

Dette virker bedst for et stort antal støttepunkter (for eksempel 500 gange 500) og et stort antal farver i `image` (for eksempel 256).

22 Avancerede funktionsparametre

Det er muligt i R at give funktioner fleksibilitet i den måde de kaldes på, fx således at parametre man sjældent ændrer ikke behøves at angives hver gang. Dermed kan man gøre funktionerne "brugervenlige" og bekvemme at bruge og gøre sit videre arbejde lettere. Dette er selvfølgelig primært relevant for funktioner som er så tilpas generelle at man regner med ofte at skulle bruge dem – også ud over den opgave man lige er i gang med.

Som gennemgående eksempel på avanceret definition af en funktion vil vi se på hvordan det kan gøres smidigere at lave et overfladeplot af en funktion af 2 variable. Undervejs i dette og de følgende afsnit vil vi se gradvist mere og mere avancerede definitioner af en funktion `overflade` til at tegne 3D overfladeplot.

Givet at vi har defineret en matematisk funktion `f(x,y)` skal vi som beskrevet i afsnit 21 igennem en række trin for at få grafen tegnet:

```
x <- seq(0, 2, len=50)          # 50 støttepunkter i intervallet [0,2]
y <- seq(-1, 1, len=50)         # 50 støttepunkter i intervallet [-1,1]
z <- outer(x, y, f)             # Funktionsværdier i støttepunkterne
persp(x, y, z)                  # Tegn grafen
```

Hvis vi ønsker at få tegnet grafen for andre intervaller, for et finere net af støttepunkter eller for en anden funktion end `f`, skal vi gentage flere af indtastningerne ovenfor med ændringer. Hvis det er noget vi skal gøre mere end nogle få gange kan det betale sig at definere en funktion der udfører alle ovenstående trin.

Vi definerer nu en funktion `overflade` der som argument tager en matematisk funktion af to variable samt intervaller for de to funktionsvariable og tegner en 3D graf for funktionen med `persp`:

```
overflade <- function(f, x.min, x.max, y.min, y.max) {
   x <- seq(x.min, x.max, len=50)  # 50 x-støttepunkter
   y <- seq(y.min, y.max, len=50)  # 50 y-støttepunkter
   z <- outer(x, y, f)             # Funktionsværdier i støttepunkterne
   persp(x, y, z)                  # Tegn grafen
}
```

Bemærk at de tre tildelinger til `x`, `y` og `z` *ikke* ændrer på eventuelle bindinger til disse variabelnavne uden for funktionen. De tre variable er lokale for funktionen.

Nu kan vi nemt få tegnet forskellige funktioner af to variable for forskellige intervaller:

```
> f <- function(x,y) { cos(x)*sin(2*y) }
> overflade(f, 0, 2, -1, 1)
> overflade(f, 0, 10, -5, 5)
> g <- function(x,y) { x^2+cos(x*y) }
> overflade(g, 0, 2, 0, 2*pi)
```

Ovenstående første forsøg på at definere funktionen `overflade` tillader os ikke at ændre antal støttepunkter. Vi udvider derfor funktionen lidt med en ny parameter `n`:

```
overflade <- function(f, x.min, x.max, y.min, y.max, n) {
   x <- seq(x.min, x.max, len=n)   # n x-støttepunkter
   y <- seq(y.min, y.max, len=n)   # n y-støttepunkter
   z <- outer(x, y, f)             # Funktionsværdier i støttepunkterne
   persp(x, y, z)                  # Tegn grafen
}
```

Med ovenstående nye udgave af `overflade` kan vi nu skrive:

```
> overflade(f, 0, 2, -1, 1, 50)
> overflade(f, 0, 10, -5, 5, 200)
> overflade(g, 0, 2, 0, 2*pi, 100)
```

22.1 Standardværdier for funktionsparametre

Der er efterhånden lidt mange parametre man skal huske at give til `overflade` for at få en graf (og vi vil indføre endnu flere senere). R har imidlertid mulighed for at angive *standardværdier* for parametrene i en funktionsdefinition. Hvis vi nu vælger, at intervallerne som standard skal være $[0, 1]$ og antal støttepunkter som standard skal være 50 kan vi definere funktionen som følger:

```
overflade <- function(f, x.min=0, x.max=1, y.min=0, y.max=1, n=50) {
   x <- seq(x.min, x.max, len=n)    # n x-støttepunkter
   y <- seq(y.min, y.max, len=n)    # n y-støttepunkter
   z <- outer(x, y, f)              # Funktionsværdier i støttepunkterne
   persp(x, y, z)                   # Tegn grafen
}
```

Bemærk, at det kun er i selve listen af formelle parametre i funktionshovedet vi har ændret; funktionskroppen er den samme som før. De parametre, der er standardværdier for, kan man så udelade når man kalder funktionen:

```
> overflade(f)
> overflade(f, 0, 10, -5, 5)
```

Det første funktionskald ovenfor tegner `f` for intervallerne $x \in [0, 1]$ og $y \in [0, 1]$ (givet af standardværdierne) mens det andet tegner `f` for $x \in [0, 10]$ og $y \in [-5, 5]$.

22.1.1 Navngivne parametre i funktionskald

Hvis nu vi ved kald af `overflade` ønsker at angive antal støttepunkter (som er sjette parameter) men ikke ønsker at angive intervalgrænserne (anden til femte parameter) kan vi skrive kaldet med angivelse af antal støttepunkter som en *navngiven* parameter:

```
> overflade(f, n=100)
```

Dette virker kun fordi alle de parametre vi "springer over" har standardværdier.

Tilsvarende kunne vi skrive

```
> overflade(f, x.max=10, y.max=10)
```

eller endda bytte om på rækkefølgen af parametrene ved at navngive dem alle:

```
> overflade(n=100, x.max=5, y.max=5, f=g)
```

Vi har tidligere i set talrige eksempler på brug af navngivne parametre når vi brugte indbyggede funktioner i R, for eksempel `plot` med dens myriader af forskellige parametre til at styre akser, tekster, stregtykkelser, farver osv. Mekanismen med at have standardværdier for de fleste parametre og angive navngive parametre når man bruger funktionerne kan gøre selv funktioner med dusinvis af parametre anvendelige.

Det er tilladt, når man angiver navngivne parametre i kald af en funktion, ikke at skrive parameternavne helt ud, så længe det er entydigt. Fx er "det fulde navn" for `seq`-funktionens `len` parameter (som styrer længden af den dannede vektor) i virkeligheden `length.out`, men da funktione ikke har andre parametre, der starter med "len" (eller overhovedet med "l"), kan den korte skrivemåde bruges i stedet. Så nedenstående skrivemåder er alle tilladte og giver alle det samme:

```
> seq(1, 7, length.out=3)
[1] 1 4 7
> seq(1, 7, len=3)
[1] 1 4 7
> seq(1, 7, l=3)
[1] 1 4 7
```

22.1.2 Mere komplekse standardværdier

Udtryk for standardværdier for de formelle parametre i en funktionsdefinition kan være vilkårligt komplekse og kan referere til andre af de formelle parametre. Hvis vi eksempelvis synes at man til `overflade` burde have mulighed for at angive intervalgrænserne som vektorer `c(`*min*`,`*max*`)` som alternativ til `x.min`, `x.max`, `y.min` og `y.max` og hvis vi gerne ville give mulighed for forskelligt antal støttepunkter på de to akser kunne det gøres således:

```
overflade <- function(f, x.min=min(x.interval), x.max=max(x.interval),
                      y.min=min(y.interval), y.max=max(y.interval),
                      n=50, x.n=n, y.n=n,
                      x.interval=c(0,1), y.interval=c(0,1) ) {
   x <- seq(x.min, x.max, len=x.n)
   y <- seq(y.min, y.max, len=y.n)
   z <- outer(x, y, f)
   persp(x, y, z)
}
```

Bemærk at de to nye parametre `x.n` og `y.n` som standardværdi har parameteren `n`. De nye intervalparametre `x.interval` og `y.interval` bruges nu til at give standardværdierne for de "gamle" parametre `x.min`, `x.max`, `y.min` og `y.max`. Grunden til at vi fx i standardværdien for `x.min` anvender `min(x.interval)` i stedet for `x.interval[1]` er at det tillader brugeren at angive `x.interval` og `y.interval` i lidt forskellige former (for eksempel størst før mindst eller som en vektor med mere end to elementer).

Eksempler på brug af den nye udgave af `overflade`:

```
> overflade(f, x.max=10, y.interval=c(-5,5) )
> overflade(g, 0, 2, 0, 2*pi, y.n=100)
```

Hvis vi ville være endnu mere fleksible kunne vi flytte de lokale variable `x` og `y` op som formelle parametre således at brugeren kunne angive vektorer med støttepunkterne eksplicit (hvis de ikke ønskes jævnt fordelt over intervallet):

```
overflade <- function(f, x.min=min(x.interval), x.max=max(x.interval),
                      y.min=min(y.interval), y.max=max(y.interval),
                      n=50, x.n=n, y.n=n,
                      x.interval=c(0,1), y.interval=c(0,1),
                      x=seq(x.min, x.max, len=x.n),
                      y=seq(y.min, y.max, len=y.n) ) {
   z <- outer(x, y, f)
   persp(x, y, z)
}
```

Det ville dog være en dårlig idé at flytte `z` op som formel parameter på en tilsvarende måde.

Et enkelt eksempel på brug af den nye definition:

```
> overflade(g, y.max=2*pi, x=c((0:10)/5,3:5))
```

I ovenstående er `x` koordinaten for støttepunkterne angivet til $0, \frac{1}{5}, \frac{2}{5}, \ldots, 2, 3, 4, 5$, altså med større tæthed for værdier af `x` under 2 end for værdier over 2.

22.2 Videregivelse af parametre med ". . ."

Ved at pakke `persp` ind i `overflade` har vi nu gjort det nemt at tegne en graf for en funktion af to variable ved at generere støttepunkterne som `persp` skal bruge. Til gengæld har vi mistet alle de muligheder som `persp` giver for at regulere betragtningsvinkel, farver, koordinatakser og så videre. Det er der

imidlertid råd for. Man kan til sidst i listen af formelle parametre i en funktionsdefinition angive (`...`), altså tre punktummer, hvilket betyder at funktionen accepterer et vilkårligt antal parametre ud over dem, der er specificeret. I funktionskroppen kan man så anvende (`...`) for at videregive parametrene til en anden funktion i et funktionskald. For at tillade at parametre til `persp` "passerer igennem" `overflade` kan vi skrive `overflade` som følger:

```
overflade <- function(f, x.min=min(x.interval), x.max=max(x.interval),
                      y.min=min(y.interval), y.max=max(y.interval),
                      n=50, x.n=n, y.n=n,
                      x.interval=c(0,1), y.interval=c(0,1),
                      x=seq(x.min, x.max, len=x.n),
                      y=seq(y.min, y.max, len=y.n),
                      ... ) {
   z <- outer(x, y, f)
   persp(x, y, z, ...)
}
```

Nu kan vi bruge funktionen som følger:

```
> overflade(g, x.max=4, y.max=2*pi, col="red")
> overflade(f, 0, 10, 0, 10, col="blue", theta=30, phi=35)
```

23 Opgaver til R

Når du løser disse opgaver kan du slå de forskellige R-funktioner op ved hjælp af det alfabetiske indeks bag i disse noter. Står der for eksempel "brug R-funktionen `uniroot` til ..." i en opgave og du ikke kan huske hvordan man bruger `uniroot` kan du slå op i indeks for at finde det afsnit hvor det er beskrevet.

Opgave 1 — Oprettelse af R-script

Denne opgave demonstrerer oprettelse af et script samt lagring af scriptet i en fil.

(1) Åbn et R Editor vindue ved at vælge File || New script (Mac: File || New Document), som beskrevet i afsnit 1.4. Indtast i R Editor vinduet følgende linje:

```
# Opgave 1
```

(Linjer der starter med # er kommentarer.)

(2) Vælg fra menuen File || Save og gem indholdet af dit R Editor vindue i en fil med filnavnet `Opgave-1.R` (bemærk "efternavn" `.R`).

(3) Luk editor vinduet med File || Close script (Mac: File || Close) eller vinduets lukke-knap. Prøv så at indlæse filen igen med File || Open script (Mac: File || Open Document) fra menuen.

Opgave 2 — Brug af R Editor

Denne opgave demonstrerer brugen af R Editor vinduet sammen med R Console. Du opfordres kraftigt til generelt at arbejde på denne måde når du løser opgaver.

(1) Åbn et R Editor vindue ved at vælge File || New script (Mac: File || New Document), som beskrevet i afsnit 1.4. Indtast i R Editor vinduet følgende tre linjer:

```
f <- function(x) { x^2 + 3*x - 7 }
f(1)
f(7)
```

Udfør derefter de tre linjer i R Console ved at markere dem og enten taste Ctrl+R (Mac: ⌘+⏎) for at udføre dem direkte eller kopiere dem med Ctrl+C og sætte dem ind i R Console med Ctrl+V (Mac: ⌘+C og ⌘+V).

Dette skulle gerne definere funktionen $f(x) = x^2 + 3x - 7$ og give værdierne af $f(1)$ og $f(7)$.

(2) Ret nu første linje i R Editor så funktionen i stedet er $f(x) = 2x^2 + 3x - 7$ og udfør de tre linjer igen.

(3) Kopiér de tre linjer ved at markere dem og taste Ctrl+C og derefter Ctrl+V (Mac: ⌘+C og ⌘+V). Du skulle nu have i alt seks linjer i R Editor vinduet.

Ret de tre nye linjer så de definerer funktionen $g(x) = 4x^2 - 2x + 3$ og udregner $g(1)$ og $g(7)$. Udfør disse tre linjer i R Console.

(4) Kopiér igen de første tre linjer og ret de nye linjer så de definerer $h(x) = x^3 + 2x^2 + 3x - 7$ og udregner $h(1)$ og $h(7)$. Udfør derefter de tre nye linjer i R Console

Der burde nu være i alt 9 linjer i R Editor vinduet.

(5) Tilføj som første linje i R Editor vinduet følgende tekst:

```
# Opgave 2
```

(6) Gem indholdet af dit R Editor vindue i en fil med filnavnet `Opgave-2.R` (se afsnit 1.4 eller opgave 1).

Opgave 3 — Definér og anvend simpel funktion

Åbn et R Editor vindue ved at vælge File || New script (Mac: File || New Document), som beskrevet i afsnit 1.4 og opgave 2.

Definér funktionen $f(x) = 3e^{-2(x-1)^2}$ i R. Brug din funktion til at udregne $f(-1)$, $f(0)$ og $f(1)$.

Gem dit script (indholdet af dit R Editor vindue) i en fil med filnavnet `Opgave-3.R`.

Opgave 4 — Definér, anvend og plot simpel funktion

Åbn et R Editor vindue ved at vælge File || New script (Mac: File || New Document), som beskrevet i afsnit 1.4 og opgave 2.

(1) Definér funktionen $f(x) = 2\cos(\frac{x}{2}) + x$ i R. Brug din funktion til at udregne $f(-1)$, $f(0)$ og $f(1)$.

(2) Lav med R-funktionen `plot` en graf for funktionen f for intervallet $x \in [-10, 10]$.

(3) Lav med `plot` en ny graf for funktionen f, nu for intervallet $x \in [-20, 20]$. Giv grafen titlen "Funktionen f" (med `main=`) og aksetekserne "x" og "y" på første- hhv. andenaksen (med `xlab=` hhv. `ylab=`). Selve kurven skal være blå (brug `col=`).

Gem dit script (indholdet af dit R Editor vindue) i en fil med filnavnet `Opgave-4.R`.

Opgave 5 — Funktion med parameter

Definér funktionen $f(x) = x^a$ hvor a er en parameter. Hvad sker der når du skriver **`f(2)`** i R Console?

Sæt nu $a = 3$ (med tildelingen `a <- 3`) og skriv igen **`f(2)`**. Udregn også værdien $f(-2)$.

Sæt nu $a = 0.5$ og udregn igen $f(2)$ og $f(-2)$. Forklar.

Opgave 6 — Find nulpunkt for funktion

(1) Definér funktionen $f(x) = x^3 + 4x^2 - 17$ i R. Brug din funktion til at udregne $f(1)$, $f(-\frac{8}{3})$ og $f(-4)$.

(2) Lav med R-funktionen `plot` grafer for funktionen f for intervallerne $x \in [-5, 5]$ og $x \in [-3, 2]$.

(3) Find et nulpunkt for funktionen f ved hjælp af R-funktionen `uniroot`. Brug graferne fra før til at vælge et passende interval at søge i.

Opgave 7 — Løs tredjegradsligning

Find de tre løsninger til tredjegradsligningen $-2x^3 + 6x^2 - 5 = 0$ ved hjælp af R.

Fremgangsmåde: Definér venstresiden som en funktion af x og lav en graf hvoraf du kan se cirka hvor nulpunkterne er. Brug derefter `uniroot` tre gange med forskellige intervaller for at finde de tre nulpunkter.

Opgave 8 — Løs ligning

Det oplyses at ligningen $x^2 - 4\sin(3x) - 1 = 0$ har netop fire løsninger. Find de fire løsninger ved hjælp af R.

Fremgangsmåde: Definér venstresiden som en funktion af x og lav en graf hvoraf du kan se cirka hvor nulpunkterne er. Brug derefter `uniroot` fire gange for at finde de fire nulpunkter.

Opgave 9 — Plot flere funktioner sammen

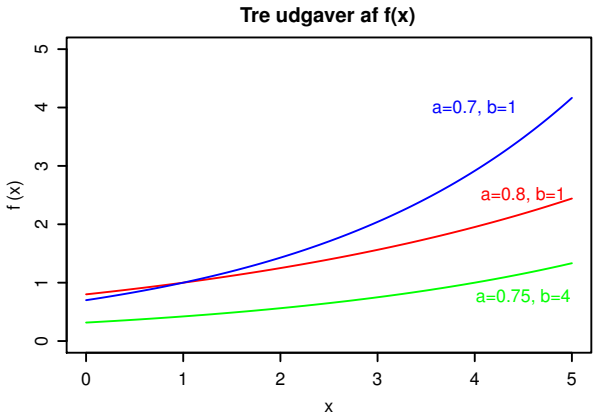
Definér funktionerne $g(x) = 0.6 \cdot e^{-x^2}$ og $h(x) = 1 - g(x + 2)$. Tegn de to funktioner for intervallet $x \in [-5, 5]$ i samme graf. Sørg for at akseteksten på andenaksen bliver "y". Lav de to kurver i forskellig farve.

Fremgangsmåde: Brug `plot(...)` for at tegne første kurve og `plot(...,add=TRUE)` for at tegne anden kurve. Sørg i det første `plot`-kald for at intervallet på andenaksen er stort nok til at begge kurver kan vises (brug `ylim=`).

Opgave 10 — Flere kurver i samme graf

Definér funktionen $f(x) = a^{b-x}$, hvor $a > 0$ og b er parametre. Du skal nu lave en graf for funktionen for intervallet $x \in [0, 5]$ for tre forskellige kombinationer af a og b. Du skal bruge `plot` tre gange, med `add=TRUE` i de to sidste kald. Sørg for at sætte andenaksens interval til $y \in [0, 5]$ i det første kald af `plot` for at ingen af kurverne skal blive beskåret. Definér først $a = 0.8$ og $b = 1$ og tegn en rød kurve. Sæt derefter $a = 0.7$ og $b = 1$ og tegn en blå kurve. Sæt endelig $a = 0.75$ og $b = 4$ og tegn en grøn kurve.

Du kan eventuelt sætte forklarende tekst ind ved siden af hver kurve (for eksempel "a=0.8, b=1" ved den første) med R-funktionen `text`. Sæt eventuelt også farven på den forklarende tekst til at være den samme som farven på kurven. Grafen kan komme til at ligne nedenstående:



Opgave 11 — Numerisk integration

Bestem ved hjælp af R-funktionen `integrate` værdien af det bestemte integral

$$\int_1^5 e^x \, dx.$$

Opgave 12 — Graddage og ukrudtsbekæmpelse

Denne opgave bygger på anvendelseseksempel A.16 "Graddage og ukrudtsbekæmpelse" fra *Noter om matematik*.

(1) Definer funktionen `T` (døgnets middeltemperatur som funktion af antal dage efter 15. april) fra anvendelseseksemplet, dvs.

$$T(t) = 8 + 8\sin\left(\frac{2\pi}{365}t\right).$$

(I R skriver man π som `pi`.)

Check at din definition er fornuftig ved at lave en graf for `T` for intervallet $t \in [0, 365]$.

(2) Funktionen Graddage defineres i *Noter om matematik* som:

$$\text{Graddage}(t_2) = \int_0^{t_2} T(t)dt.$$

I modsætning til i matematiknoterne skal du ikke selv integrere T men derimod lade R om at lave numerisk integration. Skriv nu i R funktionen `Graddage(t2)` svarende til ovenstående. Du skal bruge R-funktionen `integrate` og huske at tage `$value` for at få et tal vi kan regne videre på, altså `integrate(...)$value`. Check din funktion ved at tage værdien til $t_2 = 10$:

```
> Graddage(10)
[1] 86.8687
```

Note: Desværre kan man ikke lave en graf for `Graddage` direkte med `plot` som man kan for `T`. Det fører for vidt her at forklare hvorfor.

(3) Bestem den værdi af t_2 for hvilken `Graddage`(t_2) når værdien 350, hvilket i R gøres nemmest ved at finde nulpunkt for funktionen $f(t_2) = \text{Graddage}(t_2) - 350$. Brug R funktionen `uniroot` til at finde en løsning i intervallet $t_2 \in [0, 100]$.

Opgave 13 — Koncentrationsforøgelse i biokemisk proces

Denne opgave bygger på anvendelseseksempel A.20 "Koncentrationsforøgelse i biokemisk proces" fra *Noter om matematik*.

(1) Definer funktionen `v` (hastigheden hvormed et stof dannes i en given proces som funktion af tiden t) som

$$v(t) = (1.1 + 3.5t)e^{-1.5t}$$

og lav en graf for `v` for t mellem 0 og 3 svarende til figur A.64 i matematiknoterne.

(2) Bestem nu den totale koncentrationsforøgelse, dvs.

$$\int_0^\infty v(t)dt,$$

i R ved numerisk integration med `integrate`.

Opgave 14 — Indlæsning, plot og transformation af data fra kemisk reaktion

(1) Gå ind under "eksempelfiler" på hjemmesiden `http://matdat.life.ku.dk/R-noter`. Højreklik på filen `data/reaktion.txt` og vælg "gem som" og gem filen på din computer.

(2) Start med at åbne filen i fx. Notesblok (højreklik og "Åbn"). Undersøg om der er en overskrift, om der er brugt decimalkomma eller decimalpunktum, samt hvordan kolonnerne er adskilt (skilletegn).

Læs datasættet ind med `data <- read.table(...)` og husk at angive de nødvendige parametre så overskrift, tal og skilletegn læses korrekt.

Brug funktionen `summary` for at tjekke at data er indlæst korrekt. Det skal se således ud:

```
> summary(data)
      Tid              S1              S2              S3
 Min.   :  2.0   Min.   :0.050   Min.   :0.1000   Min.   :0.050
 1st Qu.: 26.5   1st Qu.:0.500   1st Qu.:0.5625   1st Qu.:4.088
 Median : 51.0   Median :1.275   Median :1.3000   Median :7.250
 Mean   : 51.0   Mean   :2.352   Mean   :1.3770   Mean   :6.264
 3rd Qu.: 75.5   3rd Qu.:3.413   3rd Qu.:2.1875   3rd Qu.:8.950
 Max.   :100.0   Max.   :9.400   Max.   :2.8500   Max.   :9.700
```

(3) Lav med `plot` et XY-plot af kolonnen `S2` som funktion af kolonnen `Tid`. Datapunkterne skal ikke forbindes med linjer.

Vink: For at få kolonnen `S2` fra datasættet `data` skal du skrive `data$S2`.

(4) Lav et XY-plot af kvadratroden (`sqrt`) af kolonnen `S1` som funktion af kolonnen `Tid`. Datapunkterne skal ikke forbindes med linjer.

Definér funktionen $f(t) = 2.66 - 0.025t$. Tilføj med `plot(...,add=TRUE)` grafen for f.

(5) Lav et XY-plot af den naturlige logaritme (`log`) til kolonnen `S1` som funktion af kolonnen `Tid`. Datapunkterne skal ikke forbindes med linjer.

Definér funktionen $g(t) = 2.3 - 0.04t$. Tilføj med `plot(...,add=TRUE)` grafen for g.

Opgave 15 — Indlæsning og plot af data fra væksthormonforsøg

Filen `data/Grise2Fast.txt` fra eksempelfil-samlingen på hjemmesiden indeholder data fra et forsøg med væksthormon til slagtesvin. Datasættet har tre variable `Tid`, `Kontrol` og `Vaekst`, og 20 observationer. Hver observation angiver et antal minutter efter slagtning sammen med målt pH i kødet fra grise der er opdrættet normalt (kontrolgruppen) og fra grise der har fået væksthormon (se også afsnit 10).

(1) Hent fra hjemmesiden `http://matdat.life.ku.dk/R-noter` filen `data/Grise2Fast.txt` og gem filen på din computer.

(2) Undersøg filens indhold for overskrifter, skilletegn m.v. og læs så datasættet fra filen ind i R med `read.table`. Kald datasættet `data`. Check med `summary` at det er korrekt indlæst.

(3) Lav med `plot` og `points` et XY-plot med de to kolonner `Kontrol` og `Vaekst` som funktion af kolonnen `Tid`. Datapunkterne fra `Kontrol` skal være røde og datapunkterne fra `Vaekst` blå. Husk (med `ylim=` i `plot`) at sætte intervallet på andenaksen så ingen punkter falder udenfor (brug oplysningerne fra `summary` til at vælge et interval). Aksetekserne skal være "Minutter" på førsteaksen og "pH" på andenaksen (teksterne angives med `xlab=` og `ylab=` i `plot`).

(4) Tilføj med `legend` en signaturforklaring til grafen fra delopgave (3). Den skal have en rød cirkel med teksten "Kontrolgruppe" og en blå cirkel med teksten "Væksthormon".
Vink: Cirklen som R tegner som standard er plot-symbol (plotting character) nummer 1, så for at få cirkler ud for hver tekst i `legend` skal man huske `pch=1` (jævnfør figur 30 på side 48).

Opgave 16 — Udklækningstid for flueæg

Denne opgave bygger på anvendelseseksempel A.1 "Udklækningstid for flueæg" fra *Noter om matematik*.

(1) Datasættet med måling af udklækningstid U som funktion af luftfugtighed L findes i tekstfilen `data/flueaeg.txt` under eksempelfiler på hjemmesiden `http://matdat.life.ku.dk/R-noter`. Hent filen og gem den på din computer. Indlæs datasættet med:

```
flueaeg <- read.table("flueaeg.txt", header=TRUE)
```

(2) Lav et XY-plot af `flueaeg$U` som funktion af `flueaeg$L`.

(3) Foretag med R-funktionen `lm` lineær regression for at bestemme den rette linje der bedst beskriver sammenhængen mellem L og U (kald resultatet for `lreg`). Angiv ud fra resultatet konstanterne a og b i modellen $U = aL + b$.

(4) Brug endelig R-funktionen `abline` til at tilføje linjen $U = aL + b$ til plottet fra delopgave (2).

Opgave 17 — Andemad

Denne opgave bygger på anvendelseseksempel A.4 "Andemad" fra *Noter om matematik*.

(1) Datasættet med antal blade N som funktion af antal dage siden forsøgets start t findes i filen `data/andemad.txt` under eksempelfiler på hjemmesiden `http://matdat.life.ku.dk/R-noter`. Hent filen og gem den på din computer. Indlæs datasættet med:

```
andemad <- read.table("andemad.txt", header=TRUE)
```

(2) Lav et XY-plot af logaritmen til `andemad$N` som funktion af `andemad$t`.

(3) Transformér kolonnen `N` ved at tage den naturlige logaritme. Kald de transformerede værdier for `logN`. Foretag med R-funktionen `lm` lineær regression for at bestemme den rette linje der bedst beskriver sammenhængen mellem $\ln N$ og t (kald resultatet for `lreg`). Angiv ud fra resultatet konstanterne b og r i modellen $\ln N = \ln b + rt$.

(4) Brug R-funktionen `abline` til at tilføje linjen $\ln N = \ln b + rt$ til plottet fra delopgave (2).

(5) Definér funktionen $f(t) = be^{rt}$ med de fundne værdier for b og r fra delopgave (3).

Lav nu et XY-plot af `andemad$N` som funktion af `andemad$t` og brug `plot(...,add=TRUE)` til at tilføje grafen for $f(t)$ for intervallet $t \in [0, 13]$.

Opgave 18 — Stofskifte og kropsvægt hos pattedyr

Denne opgave bygger på del (d) af anvendelseseksempel A.8 "Stofskifte og kropsvægt hos pattedyr" fra *Noter om matematik*.

(1) Datasættet med stofskifte S som funktion af kropsvægt K findes i filen `data/stofskifte.txt` under eksempelfiler på hjemmesiden `http://matdat.life.ku.dk/R-noter`. Hent filen og gem den på din computer. Indlæs datasættet med:

```
stofskifte <- read.table("stofskifte.txt", header=TRUE)
```

(2) Transformér kolonnerne `K` og `S` ved at tage den naturlige logaritme. Kald de transformerede data for `logK` og `logS`. Foretag med R-funktionen `lm` lineær regression for at bestemme den rette linje der bedst beskriver sammenhængen mellem $\ln S$ og $\ln K$ (kald resultatet for `lreg`). Angiv ud fra resultatet konstanterne a og b i modellen $\ln S = \ln b + a \ln K$.

(3) Lav et XY-plot af `logS` som funktion af `logK`. Tilføj derefter ved hjælp af `abline` den rette linje $\ln S = \ln b + a \ln K$ fundet i delopgave (2).

(4) Definér funktionen $f(K) = bK^a$ med de fundne værdier for a og b fra delopgave (2).

Lav nu et XY-plot af `stofskifte$S` som funktion af `stofskifte$K` og brug `plot(...,add=TRUE)` til at tilføje grafen for $f(K)$ for intervallet $K \in [0, 50]$.

Opgave 19 — Arbejde med data (powerdata.csv)

Filen `powerdata.csv` indeholder data fra et eksperiment hvor man har målt den tid det tager en computer at "lægge" et puslespil. Man har prøvet dette for forskellige puslespil med forskellig antal brikker N. Der er målt tidsforbrug i sekunder for to forskellige alternative løsningsstrategier, A og B.

(1) Hent fra hjemmesiden `http://matdat.life.ku.dk/R-noter` filen `data/powerdata.csv` og gem filen på din computer.

(2) Læs datasættet fra filen ind i R. Kald det `data`. Check med `summary` at data er korrekt indlæst.

(3) Lav med `plot` og `points` et XY-plot med de to kolonner `TidA` og `TidB` som funktion af kolonnen `N`. Datapunkterne fra `TidA` skal være røde og datapunkterne fra `TidB` blå. Husk at sætte intervallet på andenaksen så ingen punkter falder udenfor (brug oplysningerne fra `summary` til at vælge et interval). Akseteksterne skal være "N" på førsteaksen og "Tid" på andenaksen.

(4) Foretag med R-funktionen `lm` lineær regression for at bestemme den rette linje der bedst beskriver sammenhængen mellem `TidA` og `N`. Lav herefter et XY-plot med `TidA` som funktion af `N` og tilføj den fundne regressionslinje ved hjælp af R-funktionen `abline`.

(5) Transformér kolonnerne `N`, `TidA` og `TidB` ved at tage den naturlige logaritme. Kald de transformerede data for `logN`, `logA` og `logB`.

Lav herefter med `plot` og `points` et XY-plot for de transformerede data med røde punkter for `logA` som funktion af `logN` og blå punkter for `logB` som funktion af `logN` (ligesom i delopgave (3)). Akseteksterne skal være "ln(N)" på førsteaksen og "ln(Tid)" på andenaksen.

(6) Foretag med R-funktionen `lm` lineær regression for at bestemme den rette linje der bedst beskriver sammenhængen mellem `logA` og `logN`. Gør det samme for at finde sammenhængen mellem `logB` og `logN`. Tilføj med `abline` regressionslinjerne til plottet fra delopgave (5) i hhv. rød og blå.

(7) Tilføj nu med `legend` en signaturforklaring med følgende fire tekster: "Data A" (røde punkter), "Data B" (blå punkter), "Regression A" (rød linje) og "Regression B" (blå linje).

Opgave 20 — Oprettelse af talrækker/vektorer

(1) Konstruér ved hjælp af R-funktionen `seq` følgende talrækker/vektorer:

- Tallene 1, 2, . . . , 10.
- Tallene 10, 9, . . . , 1.
- Tallene 0.25, 0.5, 0.75, . . . , 5.0.
- De ulige tal fra 1 til 29.

(2) Konstruér udelukkende ved hjælp af kolon-operatoren (fx `1:5`) følgende talrækker/vektorer:

- Tallene 1, 2, . . . , 10.
- Tallene -1, -2, . . . , -10.
- Tallene 6.4, 7.4, . . . , 19.4.

Opgave 21 — Flere talrækker/vektorer

(1) Konstruér ved hjælp af R-funktionen `seq` følgende talrækker/vektorer:

- Tallene 0.1, 0.2, 0.3, . . . , 1.0.
- Tallene 5, 10, 15, . . . , 100.
- Tallene 1.5, 2.5, . . . , 9.5.
- Tallene 100, 90, 80, . . . , 0.

(2) Konstruér udelukkende ved hjælp af kolon-operatoren samt eventuelt regneoperationer, for eksempel `(1:5)*2` (som giver 2, 4, 6, 8, 10), de samme fire talrækker som ovenfor.

Opgave 22 — Endnu flere talrækker/vektorer

Udfordring: Konstruér ved udelukkende hjælp af kolon-operatoren samt eventuelt regneoperationer følgende talrækker/vektorer:

- Tallene 100, 81, 64, 49, 36, 25, 16, 9, 4, 1.
- Tallene 0, 0, 2, 2, 4, 4, 6, 6, 8, 8.
- Tallene 2, 4, 6, 8, 10, 7, 9, 11, 13, 15.

- Tallene -1, 0, 3, -4, 0, 6, -7, 0, 9.
- Tallene 1, 2, 9, 4, 25, 6, 49, 8, 81, 10.

Vink: Undtagen for den første talrække skal der gøres kreativ brug af R's genbrugsregel for regning med vektorer af forskellig længde.

Opgave 23 — Oprettelse af og regning med 2×2 matricer

(1) Opret følgende to 2×2 matricer:

$$A = \begin{pmatrix} 1 & 2 \\ 5 & 3 \end{pmatrix}, \quad B = \begin{pmatrix} 4 & 6 \\ 7 & 8 \end{pmatrix}.$$

(2) Udregn matrixprodukterne AB og BA. Er $AB = BA$?

Opgave 24 — Matrix invers og 2×2 ligningssystem

(1) Find ved at bruge `solve` den inverse til nedenstående matrix:

$$\begin{pmatrix} 1 & 7 \\ 9 & 13 \end{pmatrix}$$

(2) Løs følgende ligningssystem i R ved at omskrive til matrixform og bruge `solve`:

$$\begin{cases} x + 7y = 3 \\ 9x + 13y = -2 \end{cases}$$

Opgave 25 — Opret større matrix; indeksering

(1) Opret en matrix A som følger:

$$A = \begin{pmatrix} 1 & 5 & 9 & 13 & 17 \\ 2 & 6 & 10 & 14 & 18 \\ 3 & 7 & 11 & 15 & 19 \\ 4 & 8 & 12 & 16 & 20 \end{pmatrix}.$$

Vink: Bemærk at matricens elementer er talrækken $1, \ldots, 20$.

(2) Udtag (ved hjælp af indeksering) fra A følgende dele:

- Elementet i 1. række, 3. søjle.
- Elementet i 4. række, 2. søjle.
- Hele 3. række.
- Hele 4. søjle.

- Delmatricen bestående af de første tre søjler.
- Delmatricen bestående af de to sidste rækker.
- 2×2 delmatricen bestående af de to sidste søjler og rækker ("nederste højre hjørne").
- Delmatricen bestående af alle søjler på nær den tredje.

Opgave 26 — Opbygning af matrix med `cbind`

(1) Definér følgende matrix A og vektor v_0 (skriv `v0` for v_0 i R):

$$A = \begin{pmatrix} 0.20 & 0.80 \\ 0.90 & 0.15 \end{pmatrix}, \quad v_0 = \begin{pmatrix} 100 \\ 0 \end{pmatrix}.$$

Udregn herefter successivt følgende vektorer:

$$v_1 = Av_0, \;\; v_2 = Av_1, \;\; v_3 = Av_2, \;\; v_4 = Av_3, \;\; v_5 = Av_4.$$

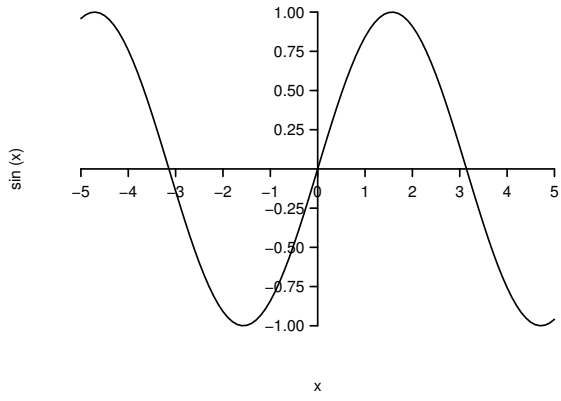
Konstruér endelig matricen V hvis søjler udgøres af vektorerne v_0 til v_5, altså hvis første søjle er v_0, anden søjle v_1 og så videre. Brug R-funktionen `cbind`.

(2) Lav med `plot` et XY-plot hvis punkter har første række af V som y-koordinater og talrækken fra 0 til 5 som x-koordinater. Punkterne skal indtegnes forbundet med røde linjer (brug `type="o",col="red"`).

Tilføj med `points` den anden række af V, indtegnet på samme måde men i blåt.

Opgave 27 — Lav plot med bestemte akser

Lav en graf for $\sin x$ for $x \in [-5, 5]$ svarende fuldstændig til denne (bemærk akserne, herunder at på andenaksen er etiketten 0.00 udeladt og etiketterne står vandret):



Vink: Du skal først plotte uden akser og derefter bruge `axis`, se afsnit 5.10. Du skal bruge parameteren `at` med passende talrækker til at bestemme hvilke aksemærker, der skal sættes. Du får også brug for parameteren `las`.

Opgave 28 — Fremskrivning i affin model med for-løkke

(1) Definér følgende matrix $\mathbf{M}$, vektor $\mathbf{q}$ og vektor $\mathbf{v}_0$ (skriv `v0` for $\mathbf{v}_0$ i R):

$$\mathbf{M} = \begin{pmatrix} 0.2 & 0.8 \\ 0.9 & 0.0 \end{pmatrix}, \quad \mathbf{q} = \begin{pmatrix} 10 \\ 0 \end{pmatrix}, \quad \mathbf{v}_0 = \begin{pmatrix} 100 \\ 0 \end{pmatrix}.$$

Udregn nu ved hjælp af en `for`-løkke vektorerne $\mathbf{v}_1$ til $\mathbf{v}_{20}$ ud fra den affine model

$$\mathbf{v}_{i+1} = \mathbf{M}\mathbf{v}_i + \mathbf{q}.$$

Vektorerne skal undervejs "opsamles" i en matrix `V` således at første søjle i `V` udgøres af $\mathbf{v}_0$, anden søjle af $\mathbf{v}_1$ og så videre.

Vink: Se på eksemplet i afsnit 16.9.

(2) Lav med `plot` et XY-plot hvis punkter har første række af `V` som y-koordinater og talrækken fra 0 til 20 som x-koordinater. Punkterne skal indtegnes forbundet med røde linjer (brug `type="o",col="red"`).

Tilføj med `points` den anden række af `V`, indtegnet på samme måde men i blåt.

Opgave 29 — Fårebestand

Denne opgave bygger på anvendelseseksempel B.4 fra *Noter om matematik*.

En fårebestand med får i to klasser modelleres v.h.a. følgende affine model, jvf. anvendelseseksempel B.4 (b):

$$\mathbf{v}_{t+1} = \mathbf{M}\mathbf{v}_t + \mathbf{q},$$

hvor

$$\mathbf{v}_t = \begin{pmatrix} x_t \\ y_t \end{pmatrix}, \quad \mathbf{M} = \begin{pmatrix} 0.70 & 0.20 \\ 0.15 & 0.45 \end{pmatrix}, \quad \mathbf{q} = \begin{pmatrix} 0 \\ 900 \end{pmatrix}.$$

I året 1992 er der 500 hhv. 1400 får i de to klasser, d.v.s. $\mathbf{v}_{1992} = (500, 1400)$.

Vi ønsker at fremskive fårebestanden til året 2023 ved hjælp af modellen.

(1) Definér i R matricen `M` og vektorerne `q` og `v1992` svarende til modellen ovenfor.

Udregn nu ved hjælp af en `for`-løkke vektorerne $\mathbf{v}_{1993}$ til $\mathbf{v}_{2023}$ og opsaml undervejs resultatet i en 2×32 matrix `V`, hvor første søjle udgøres af $\mathbf{v}_{1992}$, anden søjle af $\mathbf{v}_{1993}$ osv.

Vink: Se på eksemplet i afsnit 16.9.

(2) Lav med `plot` et XY-plot som viser x_t som funktion af årstal fra 1992 til 2023. Punkterne skal indtegnes med åbne cirkler (plotsymbol 1) og forbundet med fuldt optrukne linjer. Sørg for passende aksetitler ("år" og "bestand"). Vink: x_t værdierne udgør første række af `V`.

Tilføj med `points` y_t værdierne fra 1992 til 2023. Punkterne skal indtegnes med trekanter (plotsymbol 2) og forbindes med stiplede linjer (`lty="dashed"`).

Tilføj titlen "Udvikling af fårebestand".

Tilføj en signaturforklaring med `legend` med de to signaturtekster "Klasse I" henholdsvis "Klasse II" for x_t henholdsvis y_t værdierne.

Opgave 30 — Vektorkonstruktion med `for`-løkker

I denne opgave skal en række specielle vektorer, alle med 10 elementer, konstrueres ved hjælp af `for`-løkker. Fremgangsmåden er hver gang den samme:

1. Opret en vektor med 10 elementer med `rep(`*n*`,10)`, hvor *n* er tallet der skal stå på de(n) første plads(er) i vektoren. For eksempel oprettes den første vektor `u` nedenfor med `u <- rep(3,10)`.
2. Kør en (eller om nødvendigt flere) `for`-løkke(r) der tildeler de ønskede værdier til pladserne 2 til 10 i vektoren. For eksempel vil den for den første vektor `u` nedenfor have formen `for (i in 2:10) { u[i] <-` *udtryk* `}`.
3. Tjek til sidst at vektoren har fået det ønskede indhold (ved at skrive fx `u` på en linje for sig selv).

De vektorer, der ønskes konstrueret, er:

- Vektoren `u`, hvorom det gælder at `u[1]=3` og `u[i]=u[i-1]+i` for `i`=2, 3, ..., 10.
  Resultatet bliver talrækken 3, 5, 8, 12, 17, 23, 30, 38, 47, 57.
- Vektoren `v`, hvorom det gælder at `v[1]=1` og `v[i]=i-2*v[i-1]` for `i`=2, 3, ..., 10.
  Resultatet bliver talrækken 1, 0, 3, -2, 9, -12, 31, -54, 117, -224.
- Vektoren `w`, hvorom det gælder at `w[1]=10`, `w[2]=10` og `w[i]=w[i-2]-w[i-1]` for `i`=3, 4, ..., 10.
  Resultatet bliver talrækken 10, 10, 0, 10, -10, 20, -30, 50, -80, 130.
- Vektoren `z`, hvorom det gælder at `z[1]=1`, `z[i]=z[i-2]+i^2` for `i`=3, 5, ..., 9 og `z[j]=z[j-1]+3` for `j`=2, 4, ..., 10.
  Resultatet bliver talrækken 1, 4, 10, 13, 35, 38, 84, 87, 165, 168.
  Vink: Dette kræver to `for`-løkker.

Opgave 31 — Reversering af vektor med `for`-løkke

Opret vektoren `v<-1:10` (talrækken 1, 2, ..., 10). Skriv derefter en `for`-løkke, der ved at bytte om på elementerne i `v` vender vektoren om (så den bliver talrækken 10, 9, ..., 1).

Vink: Løkken skal gå over tallene 1, 2, ..., 5. Det er nødvendigt at indføre en hjælpevariabel til midlertidigt at gemme det ene element når man vil bytte om på to elementer i vektoren.

Opgave 32 — Brownsk bevægelse i 2D

Denne opgave er en udbygning af eksemplet med simulering af Brownsk bevægelse i afsnit 15.2. Vi betragter nu en model for en partikels bevægelse i to dimensioner, hvor partiklens position til tidstrin t er (x_t, y_t). Partiklen starter i $(0,0)$ til $t = 0$. Mellem tid t og tid $t+1$ bevæger partiklen sig et lille stykke vej d_t i den tilfældige retning v_t (i radianer), hvor d_t er et (ligefordelt) tilfældigt tal mellem 0 og 1 og v_t er et (ligefordelt) tilfældigt tal mellem 0 og 2π. Vi har således:

$$\begin{aligned} x_{t+1} &= x_t + d_t \cdot \cos v_t, \\ y_{t+1} &= y_t + d_t \cdot \sin v_t, \end{aligned}$$

med d_t og v_t tilfældige af hinanden uafhængige tal som beskrevet ovenfor.

(1) Simuler 100 trin i ovenstående model, d.v.s. op til $t = 100$. Resultatet af simuleringen skal gemmes i to vektorer `X` og `Y` med hver 100 elementer hvor `X[1]`=x_1, `X[2]`=x_2 osv. og `Y[1]`=y_1, `Y[2]`=y_2 osv. Følg den grundlæggende fremgangsmåde fra afsnit 15.2.

(2) Lav et plot der viser partiklens bevægelse i 2D. Brug `plot` med `asp=1` for at sikre at enheder bliver lige lange på de to akser. Brug `type="l"` for at få tegnet "sporet" som en række forbundne linjestykker. Sørg for at også startpositionen (0,0) kommer med.

Tilføj med `points` en åben cirkel i startpositionen.

Tilføj med `points` en udfyldt cirkel (`pch=16`) i slutpositionen.

Opgave 33 — Funktioner der giver sandhedsværdier

Du skal i denne opgave lave nogle funktioner der tager et tal som argument og giver en sandhedsværdi (`TRUE` eller `FALSE`) som svar. I hvert tilfælde vil funktionskroppen bestå af et logisk udtryk.

Eksempel: Funktion der giver `TRUE` hvis argumentet er et positivt tal, `FALSE` ellers:

```
positiv <- function(x) { x>0 }
```

Vi kan afprøve funktionen, først på tallet 17, så på -1 og endelig på heltallene fra -2 til 2:

```
> positiv(17)
[1] TRUE
> positiv(-1)
[1] FALSE
> positiv(-2:2)
[1] FALSE FALSE FALSE  TRUE  TRUE
```

Definér og afprøv følgende funktioner:

- Funktionen `negativ`, der giver `TRUE` hvis argumentet er et negativt tal og `FALSE` ellers. Afprøv på heltallene fra -2 til 2.
- Funktionen `lille`, der giver `TRUE` hvis argumentet er et tal mindre end 10. Afprøv på heltallene fra 8 til 12.
- Funktionen `stort`, der giver `TRUE` hvis argumentet er et tal større end 100. Afprøv på heltallene fra 98 til 102.
- Funktionen `mellem`, der giver `TRUE` hvis argumentet er et tal mellem 10 og 100 inklusive. Afprøv på heltallene fra 8 til 12 samt fra 98 til 102.

Opgave 34 — Numerisk værdi med `if ... else`

Betragt nedenstående ufuldstændige funktion `num`:

```
num <- function(a) {
  ???
}
```

Færdiggør `num` så den som resultat giver den numeriske (absolutte) værdi af tallet `a`. Du skal bruge et `if ... else` udtryk i funktionen (hvis `a` er mindre end nul så skal resultatet være `-a`, ellers skal resultatet bare være `a`). Du skal *ikke* benytte R-funktionen `abs`.

Afprøv herefter at funktionen virker for tallene 3 og -3.

Opgave 35 — Liebig-Wagner funktion

I en Liebig-Wagner model for begrænset vækst (*Noter om matematik* afsnit A.8.5) indgår funktionen

$$y(t) = \begin{cases} \frac{at}{b} + c & \text{for } 0 \leq t < b \\ a + c & \text{for } t \geq b \end{cases}$$

Lad i det følgende $a = 7$, $b = 15$ og $c = 3$ og antag at det altid gælder at $t \geq 0$.

(1) Definér i R funktionen $y(t)$ med brug af et `if ... else` udtryk i funktionens krop. Afprøv at $y(0) = 3$, $y(6) = 5.8$, $y(15) = 10$ og $y(16) = 10$.

(2) Definér nu $y(t)$ v.h.a. funktionen `ifelse` i stedet for v.h.a. `if ... else`. Lav med `plot` en graf for $y(t)$ for $0 \leq t \leq 20$.

Opgave 36 — Fortegn

Betragt nedenstående ufuldstændige funktion `fortegn`:

```
fortegn <- function(a) {
  ???
}
```

Færdiggør `fortegn` så den som resultat giver -1, 0 eller 1 afhængig af fortegnet af tallet `a`: -1 hvis tallet `a` er negativt, 0 hvis `a` er nul og 1 hvis `a` er positivt. Du skal bruge sammensætningen af to `if ... else` udtryk i funktionen. Du skal *ikke* benytte R-funktionen `sign`.

Afprøv herefter at funktionen virker for tallene 3, 0 og -3.

Opgave 37 — Vektorer med logiske værdier

Konstruér en vektor `x` indeholdende 100 ligefordelte tilfældige tal mellem 0 og 1 som følger:

```
x <- runif(100)
```

Man kan med et logisk udtryk afgøre, hvilke tal fra `x` der har bestemte egenskaber.
Eksempel: `x>0.5` giver en vektor med `TRUE` på alle pladser hvor `x` større end $\frac{1}{2}$ og `FALSE` på de øvrige pladser.

Ved at kombinere dette med `sum` funktionen kan man bestemme hvor mange af tallene i `x` der har en bestemt egenskab, idet `sum` tæller `TRUE` som 1 og `FALSE` som 0.
Eksempel: `sum( x>0.5 )` giver hvor mange tal i `x` der er større end $\frac{1}{2}$.

Brug nu et logisk udtryk samt `sum` til at bestemme følgende:

1. Hvor mange tal i `x` er mindre end 0.1?
2. Hvor mange tal i `x` er mellem 0.25 og 0.75?
3. Hvor mange tal i `x` er mindre end 0.1 *eller* mellem 0.25 og 0.75?
4. For hvor mange tal x_i i `x` er $\sin(1/x_i)$ større end nul?

Bemærk, at de nøjagtige svar på alle disse spørgsmål vil variere fra gang til gang man løser opgaven.

Opgave 38 — Delmængder af datasæt med logiske udtryk

Indlæs datasættet fra filen `reaktion.txt` og kald det `data` (brug `read.table` med `header=TRUE` og `dec=","`).

Brug nu funktionen `subset` i udtryk af formen `data1 <- subset(data, ???)`, hvor du udskifter de tre spørgsmålstegn med et logisk udtryk (se afsnit 20.2 og afsnit 19.1), til at fremstille følgende datasæt, der alle er delmængder af `data`:

- `data1`, der indeholder de 25 observationer hvor `Tid` er mindre end eller lig `50`.
- `data2`, der indeholder de 21 observationer hvor `S2` er større end `S1`.
- `data3`, der indeholder de 13 observationer hvor den numeriske forskel på `S1` og `S2` er mindre end 0.1 (brug `abs` til at tage numerisk værdi).
- `data4`, der indeholder de 13 observationer hvor `S3` er mellem 2 og 6 inklusive.

Brug `nrow` til at tjekke at datasættene har de ønskede antal rækker, fx skal `nrow(data1)` gerne give 25.

Opgave 39 — Nogle funktioner med `if ... else`

(1) Betragt følgende funktion `f`:

```
f <- function(a,b) {
   if (a>b) a else b
}
```

Besvar, *uden* at taste funktionen ind og prøve den, hvilket tal giver `f(1,2)`? Hvad giver `f(5,3)`?

Beskriv i ord: Hvad giver kaldet `f(x,y)` for to vilkårlige tal x og y?

(2) Betragt følgende funktion `g`:

```
g <- function(a,b,c) {
   if (a>c && b>c) {
      c
   } else {
      if (a>b) b else a
   }
}
```

Besvar, *uden* at taste funktionen ind og prøve den, hvilket tal giver `g(1,2,3)`? Hvad giver `g(5,4,3)`?

Beskriv i ord: Hvad giver kaldet `g(`x`,`y`,`z`)` for tre vilkårlige tal x, y og z?

(3) Betragt funktionen `h` defineret her:

```
h <- function(a,b) { if (a<b) h(a+1,b-a+1) else a-b }
```

Besvar, *uden* at taste funktionen ind og prøve den, hvilket tal giver `h(2,9)`?
Brug papir og blyant til at regne det ud, som følger: `h(2,9) = h(2+1,9-2+1) = h(3,8) = ....`

Opgave 40 — Midterste af tre tal

Skriv en funktion `midt <- function(a,b,c)` som giver det midterste af de tre tal `a`, `b` og `c` når de ordnes efter størrelse, altså fx:

```
> midt(1,2,3)
[1] 2
> midt(2,3,1)
[1] 2
> midt(3,1,2)
[1] 2
```

Brug udelukkende `if ... else` udtryk og ingen af R's statistiske funktioner.

Afprøv at din funktion virker ved alle rækkefølger af tallene 1, 2 og 3, altså såvel (1,2,3) som (1,3,2), (2,1,3), (2,3,1), (3,1,2) og (3,2,1).

Opgave 41 — Nogle rekursive funktioner med `if ... else`

(1) Betragt funktionen `f` defineret her:

```
f <- function(a,b) { if (a>b) f(a-b,b)+1 else a }
```

Besvar, *uden* at taste funktionen ind og prøve den, hvad giver `f(7,2)`?
Brug papir og blyant til at regne det ud, som følger: `f(7,2) = f(7-2,2)+1 = f(5,2)+1 = ...`

(2) Lav en *rekursiv* funktion fib(n) der for heltal $n \geq 1$ giver Fibonacci-tal nummer n. Brug følgende rekursive definition af Fibonacci-tallene som udgangspunkt:

$$\text{fib}(n) = \begin{cases} 1 & \text{hvis } n \leq 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{hvis } n > 2 \end{cases}$$

(3) Lav en *rekursiv* funktion `sfd` der for to naturlige tal a og b finder deres *største fælles divisor*[23] ved hjælp af *Euklids algoritme*[24], der kan opskrives som følger:

[23] Den største fælles divisor for to naturlige tal er det største heltal, der går op i dem begge.

[24] *Eukleides* af Alexandria (ca. 365 fvt. – 275 fvt.), græsk matematiker hvis kendteste værk *Elementer* bl.a. indeholder denne algoritme til at finde største fælles divisor af to naturlige tal.

$$\text{sfd}(a,b) = \begin{cases} a & \text{hvis } b = 0, \\ \text{sfd}(b, a \bmod b) & \text{ellers.} \end{cases}$$

Notationen $a \bmod b$ betyder her a modulo b, dvs. resten ved heltalsdivision af a med b. I R skrives modulo operatoren `%%`, altså `a%%b` for $a \bmod b$.

Afprøv at `sfd(34,119)` giver 17, `sfd(12,20)` giver 4 og `sfd(53,1016)` giver 1.

Opgave 42 — Flere rekursive funktioner med `if ... else`

Betragt følgende *rekursivt definerede* funktion `fra0`, der for et givet heltal `a` ≥ 0 giver sekvensen af heltal fra 0 til `a`:

```
fra0 <- function(a) {
  if (a>0) c(fra0(a-1),a) else 0
}
```

Eksempler på kald:

```
> fra0(8)
[1] 0 1 2 3 4 5 6 7 8
> fra0(0)
[1] 0
```

(1) Skriv nu med ovenstående som skabelon en tilsvarende funktion `til0`, der for et heltal `a` ≤ 0 giver sekvensen af heltal fra `a` til 0, fx.:

```
> til0(-4)
[1] -4 -3 -2 -1  0
```

(2) Skriv en funktion `med0` der afhængig af fortegnet af sit argument virker enten som `fra0` eller `til0` ovenfor, fx.:

```
> med0(8)
[1] 0 1 2 3 4 5 6 7 8
> med0(-4)
[1] -4 -3 -2 -1  0
```

Opgave 43 — Simpel `while`-løkke

Brug en `while` løkke til at bestemme det mindste heltal n således at $n^2 > 500$. Check dit resultat ved at beregne $\sqrt{500}$.

Vink: Start med `n <- 0` før løkken, lad løkken køre så længe $n^2 \leq 500$ og tæl `n` op med én i hvert gennemløb. Kig på første eksempel i afsnit 19.5 (side 93).

Opgave 44 — Antal led i en sum

Brug en `while` løkke til at afgøre, hvad der er mindste heltal n således at

$$\sum_{x=1}^{n} \frac{1}{x} \geq 10$$

Vink: Kig på andet eksempel i afsnit 19.5. (side 94).

Opgave 45 — Håndkørsel af `while`

(1) Betragt denne R-kode:

```
a <- 2 ; b <- 9
while (a<b) { b <- b-a+1 ; a <- a+1 }
```

Brug papir og blyant til at håndkøre `while`-løkken og holde rede på de skiftende værdier af `a` og `b`, se eksemplet i afsnit 19.5.1. Hvad er efter løkken værdierne af `a` og `b`?

(2) Betragt funktionen `f` defineret her:[25]

```
f <- function(a,b) {
   s <- 0
   while (a>b) { s <- s+1 ; a <- a-b }
   a+s
}
```

Find ved at håndkøre funktionen ud af: Hvad giver `f(12,3)`?
Vink: Startværdien for `a` og værdien af `b` er bestemt af funktionskaldet. Håndkør lige som i første delopgave løkken for at holde rede på de skiftende værdier af `a` og `s`. Funktionens returværdi er `a+s`, med de værdier variablene har *efter* løkken.

Opgave 46 — Snake-eyes

Man kan simulere slag med en terning ved at bruge funktionen `runif` til at generere (pseudo)tilfældige tal.

Følgende funktion genererer en vektor med `n` tal hvor hvert tal er et "slag" med en almindelig 6-sidet terning:

```
terning <- function(n) { floor(runif(n,1,7)) }
```

Forklaring: Med `runif(n,1,7)` genereres `n` tilfældige tal i intervallet $]1,7[$. Med `floor` rundes *ned* til nærmeste heltal, hvorved fås heltal mellem 1 og 6, inklusive.

[25] Funktionen er en ikke-rekursiv udgave af funktionen `f` fra opgave 41 delopgave (1).

(1) Indtast funktionen `terning` givet ovenfor. Afprøv funktionen således:

```
> terning(1)
[1] 3
> terning(10)
 [1] 2 1 4 3 3 6 4 2 2 1
```

(du får sandsynligvis andre resultater).

(2) Vi vil nu empirisk undersøge hvor mange forsøg, der skal til, før man med to terninger slår to ettere (snake-eyes).

Strategi: Lav "slag" med to terninger ind til summen af dem er to. Brug en `while`-løkke til at gentage slagene. Optæl antal slag (antal gentagelser i `while`-løkken) i en variabel.

Færdiggør nedenstående funktion, der skal returnere antal gange der blev slået før vi fik summen 2:

```
antalslag <- function() {
   n <- 1       # Vi slår mindst 1 slag.
   while ( ... ) {
      ...
   }
   n            # Efter løkken er antallet af slag optalt i variabel n
}
```

Afprøv funktionen `antal` således:

```
> antalslag()
[1] 8
> antalslag()
[1] 40
```

(du får sandsynligvis andre resultater).

(3) Lav en vektor `x` med 1000 værdier fra `antalslag` som følger:[26]

```
x <- replicate(1000, antalslag())
```

Brug herefter `summary` på `x` til at finde ud af det gennemsnitlige og maksimale antal slag, der skulle slås for at få snake-eyes. Lav endelig et *histogram* over værdierne i `x` som følger:

```
hist(x, plot=TRUE)
```

Opgave 47 — Begrænset Brownsk bevægelse

Denne opgave tager udgangspunkt i eksemplet med simulering af Brownsk bevægelse i afsnit 15.2. Vi betragter nu en model for en partikel der bevæger sig i mellemrummet mellem to parallelle plader placeret tæt på hinanden. Partiklen starter midt mellem pladerne og bevæger sig tilfældigt ind til den

[26] Forskellen på `replicate(`*n*`,`*udtryk*`)` og `rep(`*udtryk*`,`*n*`)` er at med `rep` udregnes *udtryk* én gang og resultatet gentages *n* gange, så man får en vektor med *n* identiske tal, mens med `replicate` udregnes *udtryk* *n* gange, hvilket med et udtryk, der ikke giver samme resultat hver gang, vil give en vektor med *n* forskellige tal.

støder på en af pladerne, hvor den så hænger fast og dens bevægelse dermed slutter. Pladerne er tilpas store til, at partiklen i praksis aldrig bevæger uden for mellemrummet før den rammer den ene af dem, og vi modellerer derfor kun partiklens position i én dimension, vinkelret på pladerne. Vi er interesserede i hvor lang tid, der går, før partiklen rammer den ene eller den anden plade.

Som i afsnit 15.2 er modellen

$$x_{t+1} = x_t + \epsilon_t,$$

hvor x_t er partiklens position til tidspunktet t og ϵ_t et tilfældigt tal mellem -1 og 1. Det gælder at $x_0 = 0$. Pladerne er placeret i $x = -5$ og $x = 5$.

(1) Omskriv den første af de to `for`-løkker fra afsnit 15.2 til en `while`-løkke der kører så længe partiklens position `x` er mellem -5 og 5. Husk først at give `x` værdien nul inden løkken. Indføj en variabel `antal` hvor antallet af trin, partiklen tager, tælles op (**vink:** se også på eksemplet i afsnit 19.5.2).

(2) Skriv nu en funktion `antaltrin`, der indkapsler simulering af en partikels vandring mellem de to plader og returnerer antallet af trin, partiklen tog. Du skal med andre ord pakke besvarelsen af første delopgave ind i en funktion således at du let kan gentage simulationen mange gange (lige som med funktionen `antalslag` i opgave 46).

Du burde nu kunne skrive `antaltrin()` og få et antal trin som svar (afprøv dette nogle gange).

(3) Lav en vektor `a` med 1000 simulerede værdier fra `antaltrin` som følger:

```
a <- replicate(1000, antaltrin())
```

Brug herefter `summary` på `a` til at finde ud af det gennemsnitlige, minimale og maksimale antal trin, partiklerne vandrede før de ramte en plade. Lav endelig et *histogram* over værdierne i `a` som følger:

```
hist(a, plot=TRUE, breaks=50)
```

APPENDIKS

A Sådan installerer du R

A.1 Grundlæggende installation af R

Hjemmesiden for R er http://www.r-project.org/ hvorfra man kan hente den nyeste udgave af R, samt manualer, ekstra pakker osv. Der findes en kopi af hele hjemmesiden i Danmark på adressen http://cran.dk.r-project.org/. Brug denne adresse hvis du skal installere R.

I det følgende gennemgås installation af R version 2.11.1, som var seneste version da dette blev skrevet. Når der kommer nye versioner af R skal man udskifte versionsnummeret 2.11.1 med det seneste versionsnummer overalt i nedenstående.

A.1.1 Installation under Windows

Gør sådan her for at installere R under Microsoft Windows:

- Gå til http://cran.dk.r-project.org/
- Klik på Windows under Download and Install R.
- Klik på base.
- Klik på Download R 2.11.1 for Windows og vælg at gemme filen på disk (fx Skrivebord).
- Dobbeltklik på `R-2.11.1-win32.exe` når download er færdig, vælg Dansk installationsdialog, acceptér licensen, vælg Brugerinstallation, og acceptér alle de foreslåede indstillinger.[27]

  Når installationen er slut kan du slette `R-2.11.1-win32.exe` fra disken.

A.1.2 Installation under MacOS X

Gør sådan her for at installere R under MacOS X:

- Gå til http://cran.dk.r-project.org/
- Klik på MacOS X under Download and Install R.
- Klik på R-2.11.1.pkg under Files.
- Der burde, når filen er hentet, automatisk åbne en installations-dialog.[28]
- Acceptér licensen og alle de foreslåede indstillinger.

  Når installationen er færdig, kan du slette filen `R-2.11.1.pkg` fra dine overførsler.
- Pakken indeholder både en 32-bit udgave og en 64-bit udgave af R, der installeres som hhv. `R` og `R64` i din applikations-menu. De burde i princippet køre ens, men 32-bit udgaven er mere gennemtestet end den anden, som til gengæld muligvis er lidt hurtigere end 32-bit udgaven.

[27] Hvis du installerer under Windows Vista eller Windoes 7 kan der senere opstå problemer med rettigheder. Se nærmere om dette i "R for Windows FAQ" på R-hjemmesiden; følg fx link'et How do I install R when using Windows Vista? fra R for Windows download siden.

[28] Hvis ikke, skal du finde papkasse-ikonen med navnet `R-2.11.1` under dine overførsler og dobbelt-klikke på den.

A.2 Installation af ekstra R-pakker

Der findes et stort antal ekstra pakker til R, især til specielle statistiske analyser og til import og eksport af data. Dette appendiks gennmgår som eksempel installation af pakken `scatterplot3d`, som kan bruges til at lave 3-dimensionelle punktgrafer (scatterplots).

Først og fremmest skal din computer være tilsluttet internettet. Du skal så benytte en af nedenstående fremgangsmåder, afhængig af operativsystem.

A.2.1 Installation af pakker under Windows

I Windows-udgaven af R skal du så gøre sådan her:[29]

- Fra RGui menuen vælges Packages || Set CRAN mirror || Denmark.
- Fra RGui menuen vælges Packages || Install package(s).
- Efter et øjeblik dukker der en liste af pakker op. Find pakken du vil installere på listen (fx `scatterplot3d`), markér den, og klik *OK*.
- Første gang vil R spørge om du vil lave et personligt bibliotek til dine R-pakker. Det giver færrest problemer at sige "Ja" (specielt under Windows Vista), men hvis du ønsker at installere pakken også for andre brugekonti på din computer skal du svare "Nej".
- Hvis alt går godt kommer der en meddelelse om "successfully unpacked" i R Console. Du kan dog under Windows Vista risikere også at få nogle fejlmeddelelser på grund af manglende rettigheder til at installere nogle hjælpetekstfiler – dette burde dog ikke have nogen indflydelse på brugen af pakken.

A.2.2 Installation af pakker under MacOS X

I MacOS-udgaven af R skal du så gøre sådan her:

- Fra R menuen vælges Packages & Data || Package Installer.
- I Installer vinduet skal du sørge for at den øverste rullemenu står på *CRAN (binaries)*. Tryk så på knappen *Get List*.
- Efter et øjeblik dukker der en liste af pakker op. Find pakken du vil installere på listen (fx `scatterplot3d`) og markér den.
- Du kan nederst under "install location" vælge om pakken skal installeres for alle brugere af computeren (system level) eller kun for den aktuelle bruger (user level). Hvis du har adgang til at installere programmer på computeren bør du beholde indstillingen "At System Level".
- Tryk på knappen *Install Selected*. Når pakken er installeret vil den i oversigten blive vist med et nummer under "Installed Version" – dette er ud over en række meddelelser i R Console den eneste kvittering for, at installationen er gået godt.
- Du kan nu lukke Installer vinduet.

[29] Der kan være problemer med manglende administratorrettigheder, specielt under Windows Vista og Windows 7 – se nærmere herom på R's hjemmeside. Som hovedregel skal du for at installere pakker køre R som den samme bruger, som da du installerede R – dog ikke hvis du vælger at benytte muligheden for at oprette dit eget bibliotek til pakker.

B Nogle almindelige R fejlmeddelelser

Nedenstående liste af fejlmeddelelser er langt fra alle fejl du kan møde i R, men forhåbentlig er de mest almindelige og forvirrende med:

Error: object "*navn*" not found

Variablen/funktionen *navn* er ikke defineret eller tildelt.
Hvis det ukendte navn burde være en standard R-funktion har du lavet en skrivefejl – tjek at du har skrevet rigtigt. *Husk at der er forskel på store og små bogstaver.*
Hvis det ukendte navn er en parameter eller variabel har du ikke defineret den med en tildeling endnu.

Error: syntax error, unexpected SYMBOL, expecting ...

Typisk et symptom på at du har glemt et gangetegn eller en anden operator eller måske et komma. Du har for eksempel skrevet `2x` med "matematiknotation" i stedet for `2*x` som er R-notation.

Error: syntax error, unexpected NUM_CONST, expecting ...

Samme fejl som ovenstående. Måske har du glemt et komma mellem argumenterne i et funktionskald og har skrevet `c(2 3)` i stedet for `c(2,3)` eller `plot(f, 0 5)` i stedet for `plot(f,0,5)`.

Error: attempt to apply non-function

Du har noget som ikke er en funktion stående foran en parentes. Typiske eksempler:
Glemt gangetegn: `2(x+4)` hvor der skulle stå `2*(x+4)`.
Ikke funktion: `f(3)` hvor `f` er defineret som noget andet end en funktion. Måske er du kommet til at bruge en af de indbyggede funktioner som en variabel, dvs. har tildelt en ny værdi til et "reserveret" navn, fx `c` eller `t`. I dette tilfælde må du retablere den oprindelige betydning ved at skrive fx `rm(c)`.
Forkert type parentes: funktionskaldet `x(3)` hvor du mente indekseringen `x[3]`.

Error in *udtryk* : object is not subsettable

Du har brugt firkantede parenteser efter noget der ikke er en vektor, matrix eller lignende. Typisk er du kommet til at bruge firkantede i stedet for runde parenteser i et funktionskald og har skrevet `f[2]` i stedet for `f(2)`.

Error in C(...) : object not interpretable as a factor

Du er kommet til at skrive `C(...)` med stort "C" i stedet for `c(...)` med lille "c".

Error in plot.xy(xy.coords(x, y), type = type, ...) :
plot.new has not been called yet

Du har startet et plot med `plot(...,add=TRUE)` eller med `points` eller `lines`. Husk at et nyt plot altid startes med `plot` uden `add=TRUE`.

Warning messages:
In ... : "add" is not a graphical parameter

Du har brugt `add=TRUE` i andet end et funktionsplot (for eksempel `points`).

In ... : "*parameter*" is not a graphical parameter

Man får en hel række af disse hvis man angiver et forkert parameternavn til `plot`, `points` eller en af de andre plotfunktioner. Parameteren *parameter* findes ikke, dvs. du har måske skrevet navnet forkert.

C Polynomiel regression og regressionskurve i R

I afsnit 13 så vi hvordan man i R nemt foretager lineær regression med funktionen `lm`.

Det er næsten lige så nemt at beregne en polynomiel regressionskurve, for eksempel at finde det andengradspolynomium der bedst passer til væksthormonforsøgsdatasættet `d` (se afsnit 13). Man bruger igen funktionen `lm` til at finde de koefficienter c, b og a der bedst passer med $y = c + bx + ax^2$ hvor y er `Kontrol` og x er `Tid`.

```
> preg <- lm(Kontrol ~ Tid + I(Tid^2), data=d)
> preg
Coefficients:
(Intercept)          Tid     I(Tid^2)
  6.318e+00   -2.801e-03    1.523e-06
```

Det beregnede andengradspolynomium `preg` er $y = 6.318 - 0.002801x + 0.000001523x^2$. I kaldet til `lm` betyder funktionen `I` i R-formlen at `Tid^2` skal betragtes som en variabel der indgår i den lineære model, ikke som en specifikation i R's modelsprog, hvor operatoren `^` har speciel betydning. Hvis man inkluderer regneudtryk som variable er det altid sikrest at pakke dem ind i `I()`.

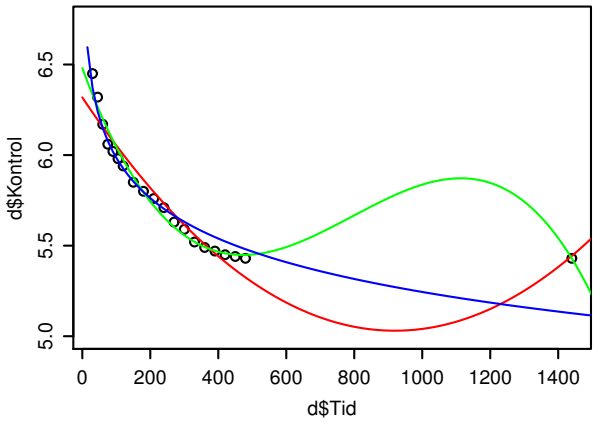
For at indtegne polynomiet på plottet over dataene skal man opbygge en vektor `m` af x-værdier, såkaldte "støttepunkter". For hvert af disse beregnes polynomiets y-koordinat ved hjælp af funktionen `predict`. Tilsammen udgør `m` og værdierne beregnet af `predict` et sæt (x, y)-koordinater for polynomiet, og de bruges i funktionen `lines` til at tilføje polynomiet til dataplottet:

```
> m <- seq(0, 1500, len=101)
> plot(d$Tid, d$Kontrol, ylim=c(5, 6.75))
> lines(m, predict(preg, data.frame(Tid=m)), col="red")
```

Polynomiel regression med hensyn til polynomier af højere grad, eller med hensyn til andre transformerede variable, kan laves på samme måde:

```
> preg3 <- lm(Kontrol ~ Tid + I(Tid^2) + I(Tid^3), data=d)
> lines(m, predict(preg3, data.frame(Tid=m)), col="green")
> logreg <- lm(Kontrol ~ I(log(Tid)), data=d)
> lines(m, predict(logreg, data.frame(Tid=m)), col="blue")
```

Det resulterende plot med både de oprindelige data og de tre regressionskurver ses i figur 53, der i øvrigt viser at dataene dårligt kan beskrives med af nogle af regressionerne.



Figur 53: Forsøgsdata med tre regressionskurver.

D Plot af punkter og kurver i rummet med `scatterplot3d` i R

Der er ingen funktion til at lave plot af punkter i rummet (3D scatterplot) i standard R, man er nødt til at installere den ekstra pakke `scatterplot3d` (se afsnit A.2 for en vejledning).

Under forudsætning af, at pakken `scatterplot3d` er installeret, gør man nu følgende for at få adgang til funktionen `scatterplot3d`:

```
> library("scatterplot3d")
```

Lad os først producere nogle tilfældige data, der skal plottes. Først defineres `x` og `y` til at være to vektorer med hver 100 ligefordelte tilfældige tal mellem 0 og 10:

```
> x <- runif(100,0,10)
> y <- runif(100,0,10)
```

Herefter konstruerer vi `z` således at vi til hvert talpar `(x[i],y[i])` får `z[i]` på planen $z = x + 2y + 3$ plus et normalfordelt tilfældigt tal (støj):

```
> z <- x + 2*y + 3 + rnorm(100)
```

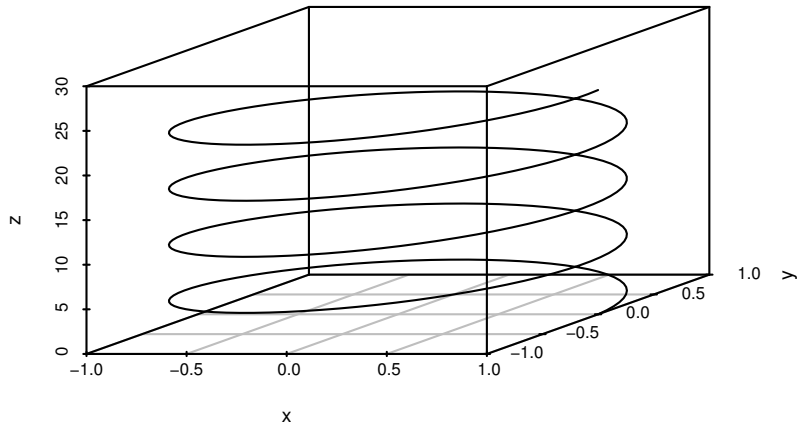
Herefter kan vi tegne et 3D scatterplot:

```
> scatterplot3d(x, y, z)
```

Det er muligt at variere 3D projektionen med parameteren `angle`. Vi kan også bruge `scatterplot3d` til at tegne kurver i rummet:

```
> t <- seq(0, 8*pi, len=300)
> x <- cos(t)
> y <- sin(t)
> z <- t
> scatterplot3d(x, y, z, type="l")
```

I ovenstående har vi 300 støttepunkter for værdier af parameteren `t` mellem 0 og 8π. Argumentet `type="l"` (bogstavet ℓ) angiver at `scatterplot3d` skal forbinde punkterne med linjer. Resultatet ses nedenfor:



E Litteratur om R

- *An Introduction to R* er en introduktion til R-sproget, statistisk analyse og grafik.
  Fås fra `http://cran.r-project.org/doc/manuals/R-intro.pdf`
- *R Language Definition* beskriver R-sproget i detaljer og er nyttig hvis man skal skrive sine egne R-funktioner. Fås fra `http://cran.r-project.org/doc/manuals/R-lang.pdf`
- *R Data Import/Export* beskriver hvordan man får data ind i og ud af R.
  Fås fra `http://cran.r-project.org/doc/manuals/R-data.pdf`
- *R: A Language and Environment for Statistical Computing* indeholder alle hjælpefiler fra R's standardpakker og anbefalede pakker. Fås fra `http://cran.r-project.org/doc/manuals/fullrefman.pdf`
- Peter Dalgaard: *Introductory Statistics with R*. 2nd edition. Springer 2008. ISBN 0-387-79053-5.
- Paul Murrell: *R Graphics*. Chapman & Hall/CRC Computer Science 2005. ISBN 158488486X.
- Flere bøger kan findes ved at søge efter `R statistics` på `http://www.amazon.com/`

F Facitliste til opgaver

Dette er en "facitliste" til opgaverne i afsnit 23, i den forstand at den viser resultater (R Console output) og grafer fra opgaverne, for de opgaver, hvor de ønskede resultater ikke står direkte i opgaveteksten. De egentlige *løsninger* af opgaverne står *ikke* her. Løsninger til R-opgaver er nemlig primært den R kode (i form af indtastninger og/eller scripts), der skal til for at producere det R Console output og de grafer, der står i denne facitliste.

Tanken er, at man under løsning af opgaverne kan sammenligne hvad man får ud af R med denne "facitliste". Hvis det ikke stemmer overens er ens R-kode forkert.

Opgave 2

Delopgave (1)

Værdierne af $f(1)$ og $f(7)$:

```
[1] -3
[1] 63
```

Delopgave (2)

Værdierne af $f(1)$ og $f(7)$:

```
[1] -2
[1] 112
```

Delopgave (3)

Værdierne af $g(1)$ og $g(7)$:

```
[1] 5
[1] 185
```

Delopgave (4)

Værdierne af $h(1)$ og $h(7)$:

```
[1] -1
[1] 455
```

Opgave 3

De tre funktionsværdier:

```
[1] 0.001006388
[1] 0.4060058
[1] 3
```

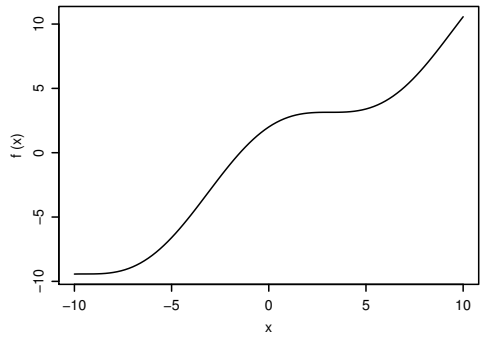
Opgave 4

Delopgave (1)

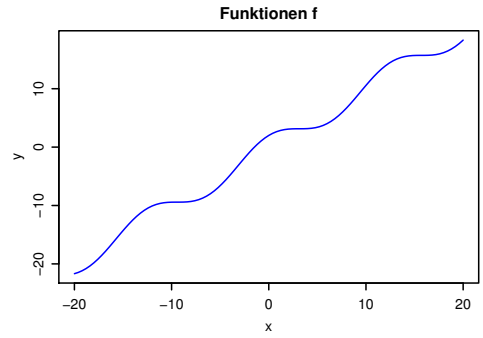
De tre funktionsværdier:

```
[1] 0.7551651
[1] 2
[1] 2.755165
```

Delopgave (2)



Delopgave (3)



Opgave 5

De 5 gange anvendelse af f giver:

```
Error in f(2) : object "a" not found
[1] 8
[1] -8
[1] 1.414214
[1] NaN
```

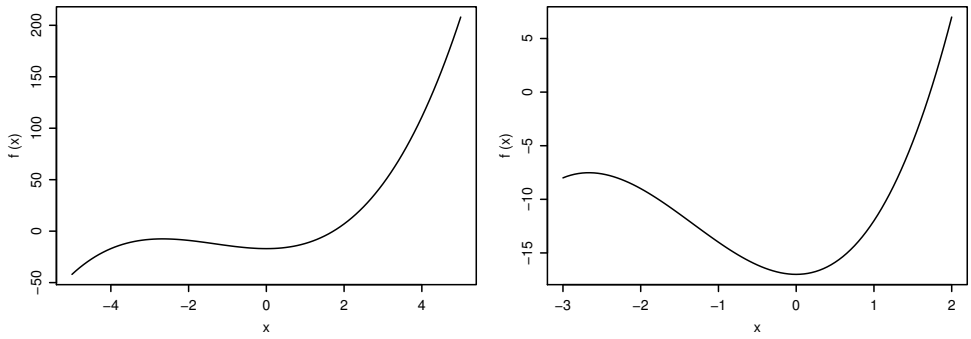
Opgave 6

Delopgave (1)

De tre funktionsværdier:

```
[1] -12
[1] -7.518519
[1] -17
```

Delopgave (2)



Delopgave (3)

Nulpunktet (ikke hele output):

```
$root
[1] 1.723439
```

Opgave 7

Løsningerne til ligningen (grafen vises ikke her):

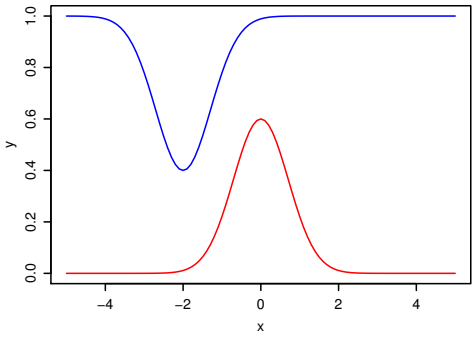
```
[1] -0.8100384
[1] 1.168259
[1] 2.641794
```

Opgave 8

Løsningerne til ligningen (grafen vises ikke her):

```
[1] -1.869779
[1] -1.056969
[1] -0.08364099
[1] 1.040338
```

Opgave 9



Opgave 10

Den ønskede graf er vist i selve opgaven.

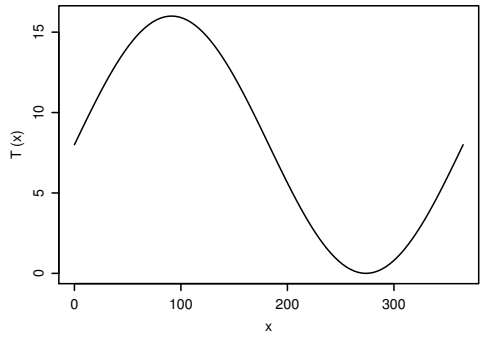
Opgave 11

Integralets værdi:

```
145.6949 with absolute error < 1.6e-12
```

Opgave 12

Delopgave (1)



Delopgave (2)

Værdien af `Graddage(10)` (også givet i opgaven):

```
[1] 86.8687
```

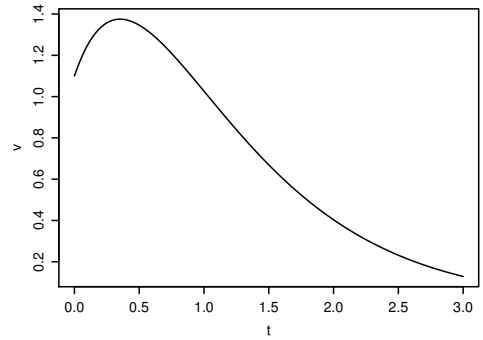
Delopgave (3)

Løsning til ligningen:

```
[1] 34.05216
```

Opgave 13

Delopgave (1)



Delopgave (2)

Værdien af integralet:

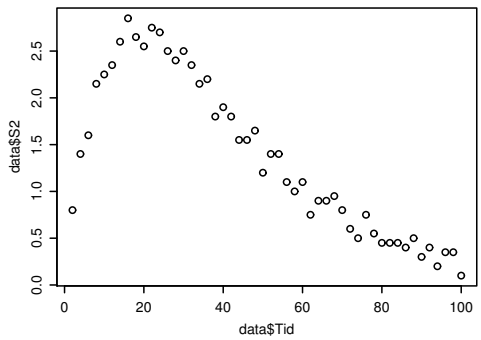
```
2.288889 with absolute error < 0.00018
```

Opgave 14

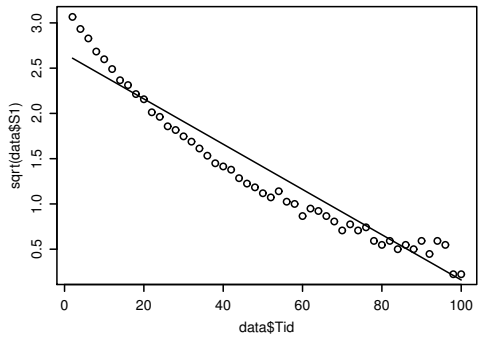
Delopgave (2)

Output fra `summary` er vist i opgaven.

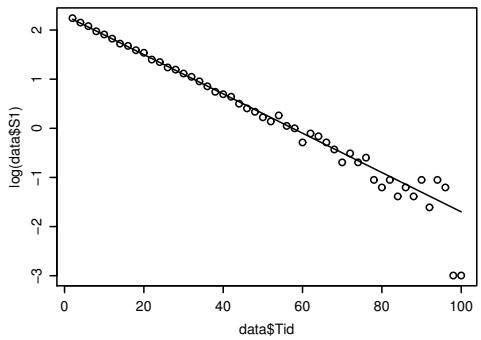
Delopgave (3)



Delopgave (4)



Delopgave (5)



Opgave 15

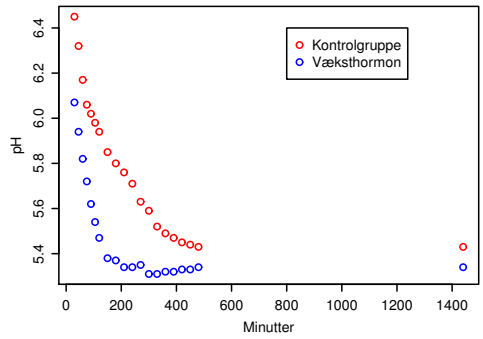
Delopgave (2)

Output fra `summary`:

```
      Tid             Kontrol          Vaekst
Min.   :  30.0   Min.   :5.430   Min.   :5.310
1st Qu.: 101.2   1st Qu.:5.485   1st Qu.:5.330
Median : 225.0   Median :5.735   Median :5.345
Mean   : 287.2   Mean   :5.776   Mean   :5.478
3rd Qu.: 367.5   3rd Qu.:5.990   3rd Qu.:5.560
Max.   :1440.0   Max.   :6.450   Max.   :6.070
```

Delopgave (3) og (4)

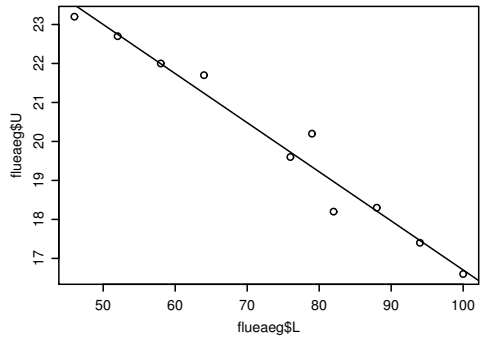
Dette er den komplette graf inklusiv signaturforklaring fra delopgave (4):



Opgave 16

Delopgave (2) og (4)

Dette er den komplette graf inklusiv regressionslinjen fra delopgave (4):



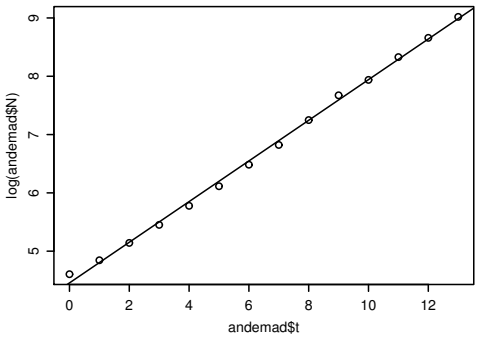
Delopgave (3)

Man får værdierne $a = -0.1259,\ b = 29.2967$.

Opgave 17

Delopgave (2) og (4)

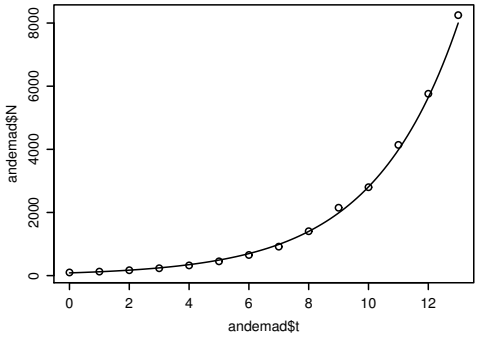
Dette er den komplette graf inklusiv regressionslinjen fra delopgave (4):



Delopgave (3)

Man får værdierne $b = exp(4.4555) = 86.09522,\ r = 0.3486.$

Delopgave (5)

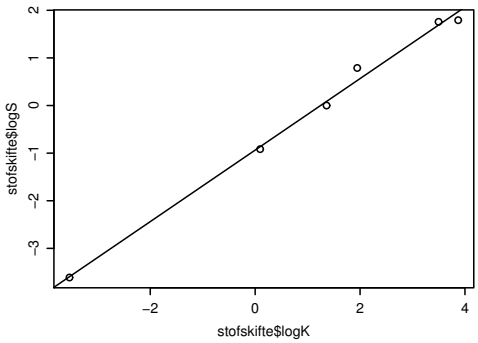


Opgave 18

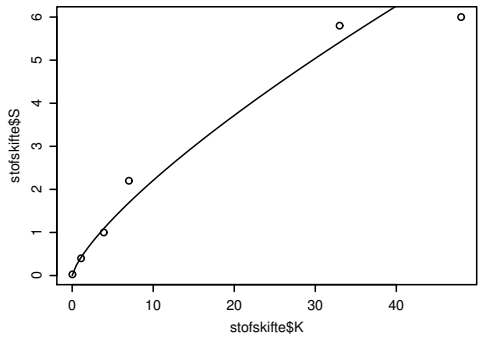
Delopgave (2)

Man får $b = 0.3920062,\ a = 0.7509.$

Delopgave (3)



Delopgave (4)



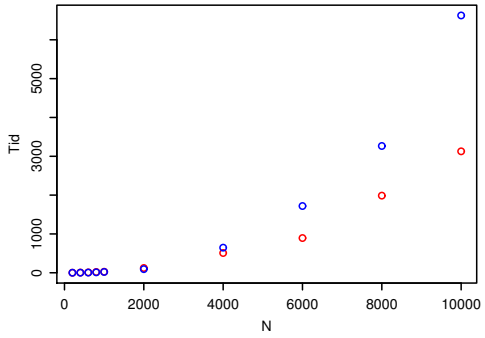
Opgave 19

Delopgave (2)

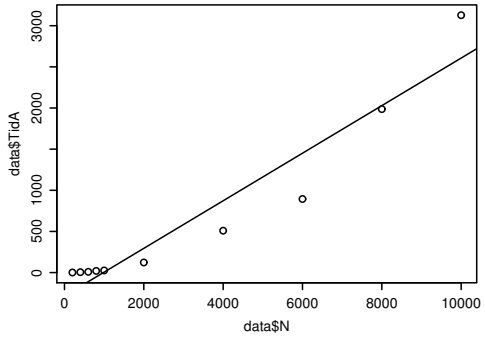
Output fra `summary`:

```
       N              TidA              TidB
 Min.   :  200   Min.   :   1.10   Min.   :   0.3
 1st Qu.:  650   1st Qu.:  12.00   1st Qu.:   6.6
 Median : 1500   Median :  74.35   Median :  57.0
 Mean   : 3300   Mean   : 670.16   Mean   :1238.9
 3rd Qu.: 5500   3rd Qu.: 797.85   3rd Qu.:1450.7
 Max.   :10000   Max.   :3127.70   Max.   :6626.8
```

Delopgave (3)

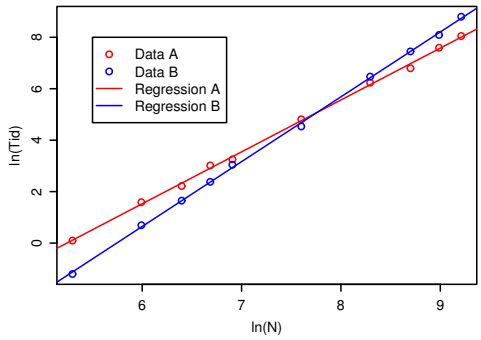


Delopgave (4)



Delopgave (5), (6) og (7)

Dette er den komplette graf inklusiv regressionslinjer fra delopgave (6) og signaturforklaring fra delopgave (7):



Opgave 23

Delopgave (2)

De to matrixprodukter, AB og BA:

```
     [,1] [,2]
[1,]   18   22
[2,]   41   54

     [,1] [,2]
[1,]   34   26
[2,]   47   38
```

Opgave 24

Delopgave (1)

Den inverse matrix:

```
      [,1]  [,2]
[1,] -0.26  0.14
[2,]  0.18 -0.02
```

Delopgave (2)

Løsning til ligningen:

```
[1] -1.06  0.58
```

Opgave 26

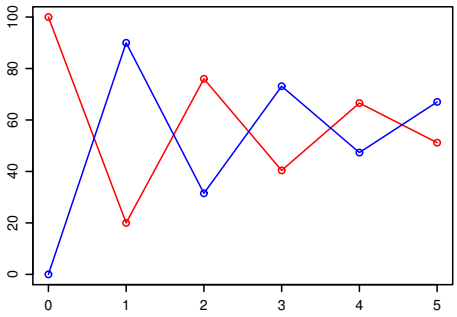
Delopgave (1)

Matricen V:

```
      v0
[1,] 100 20 76.0 40.400 66.58000 51.17900
[2,]   0 90 31.5 73.125 47.32875 67.02131
```

Det kan godt være rigtigt selv om søjlerne har en anden overskrift, det er tallene der er vigtige.

Delopgave (2)



Opgave 27

Plottet er givet i opgaven. Læg specielt mærke til at der ikke er nogen angivelse af 0.00 på y-aksen.

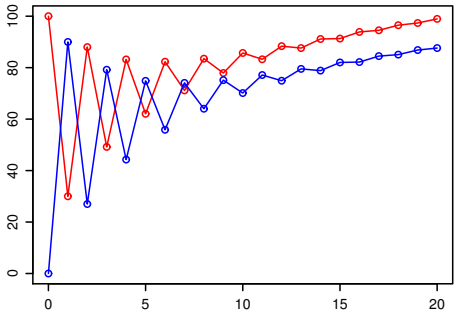
Opgave 28

Delopgave (1)

Den færdige matrix V:

```
     [,1] [,2] [,3] [,4]  [,5]   [,6]    [,7]     [,8]     [,9]    [,10]
[1,]  100   30   88 49.2 83.20 62.064 82.3168 71.14944 83.49798 77.92719
[2,]    0   90   27 79.2 44.28 74.880 55.8576 74.08512 64.03450 75.14819
        [,11]    [,12]    [,13]    [,14]    [,15]    [,16]    [,17]
[1,] 85.70399 83.24838 88.35655 87.61014 91.13874 91.30705 93.88130
[2,] 70.13447 77.13359 74.92354 79.52089 78.84913 82.02487 82.17634
        [,18]   [,19]    [,20]    [,21]
[1,] 94.51734 96.4980 97.35208 98.94898
[2,] 84.49317 85.0656 86.84821 87.61688
```

Delopgave (2)



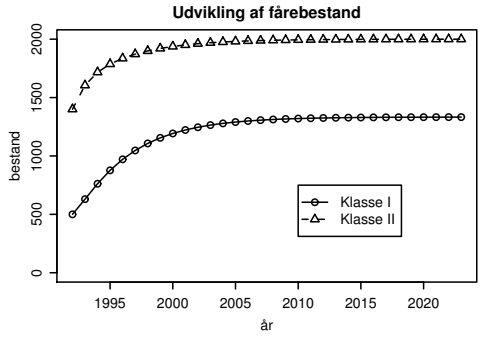
Opgave 29

Delopgave (1)

Den færdige matrix `V`:

```
     [,1] [,2]    [,3]     [,4]      [,5]     [,6]     [,7]     [,8]     [,9]
[1,]  500  630  762.00  876.750  971.0925 1046.883 1107.154 1154.865 1192.556
[2,] 1400 1605 1716.75 1786.838 1835.5894 1871.679 1899.288 1920.753 1937.568
        [,10]    [,11]    [,12]    [,13]    [,14]    [,15]    [,16]    [,17]
[1,] 1222.303 1245.770 1264.279 1278.877 1290.388 1299.467 1306.626 1312.272
[2,] 1950.789 1961.201 1969.406 1975.874 1980.975 1984.997 1988.169 1990.670
        [,18]    [,19]    [,20]    [,21]    [,22]    [,23]    [,24]    [,25]
[1,] 1316.724 1320.236 1323.004 1325.188 1326.910 1328.268 1329.339 1330.183
[2,] 1992.642 1994.198 1995.424 1996.392 1997.154 1997.756 1998.230 1998.604
        [,26]    [,27]    [,28]    [,29]    [,30]    [,31]    [,32]
[1,] 1330.849 1331.374 1331.788 1332.115 1332.373 1332.576 1332.736
[2,] 1998.899 1999.132 1999.316 1999.460 1999.574 1999.664 1999.735
```

Delopgave (2)



Opgave 30

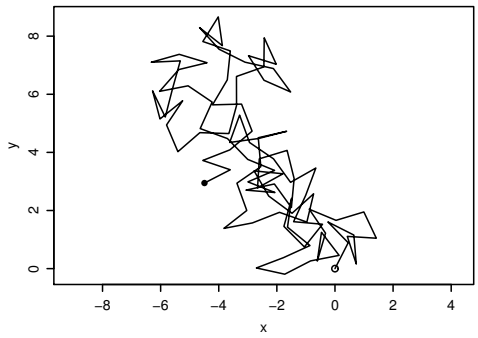
De fire vektorer `u`, `v`, `w` og `z`:

```
> u
 [1]  3  5  8 12 17 23 30 38 47 57
> v
 [1]    1    0    3   -2    9  -12   31  -54  117 -224
> w
 [1]  10  10   0  10 -10  20 -30  50 -80 130
> z
 [1]   1   4  10  13  35  38  84  87 165 168
```

Opgave 32

Delopgave (2)

Plottet bliver naturligvis forskelligt hver gang - her et eksempel:



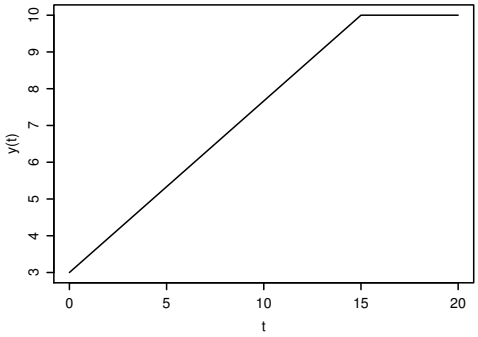
Opgave 33

Resultaterne af afprøvningerne:

```
> negativ(-2:2)
[1]  TRUE  TRUE FALSE FALSE FALSE
> lille(8:12)
[1]  TRUE  TRUE FALSE FALSE FALSE
> stort(98:102)
[1] FALSE FALSE FALSE  TRUE  TRUE
> mellem(8:12)
[1] FALSE FALSE  TRUE  TRUE  TRUE
> mellem(98:102)
[1]  TRUE  TRUE  TRUE FALSE FALSE
```

Opgave 35

Delopgave (2)



Opgave 37

Da tallene er tilfældige får man forskellige resultater hver gang. Nedenstående er dermed blot forventede værdier:

- 10
- 50
- 60 (skal være summen af de to foregående)
- 78

Opgave 38

Man kan som beskrevet i opgaven kontrollere at man har det rigtige antal rækker i et resultat ved at bruge `nrow`:

```
> nrow(data1)
[1] 25
> nrow(data2)
[1] 21
> nrow(data3)
[1] 13
> nrow(data4)
[1] 13
```

Opgave 39

Delopgave (3)

Funktionskaldet `h(2,9)` giver 2.

Opgave 41

Delopgave (1)

Funktionskaldet `f(7,2)` giver 4.

Opgave 43

Man skulle gerne ende med:

```
> n
[1] 23
> sqrt(500)
[1] 22.36068
```

Opgave 44

Man skulle gerne ende med:

```
> n
[1] 12367
```

Opgave 45

Delopgave (1)

Efter løkken har `a` værdien 5 og `b` værdien 3.

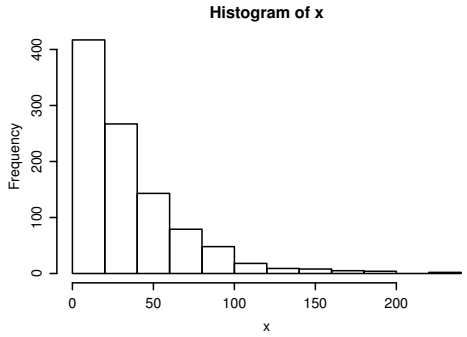
Delopgave (2)

Funktionskaldet `f(12,3)` giver 6.

Opgave 46

Delopgave (3)

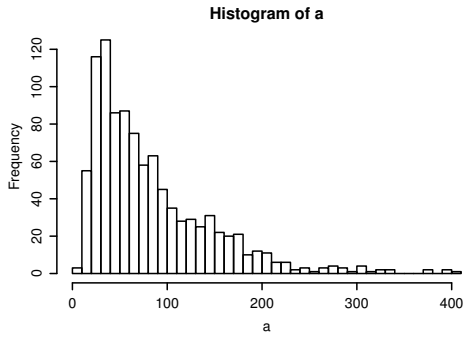
Man får naturligvis 1000 forskellige værdier i `x` hver gang, man løser opgaven, og dermed forskelligt output fra `summary`. Det vil være ekstremt usandsynligt at det laveste antal slag (minimum) ikke er 1. Gennemsnitligt antal slag burde være i omegnen af 36 og det største antal slag er nok over 200 - måske endda over 300. Her er et eksempel på hvordan histogrammet kan se ud:



Opgave 47

Delopgave (3)

Man får naturligvis 1000 forskellige værdier i `a` hver gang, man løser opgaven, og dermed forskelligt output fra `summary`. Det laveste antal trin før partiklen stopper (minimum) vil nok være 7 eller 8; det højeste kan være over 500. Det gennemsnitlige antal trin burde ligge omkring 85. Her er et eksempel på hvordan histogrammet kan se ud:



G Oversigt over R-funktioner og deres vigtigste parametre

Dette appendiks er tænkt som en slags "referencemanual" for de R-funktioner, der beskrives i noterne (og nogle få ekstra). Dette er langt fra alle de indbyggede funktioner i R!

Oversigten er emneopdelt og under hvert emne står funktionerne nogenlunde i den rækkefølge, de tidligere er introduceret i, eller i den rækkefølge man "naturligt" vil have brug for dem. For alfabetisk opslag kan man bruge indekset bagerst i noterne, hvorfra der med sidenummer både findes henvisning til teksten og til skemaerne i dette appendiks (henvisning til skemaerne er med *kursiv*).

Der er for de fleste funktioner et skema for hver funktion, men de allersimpleste funktioner som `exp` og `sqrt` er i stedet samlet i et par oversigtsskemaer.

Hovedet i hvert funktionsskema indeholder en kort beskrivelse af hvad funktionen gør samt en henvisning til de(t) relevante afsnit i noterne hvor en nærmere beskrivelse kan findes. Herefter kommer oftest et eller flere små eksempler på brug af funktionen – nogle gange med "kontekst" i form af anden R-kode. Så følger syntaks for brug af funktionen og en oversigt over de vigtigste parametre man kan/skal give. Rækkefølgen af beskrivelsen af parametrene er sådan at først beskrives obligatoriske parametre, dernæst dem man oftest vil bruge (og som er beskrevet nærmere i det relevante afsnit) og endelig nogle som ikke bruges så ofte men som er nyttige at kende – og som ikke nødvendigvis er beskrevet andre steder i disse noter. Denne rækkefølge er dog fraveget hvor parametre naturligt hører sammen i grupper. Mange af funktionerne tager flere parametre end dem der vises i skemaerne, der derfor ikke nødvendigvis er udtømmende. Slå eventuelt op i R's hjælpefunktion for at få "den fulde forklaring".

G.1 Matematiske funktioner

G.1.1 Numeriske funktioner

Funktion	Matematik	Betydning
`abs(x)`	$\|x\|$	numerisk (absolut) værdi af `x`
`sign(x)`		fortegn af `x` (-1, 0 eller 1)
`sqrt(x)`	$\sqrt{x}$	kvadratrod af `x`
`log(x)`	$\ln(x)$	naturlig logaritme af `x`
`log10(x)`	$\log(x)$	titalslogaritme af `x`
`exp(x)`	e^x	eksponentialfunktionen af `x`
`sin(x)`	$\sin(x)$	sinus til `x` radianer
`cos(x)`	$\cos(x)$	cosinus til `x` radianer
`tan(x)`	$\tan(x)$	tangens til `x` radianer
`asin(x)`	$\sin^{-1}(x)$	arcus sinus til `x`
`acos(x)`	$\cos^{-1}(x)$	arcus cosinus til `x`
`atan(x)`	$\tan^{-1}(x)$	arcus tangens til `x`
`factorial(x)`	$x!$	`x` fakultet
`floor(x)`	$\lfloor x \rfloor$	`x` rundet *ned* til nærmeste heltal
`ceiling(x)`	$\lceil x \rceil$	`x` rundet *op* til nærmeste heltal
`round(x)`	$[x]$	`x` afrundet til nærmeste heltal; halve rundes til lige tal
`Re(x)`	$\mathrm{Re}(x)$	Den reelle del af det komplekse tal `x`
`Im(x)`	$\mathrm{Im}(x)$	Den imaginære del af det komplekse tal `x`

G.1.2 Nulpunkter, ekstrema, integration

uniroot (afsnit 7)

Finder nulpunkt for en funktion af én variabel.

Eksempel på brug:

```
f <- function(x) { log(x+2*sin(x)) }
uniroot(f, c(0,1))
```

Kald og parametre:

uniroot(*funktion*, *interval* **)**

Parameter	Betydning
funktion	Funktionen, der skal findes et nulpunk for.
interval	Intervallet som nulpunktet skal findes inden for, givet som en vektor med to elementer **c(**x_1, x_2**)**. Funktionens fortegn skal være forskelligt i x_1 og x_2.

polyroot

Finder rødder i et polynomium i én variabel (både reelle og komplekse). Giver resultatet som en vektor af komplekse tal.

Eksempel på brug:

```
polyroot(c(2, 0, 4, -1))
```

Kald og parametre:

polyroot(*koefficienter* **)**

Parameter	Betydning
koefficienter	Vektor **c(**a_0, a_1, a_2, …, a_n**)** med koefficienterne i polynomiet $y = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$. Således findes i eksemplet ovenfor rødderne i $y = 2 + 4x^2 - x^3$

optimize (afsnit 8)

optimise (synonym)

Finder minimum eller maksimum for en funktion af én variabel.

Eksempel på brug:

```
f <- function(x) { log(x+2*sin(x)) }
optimize(f, c(2,6))
```

Kald og parametre:

optimize(*funktion*, *interval*, ...)

Parameter	Betydning
funktion	Funktionen, der skal findes et ekstremum for.
interval	Intervallet som ekstremumet skal findes inden for, givet som en vektor med to elementer **c**(x_1, x_2). Hvis ekstremum er i det ene intervalendepunkt giver **optimize** et resultat tæt på men ikke lig med endepunktet.
maximum=værdi	Om der skal søges et minimum (**maximum=FALSE**, standard) eller et maksimum (**maximum=TRUE**).

integrate (afsnit 9)

Numerisk integration af funktion af én variabel.

Eksempel på brug:

```
f <- function(x) { log(x+2*sin(x)) }
integrate(f, 1, 5)
```

Kald og parametre:

integrate(*funktion*, x_1, x_2, ...)

Parameter	Betydning
funktion	Funktionen, der skal integreres.
x_1, x_2	Intervallet der skal integreres for. Grænserne kan være i $\pm\infty$, men resultatet må ikke være uendeligt.
subdivisions=værdi	Det maksimale antal delintervaller der skal bruges. Standard er 100. For meget "takkede" funktioner bør dette sættes op.

G.1.3 Simulering

runif (afsnit 11)

Generering af ligefordelte tilfældige tal.

Eksempel på brug:

```
terning <- function(n) { floor(runif(n,1,7)) }
```

Kald og parametre:

runif(*antal*)

runif(*antal*, *min*, *max*)

Parameter	Betydning
antal	Antal tilfældige tal, der ønskes. Resultatet bliver en vektor af denne længde.
min, *max*	Intervallet, tallene skal være inden for. Standard er 0 og 1.

rnorm

(afsnit 11)

Generering af normalfordelte tilfældige tal.

Kald og parametre:

```
rnorm(antal)
rnorm(antal, μ, σ)
```

Parameter	Betydning
antal	Antal tilfældige tal, der ønskes. Resultatet bliver en vektor af denne længde.
μ	Middelværdi i normalfordelingen. Standard er 0.
σ	Spredning i normalfordelingen. Standard er 1.

replicate

(opgave 46 og 47)

Gentagen evaluering af et udtryk.

Eksempel på brug:

```
terning <- function(n) { floor(runif(n,1,7)) }
replicate(10, sum(terning(2)))
```

(Simulerer 10 terningslag der hver er summen af to terninger.)

Kald og parametre:

```
replicate(antal, udtryk)
```

Parameter	Betydning
antal	Antal gentagelser. Resultatet bliver en vektor af denne længde hvis *udtryk* giver et enkelt tal som resultat, men en matrix med *antal* søjler hvor hver søjle er et resultat ellers.
udtryk	Udtryk der skal udregnes gentagne gange. Giver kun mening hvis udtrykket kan give et forskelligt resultat fra gang til gang, for eksempel hvis der indgår tilfældige tal.

set.seed

Initialisering af R's tilfældighedsgenerator. Hvis man ønsker, at en simulering hvor tilfældige tal indgår skal give samme, reproducérbare resultat hver gang man kører den, må man sørge for at initialisere tilfældighedsgeneratoren først. Et givent "frø" (seed) giver den samme sekvens af tilfældige tal hver gang.

Eksempel på brug:

```
terning <- function(n) { floor(runif(n,1,7)) }
set.seed(21)
replicate(10, sum(terning(2)))
```

Kald og parametre:

```
set.seed(frø)
```

Parameter	Betydning
frø	Det nye "frø", et heltal.

G.2 Plot

G.2.1 Funktionsplot og XY-plot

`plot` (afsnit 5 og 12)

Tegner en funktionsgraf eller et XY-plot; påbegynder et nyt plot.

Eksempler på brug:

```
plot(f, 0, 5, col="red", ylim=c(0,80),
     xlab="t (min)", ylab="konc. (g/L)")
plot(g, 0, 5, col="blue", add=TRUE)

plot(x, y1, type="o", col="red", ylim=c(-5, 5), xlab="t", ylab="y")
points(x, y2, type="o", pch=2, col="green")
```

Kald og parametre:

`plot(`*funktion*`, `x_1`, `x_2`, ...)`* *(funktionsgraf)*

`plot(`*x-værdier*`, `*y-værdier*`, ...)`* *(XY-plot)*

Parameter	Betydning
Følgende parametre anvendes til funktionsgrafer:	
funktion	Funktionen, hvis graf skal tegnes.
x_1	Den nedre grænse for intervallet, funktionen skal tegnes for.
x_2	Den øvre grænse for intervallet, funktionen skal tegnes for.
`add=`*værdi*	Angiver om grafen skal starte et nyt plot (**`add=FALSE`**, standard) eller tilføjes til det aktuelle plot (**`add=TRUE`**).
`n=`*værdi*	Antal støttepunkter. Standard er 101.
Følgende parametre anvendes til XY-plot:	
x-værdier	En vektor med x-værdier.
y-værdier	En vektor med de tilsvarende y-værdier.
Følgende parametre anvendes uanset plottype og bestemmer det generelle layout af koordinatsystem med videre. De kan *ikke* anvendes når en funktionsgraf tilføjes til det aktuelle plot med **`add=TRUE`**.	
`ylim=c(`y_1**`,`**y_2**`)`**	Aksegrænserne på andenaksen. Som standard beregnes det fra funktionen eller datapunkterne der tegnes.
`xlim=c(`x_1**`,`**x_2**`)`**	Aksegrænserne på førsteaksen. Som standard tages det interval, funktionen tegnes for (funktionsgraf) eller hvor der er datapunkter (XY-plot).
`asp=`*værdi*	"Aspect ratio", dvs. skalering af enheder på andenaksen ift. førsteaksen. Angives **`asp=1`** er enheder på de to akser lige lange; angives fx **`asp=2`** er enheder på andenaksen dobbelt så lange som på førsteaksen. Som standard udnyttes plotområdet bedst muligt.
`log="`*akser***`"`**	Hvilke akser (om nogen) skal være logaritmiske. Angiv **`log="y"`** eller **`log="x"`** for logaritmisk og **`log="xy"`** for dobbeltlogaritmisk plot. Standard er **`log=""`** (ingen).
`axes=`*værdi*	Om der skal tegnes akser i plottet (**`axes=TRUE`**, standard) eller ikke (**`axes=FALSE`**). Styrer også kassen om plotområdet hvis ikke **`frame.plot`** angives, se nedenfor.
`frame.plot=`*værdi*	Om der skal tegnes kasse om plotområdet (**`frame.plot=TRUE`**) eller ikke (**`frame.plot=FALSE`**). Standard er værdien af **`axes`** parameteren.
`xlab=`*tekst*	Tekst, der skrives som titel på førsteaksen, for eksempel **`xlab="t"`**. Hvis man ikke angiver denne forsøger R selv at finde en fornuftig tekst. For helt at undgå titel på førsteaksen skrives **`xlab=""`**.
`ylab=`*tekst*	Som **`xlab`** men for andenaksen.

Se oversigtsskemaet for **`points`** for parametre der styrer udseende af datapunkter og linjer/kurver: **`type`**, **`col`**, **`lty`**, **`lwd`**, **`pch`**, **`bg`**, **`cex`**, *et cetera*.

Se skemaet for **`title`** på side 162 for øvrige parametre der styrer titler og tekster: **`col.lab`**, **`main`**, **`col.main`**, **`sub`**, **`col.sub`**, *et cetera*.

Se skemaet for **`axis`** på side 164 for parametre der styrer udseendet af akserne: **`las`**, **`fg`**, **`col.axis`**, *et cetera*.

points

(afsnit 12.1)

Tegner XY-punkter ind i det aktuelle plot.

Eksempel på brug:

```
plot(d$Tid, d$Kontrol, ylim=c(5,7), xlab="min", ylab="pH")
points(d$Tid, d$Vaekst, pch=4)
```

Kald og parametre:

points(*x-værdier*, *y-værdier*, ...)

Parameter	Betydning
x-værdier	En vektor med x-værdier.
y-værdier	En vektor med de tilsvarende y-værdier.
Alle følgende parametre kan også anvendes i **plot**:	
type=*graftype*	Angiver typen af graf, fx **"p"** eller **"l"**; se figur 31 på side 48 for en oversigt over typer. Man kan også angive typen **"n"** hvilket betyder at datapunkterne ikke vises — dette kan bruges til med **plot** at lave et "tomt" plot hvor plotområde og koordinatsystem mv. sættes op. Standard er **type="p"**.
col=*farve*	Farven symboler/linjer skal tegnes med. Standard er sort.
lty=*linjetype*	Linjetype. Mulige værdier er **"solid"** (standard), **"dashed"**, **"dotted"**, **"dotdash"**, **"longdash"** eller **"twodash"**.
lwd=*linjebredde*	Linjebredde. Standard er 1.
pch=*plotsymbol*	Symbol som datapunkter skal markeres med. Enten et tal fra 0 til 25 for et grafisk symbol (se figur 30 på side 48 for en oversigt) eller et tegn/bogstav/tal i gåseøjne, fx **pch="A"**. Specielt giver **pch="."** (punktum) en meget lille prik. Standard er **pch=1**.
bg=*farve*	"Fyldfarve" for de specielle symboler man kan vælge med **pch** 21 til 25.
cex=*værdi*	Skaleringsfaktor for symboler. Standard er 1.

lines

(afsnit 12.1)

Tegner linjestykker ind i det aktuelle plot.

Eksempel på brug:

```
plot(x, y1, type="l", col="red", ylim=c(-5, 5), xlab="t", ylab="y")
lines(x, y2, pch=2, col="green")
```

Kald og parametre:

lines(*x-værdier*, *y-værdier*, ...)

— synonym for: **points**(*x-værdier*, *y-værdier*, **type="l"**, ...)

Se skemaet for **points** på side 161 for mulige parametre.

abline

(afsnit 13)

Indtegn en ret linje, fx en regressionslinje, i det aktuelle plot.

Eksempler på brug:

```
plot(d$Tid, d$Kontrol, ylim=c(5,7), xlab="min", ylab="pH")
reg <- lm(Kontrol ~ Tid, data=d)
abline(reg, col="red")
```

```
plot(x, y)
abline(1, 1)
```

(Indtegner her linjen $y = x$ i plottet.)

Kald og parametre:

abline(*regr***)** *(regressionslinje)*

abline(a_0**,** a_1**)** *(given linje)*

Parameter	Betydning
regr	Et resultat fra **lm** (se skema side 178), dvs. resultatet af en lineær regression.
a_0, a_1	Alternativt kan man give koefficienterne a_0 og a_1 for linjen $y = a_0 + a_1 x$, dvs. skærings-punkt med y-aksen og hældning.

Se også oversigtsskemaet for **points** (side 161) for parametre der styrer udseendet af linjen: **col**, **lty** og **lwd**.

text

(afsnit 5.6)

Indsætter tekst i det aktuelle plot.

Eksempel på brug:

```
plot(sin, 0,10, ylab=) text(9.8, 0.2, "sin(x)", cex=1.5)
```

Kald og parametre:

text(*x***,** *y***,** *tekst***,** ...**)**

Parameter	Betydning
Alle parametre kan være vektorer, i hvilket tilfælde flere tekster indsættes på en gang.	
x, *y*	Placering af teksten.
tekst	Selve teksten, omsluttet af gåseøjne. Kan være et matematisk udtryk som $\sqrt{x}$ i stedet – Se R's hjælp for "plotmath" og/eller skriv **demo(plotmath)** i R Console.
col=*farve*	Farven teksten skal tegnes med. Standard er sort.
cex=*værdi*	Skaleringsfaktor for teksten. Standard er 1.
font=*værdi*	Kode for skrifttypen. Almindelig skrift er 1, **fed** er 2, *kursiv* er 3 og ***fed kursiv*** er 4. Standard er 1 (almindelig skrift).
adj=c(*b***,** *h***)**	Justering af teksten i forhold til referencepunktet (x, y). Se figur 13 på side 22.
srt=*værdi*	Rotation af teksten i forhold til vandret; en vinkel i grader. Standard er 0.

`title`

(afsnit 5.3)

Tilføjer overskrifter og/eller aksetekster til det aktuelle plot.

Eksempler på brug:

```
plot(sin,0,2*pi)
title(main="Sinus funktionen", sub="(vinkel i radianer)")
```

Kald og parametre:

```
title(...)
```

Parameter	Betydning
Standardværdi for alle tekster er `NULL` hvilket vil sige at `title` kun skriver de tekster, man eksplicit beder om.	
Følgende parametre kan alle også anvendes i `plot`:	
`xlab=`*tekst*	Tekst, der skrives som titel på førsteaksen, for eksempel `xlab="t"`.
`ylab=`*tekst*	Som `xlab` men for andenaksen.
`col.lab=`*farve*	Farve for aksetitlerne (`xlab` og `ylab`). Standard er sort.
`cex.lab=`*værdi*	Skalering af aksetitlerne (`xlab` og `ylab`). Standard er 1.
`font.lab=`*værdi*	Kode for skrifttypen for aksetitlerne (`xlab` og `ylab`). Almindelig skrift er 1, **fed** er 2, *kursiv* er 3 og ***fed kursiv*** er 4. Standard er 1 (almindelig skrift).
`main=`*tekst*	Hovedoverskrift for plottet, fx `main="Populationsudvikling"`.
`col.main=`*farve*	Farve for hovedoverskriften (`main`). Standard er sort.
`cex.main=`*værdi*	Skalering af hovedoverskriften (`main`). Standard er 1.2.
`font.main=`*værdi*	Som `font.lab` ovenfor men for hovedoverskriften (`main`). Standard er 2 (fed skrift).
`sub=`*tekst*	Underoverskrift for plottet (kommer neden for plotområdet), fx `sub="Kaniner"`.
`col.sub=`*farve*	Farve for underoverskriften (`sub`). Standard er sort.
`cex.sub=`*værdi*	Skalering af underoverskriften (`sub`). Standard er 1.
`font.sub=`*værdi*	Som `font.lab` med for underoverskriften (`sub`). Standard er 1 (almindelig skrift).
Følgende parameter kan *ikke* anvendes i `plot`:	
`line=`*værdi*	Placering af titlen/titlerne i forhold til plotområdets kant, angivet som antal tekstlinjehøjder "udad" fra den relevante kant – dvs. negative værdier betyder "indad". Standard er 3 for `xlab` og `ylab`, 1.7 for `main` og 4 for `sub`.

`mtext`

Tilføjer tekst i margen af det aktuelle plot.

Eksempler på brug:

```
plot(sin, 0, 2*pi, main="Sinus funktionen")
mtext("En periode", 1, line=2)
```

Kald og parametre:

```
mtext(tekst, side, ...)
```

Parameter	Betydning
tekst	Teksten.
side	Hvilken side/margen af plottet, teksten skal skrives i: 1 = for neden, 2 = venstre side, 3 = for oven (standard), 4 = højre side.
`col=`*farve*	Farve for teksten.
`cex=`*værdi*	Skalering af teksten. Standard er 1.
`font=`*værdi*	Skrifttypen. Almindelig skrift (standard) er 1, **fed** er 2, *kursiv* er 3 og ***fed kursiv*** er 4.
`line=`*værdi*	Placering af teksten i forhold til plotområdets kant, angivet som antal tekstlinjehøjder "udad" fra den relevante kant – dvs. negative værdier betyder "indad". Standard er 0.
`adj=`*værdi*	Justering af teksten i læseretningen, `adj=0` er venstrestillet og `adj=1` er højrestillet (værdier mindre end 0 eller større end 1 er også mulige). Standard er 0.5.
`padj=`*værdi*	Justering af teksten vinkelret på læseretningen, dvs. samme retning som `line` justerer men i andre enheder (i forhold til tekstens størrelse i stedet for standard "tekstlinjer"). Standard afhænger af `las` samt hvilken margen, der skrives i.
`las=`*værdi*	Tekstens retning i forhold til aksen for den margen, den skrives i. Enten 0 for parallelt med akserne (standard), 1 for tvungen vandret, 2 for vinkelret på akserne eller 3 for tvungen lodret. Lodrette tekster skrives altid fra neden og op.

axis

(afsnit 5.10)

Tilføjer en koordinatakse til det aktuelle plot.

Eksempler på brug:

```
plot(sin,0,2*pi, axes=FALSE)
axis(1, pos=0, at=c(0, pi, 2*pi), labels=c(NA,"pi","2 pi"))
axis(2, pos=0)
```

Kald og parametre:

```
axis( akse, ...)
```

Parameter	Betydning
akse	Hvilken akse der skal tegnes: 1 hhv. 3 er førsteaksen (x-aksen) for neden hhv. for oven i plottet; 2 hhv. 4 er andenaksen (y-aksen) til venstre hhv. til højre i plottet.
`pos=`*værdi*	Hvilken y- hhv. x-værdi aksen går igennem for x- hhv. y-aksen. Standard er placering i plottets kant.
`at=`*værdier*	En vektor af værdier hvor aksemærkerne skal sættes. Bemærk at aksen kun tegnes så langt som de yderste aksemærker. Som standard vælger R 5–7 jævnt fordelte mærker langs aksen.
`labels=`*værdier*	En vektor af tekster eller tal der skal bruges som akseetiketter. Der skal enten være lige så mange elementer som i `at`-vektoren eller også skal man angive `labels=NA` for "ingen etiketter". Standard er at bruge værdierne fra `at`.

Alle følgende parametre kan også anvendes i `plot`:

`tcl=`*værdi*	Længde af aksemærker, positiv for indad og negativ for udad. Standard er -0.5.
`fg=`*farve*	Farve for aksen (men ikke akseetiketterne, se `col.axis` nedenfor). Standard er sort.
`las=`*værdi*	Hvordan akse-etiketter skrives. Enten 0 for parallelt med akserne (standard), 1 for tvungen vandret, 2 for vinkelret på akserne eller 3 for tvungen lodret. Lodrette tekster skrives altid fra neden og op.
`col.axis=`*farve*	Farve for akse-etiketter. Standard er sort.
`cex.axis=`*værdi*	Skalering af akse-etiketter. Standard er 1.
`font.axis=`*værdi*	Kode for skrifttypen for akseetiketterne. Almindelig skrift er 1, **fed** er 2, *kursiv* er 3 og ***fed kursiv*** er 4. Standard er 1 (almindelig skrift).

box

(afsnit 5.10)

Tegner en kasse om det aktuelle plot.

Eksempel på brug:

```
plot(sin, 0, 2*pi, axes=FALSE)
box(fg="red")
```

Kald og parametre:

```
box(...)
```

Parameter	Betydning
`which=`*hvor*	Hvor kassen tegnes. Standard er `"plot"` hvilket giver den sædvanlige kasse om plotområdet. Andre muligheder er `"figure"`, der giver en kasse om hele figuren, og `"inner"` eller`"outer"`, der giver en kasse omkring hele "tegneområdet", evt. justeret for margener (tegneområdet kan indeholde mere end én figur).
`fg=`*farve*	Farve for kassen. Standard er sort.

Se også oversigtsskemaet for `points` (side 161) for øvrige parametre der styrer udseendet af linjen: `lty` og `lwd`.

legend

(afsnit 5.7 og 12.3)

Tegner en signaturforklaring i det aktuelle plot.

Eksempel på brug:

```
legend("topleft",
       c("Kontrol", "Behandling 1", "Behandling 2"),
       pch=1, col=c("red", "blue", "green"))
```

Kald og parametre:

`legend(`*placering*`,` *tekster*`, ...)`

Parameter	Betydning
placering	Hvor signaturforklaringen skal placeres i plottet. Enten koordinater *x*, *y* for øverste venstre hjørne eller en af teksterne `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"` eller `"center"`.
tekster	En vektor af tekster, én for hver signatur i signaturforklaringen, fx `c("Køer","Grise","Får")`.
For de følgende parametre `lty` til `fill` gælder at man kan angive en vektor af værdier, én for hver signatur, men at *genbrugsreglen* gælder hvis man angiver færre værdier end der er signaturer. Man kan altså nøjes med at angive én værdi for en egenskab der er den samme for alle signaturer, fx `lty="solid"` eller `pch=1`.	
`lty=c(...)`	Linjetypen hørende til hver signatur. For at udelade en linje anvendes typen `"blank"`.
`lwd=c(...)`	Linjebredden hørende til hver signatur (standard er 1).
`col=c(...)`	Linje/symbolfarven hørende til hver signatur (standard er sort).
`pch=c(...)`	Symbol hørende til hver signatur. For at udelade et symbol anvendes værdien `-1`.
`pt.bg=c(...)`	"Fyldfarve" for hver signatur (kun for `pch` 21 til 25).
`pt.cex=c(...)`	Symbolskaleringsfaktor hørende til hver signatur. Standard er værdien af `cex` (herunder).
`fill=c(...)`	Tegner et lille farvet rektangel for hver signatur, hvor `fill` værdierne angiver farverne.
Følgende parametre er knyttet til signaturforklaringen som et hele og ikke de enkelte signaturer:	
`title=`*tekst*	Titel til signaturforklaringen, fx `title="Symbolforklaring"`. Som standard er der ingen titel.
`cex=`*faktor*	Skaleringsfaktor for hele signaturforklaringen, fx `cex=1.5`. Standard er 1.
`title.cex=`*faktor*	Skaleringsfaktor for titlen, fx `tile.cex=1.5`. Standard er værdien af `cex`.
`text.col=`*farve*	Farve på teksterne i signaturforklaringen.
`title.col=`*farve*	Farve på titlen. Som standard samme som `text.col`.
`ncol=`*antal*	Antal søjler i signaturforklaringens layout. Standard er 1 (dvs. en "lodret" signaturforklaring). Hvis man her angiver antallet af signaturer får man en "vandret" signaturforklaring (men brug `horiz=TRUE` i stedet).
`horiz=`*værdi*	Angiv `horiz=TRUE` for at ignorere `ncol` og lave en "vandret" signaturforklaring. Standard er `horiz=FALSE`.
`inset=`*margen(er)*	Anvendes til at give en margen mellem plottets kant og signaturforklaringen når positionen angives med et af de specielle nøgleord (fx `"topleft"`). Hvis man angiver et enkelt tal, fx `inset=0.1`, bruges værdien både som vandret og lodret margen. Hvis man angiver en vektor med to tal, fx `inset=c(0.05,0.1)` angiver det første tal vandret margen og andet lodret margen. Tallene er andel af plottes bredde hhv. højde.
`bty=`*type*	Typen af ramme der tegnes om signaturforklaringen. Mulige værdier er `"o"` for en ramme (standard) eller `"n"` for ingen ramme.
`box.col=`*farve*	Rammens farve.
`bg=`*farve*	Man kan angive `bg` for at få en baggrundsfarve i signaturforklaringen, fx `bg="yellow"`, men kun hvis der også tegnes en ramme.

par

(afsnit 5.6)

Ændrer eller spørger om R's grafikparametre.

Med **par** kan man ændre standardværdierne for de fleste parametre i grafikfunktionerne (**plot**, **points**, **title**, ...) og/eller spørge hvad de aktuelle standardværdier er. Det vil dog som regel være bedre simpelthen at angive de nødvendige parametre i selve kaldene til grafikfunktionerne, og det er denne filosofi der er anvendt i disse noter. Nedenfor er derfor blot beskrevet nogle få ting man *kun* kan opnå med **par**.

Eksempel på brug:

```
opar <- par(mar=c(4.1,4.1,0,0))
plot(sin,0,2*pi)
par(opar)
```

Dette eksempel sætter mindre margen i bunden og fjerner margen i toppen og til højre i plottet for at opnå et større plotområde. Efter plottet er tegnet sættes margenstørrelserne tilbage til hvad de var før (hvilket er gemt i variablen **opar**).

Kald og parametre:

par("*navn*"**)** *(forespørg på parameter)*
par(*navn***=***værdi***)** *(ændring af parameter)*
par(*liste***)** *(ændring af flere parametre)*

Parameter	Betydning
navn	Navnet på parameteren der spørges på eller ændres. Se R's hjælp for de mange parametre. Der kan spørges på eller flere værdier ad gangen ved at angive flere navne, fx **par("lty","lwd")** eller **par(lty="dashed",lwd=3)**.
værdi	Ny værdi af parameteren. Bemærk at **par** ved ændring af parametre returnerer en (usynlig) liste med de gamle parameterværdier; denne kan man gemme i en variabel og så senere anvende med **par** for at genetablere de gamle værdier, jævnfør eksemplet ovenfor.
liste	En liste af parametre med deres nye værdier. Er som regel fremkommet som resultat fra et tidligere kald af **par** og bruges til retablering af gamle værdier, jævnfør eksemplet ovenfor.
Her følger nogle få af de mulige parametre der kan spørges om og/eller ændres på:	
"usr"	Giver koordinaterne for plotområdets hjørner i en vektor (x_1, x_2, y_1, y_2). Kan ikke ændres, kun spørges på.
mar=c(*b***,***v***,***t***,***h***)**	Margenerne omkring plotområdet (bund, venstre, top, højre), givet i antal "tekstlinjehøjder". Standard er **c(5.1,4.1,4.1,2.1)** hvilket giver plads til diverse tekster på alle sider af plotområdet. Kan med fordel reduceres hvis man ikke bruger disse margener til tekst.
bg=*farve*	Sætter en baggrundsfarve for hele plottet (tager effekt fra næste plot). Standard er **"transparent"** hvilket ikke tegner nogen baggrund.

locator

(afsnit 5.8)

Giver koordinator på punkter i et plot klikket med musen. Se **options** på side 189 for hvordan man slipper af med den irriterende bip-lyd.

Eksempel på brug:

```
plot(sin, 0, 2*pi)
locator(3)
```

Kald og parametre:

locator(*antal* **)**

Parameter	Betydning
antal	Antal punkter, der skal klikkes ind. Man har mulighed for at stoppe **locator** efter færre punkter ved at taste Esc, ved at højreklikke og vælge Stop (Windows) eller ved at ctrl-klikke (MacOS).

G.2.2 Farver

colors (afsnit 5.4)

colours (synonym)

Returnerer en tekstvektor med alle navnene der kan bruges til at specificere farver (fx `"red"`).

Eksempel på brug:

```
opar <- par(mar=c(0,0,0,0))
plot(0, 0, type="n", axes=FALSE, xlab=, ylab=)
legend("center", colors(), ncol=10, fill=colors(), cex=.5)
par(opar)
```

Ovenstående eksempel laver et plot med oversigt over alle de navngivne farver. Kræver, at man gør plotvinduet stort *inden* man kører linjerne.

Kald og parametre:

```
colors()
```

gray (afsnit 21.1.1)

grey (synonym)

Giver en gråtonefarve, dvs. en "farve" på en skala fra sort til hvid.

Eksempel på brug:

```
plot(sin, 0, 2*pi, col=gray(0.7))
```

Kald og parametre:

```
gray( værdi )
```

Parameter	Betydning
værdi	Et tal mellem 0 (sort) og 1 (hvid) der bestemmer hvor lys en grå man vil have. Hvis man giver en vektor får man en vektor af grå farver.

rgb (afsnit 21.1)

Specificerer en farve i RGB farverummet.

Eksempel på brug:

```
plot(sin, 0, 2*pi, col=rgb(1, .5, 0))
```

(Tegner en orange sinuskurve.)

Kald og parametre:

```
rgb(r, g, b)
```

Parameter	Betydning
r, g, b	Intensiteten af rød hhv. grøn og blå. Kan være vektorer for at give en vektor af farver.

rainbow (afsnit 21.1)

Giver en vektor af farver på en "regnbueskala".

Eksempel på brug:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
persp(x, y, z, col=rainbow(50))
```

Kald og parametre:

```
rainbow( antal )
```

Parameter	Betydning
antal	Hvor mange farver, der skal være i resultatvektoren.

heat.colors (afsnit 21.2)

Giver en vektor af farver på en "varmeskala".

Eksempel på brug:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
image(x, y, z, col=heat.colors(256))
```

Kald og parametre:

```
heat.colors( antal )
```

Parameter	Betydning
antal	Hvor mange farver, der skal være i resultatvektoren.

G.2.3 Eksport til grafikfiler

dev.off (afsnit 6.2)

Færdiggør og lukker den aktuelle grafikfil.

Eksempel på brug:

```
pdf("MitPlot.pdf", width=8, height=5)
plot(sin, 0, 2*pi, main="Sinus funktionen")
dev.off()
```

Kald og parametre:

```
dev.off()
```

pdf

(afsnit 6.2)

Åbner en grafikfil i PDF format. Efterfølgende plotfunktioner gemmer plottet i denne fil. Færdiggør og luk filen med `dev.off()`.

Eksempel på brug:

```
pdf("MitPlot.pdf", width=8, height=5)
plot(sin, 0, 2*pi, main="Sinus funktionen")
dev.off()
```

Kald og parametre:

`pdf(` *filnavn*`, ...)`

Parameter	Betydning
filnavn	Navnet på filen, plottet skal gemmes i. Relativt til det aktuelle katalog.
`width=`*bredde*	Bredden af grafikområdet, i amerikanske tommer (2,54 cm). Standard er 6.
`height=`*højde*	Højden af grafikområdet, i amerikanske tommer (2,54 cm). Standard er 6.
`paper=`*format*	Papirformatet. Standard er `"special"`, hvilket betyder at papirets størrelse er dikteret af `width` og `height` – det er som regel hvad man ønsker hvis plottet skal inkluderes i et dokument. Andre muligheder er `"a4"` og `"a4r"` for standard A4 hhv. "liggende" A4 og forskellige amerikanske formater `"letter"`, `"legal"`, `"executive"` og `"USr"` (sidstnævnte er "liggende" Letter-format). Bemærk at det at specificere papirformat ikke i sig selv ændrer størrelsen af grafikområdet – hvis man ønsker automatisk tilpasning af grafikområdet til hele papiret må man også angive `width=0` og `height=0`.

postscript

(afsnit 6.2)

Åbner en grafikfil i PostScript format. Efterfølgende plotfunktioner gemmer plottet i denne fil. Færdiggør og luk filen med `dev.off()`.

Eksempel på brug:

```
postscript("MitPlot.eps", width=8, height=5,
           paper="special", horizontal=FALSE)
plot(sin, 0, 2*pi, main="Sinus funktionen")
dev.off()
```

Kald og parametre:

`postscript(` *filnavn*`, ...)`

Parameter	Betydning
filnavn	Navnet på filen, plottet skal gemmes i. Relativt til det aktuelle katalog.
`width=`*bredde*	Bredden af grafikområdet, i amerikanske tommer (2,54 cm). Standard er at bruge hele papiret minus en lille margen.
`height=`*højde*	Højden af grafikområdet, i amerikanske tommer (2,54 cm). Standard er at bruge hele papiret minus en lille margen.
`paper=`*format*	Papirformatet. Standard er `"a4"` (for A4 papir), andre muligheder er forskellige amerikanske formater: `"letter"`, `"legal"` og `"executive"`. Man kan også angive `"special"`, hvis man samtidig specificerer bredde og højde med `width` og `height` (dette er sædvanligvis hvad man ønsker hvis plottet skal inkluderes i et dokument og ikke skrives ud direkte).
`horizontal=`*værdi*	Om plottet skal tegnes "liggende" på papiret (standard, `horizontal=TRUE`) eller ej (`horizontal=FALSE`). Hvis plottet skal inkluderes i et dokument kan det være en fordel at specificere `horizontal=FALSE` så man slipper for efterfølgende at skulle rotere det.

win.metafile

(afsnit 6.2)

Kun Windows. Åbner en grafikfil i Windows Metafile format. Efterfølgende plotfunktioner gemmer plottet i denne fil. Færdiggør og luk filen med `dev.off()`.

Eksempel på brug:

```
win.metafile("MitPlot.emf", width=8, height=5)
plot(sin, 0, 2*pi, main="Sinus funktionen")
dev.off()
```

Kald og parametre:

`win.metafile(`*filnavn*`, ...)`

Parameter	Betydning
filnavn	Navnet på filen, plottet skal gemmes i. Relativt til det aktuelle katalog. Hvis man angiver filnavnet `""` (altså et tomt navn) gemmes plottet på Windows' klippebord i stedet.
`width=`*bredde*	Bredden af grafikområdet, i amerikanske tommer (2,54 cm). Standard er 7.
`height=`*højde*	Højden af grafikområdet, i amerikanske tommer (2,54 cm). Standard er 7.

png

(afsnit 6.2)

Åbner en grafikfil i PNG format (et rasterbillede). Efterfølgende plotfunktioner gemmer plottet i denne fil. Færdiggør og luk filen med `dev.off()`.

Eksempel på brug:

```
png("MitPlot.png", width=800, height=600, bg="transparent")
plot(sin, 0, 2*pi, main="Sinus funktionen")
dev.off()
```

Kald og parametre:

`png(`*filnavn*`, ...)`

Parameter	Betydning
filnavn	Navnet på filen, plottet skal gemmes i. Relativt til det aktuelle katalog.
`width=`*bredde*	Bredden af grafikområdet, i pixels. Standard er 480.
`height=`*højde*	Højden af grafikområdet, i pixels. Standard er 480.
`bg=`*farve*	Baggrundsfarven, standard er hvid. Man kan angive `"transparent"` for transparent baggrund (nyttigt på websider).

dev.print

Kopierer det aktuelle plot til en grafikfil.

Eksempel på brug:

```
plot(sin, 0, 2*pi, main="Sinus og cosinus", ylab=)
plot(cos, 0, 2*pi, add=TRUE)
dev.print(pdf, "MitPlot.pdf", width=8, height=5)
```

Kald og parametre:

`dev.print(`*funktion*`,` *filnavn*`, ...)`

Parameter	Betydning
funktion	Den funktion, der skal bruges til at åbne grafikfilen, fx `pdf`, `postscript`, `win.metafile` eller `png`. Dette bestemmer formatet af filen samt hvilke parametre man kan angive ud over filnavnet.
filnavn	Navnet på filen, plottet skal gemmes i. Relativt til det aktuelle katalog.
...	Yderligere parametre videregives til *funktion* og man kan dermed specificere fx bredde, højde og papirformat.

G.2.4 Specielle plot

plot (af datasæt, afsnit 12)

Oversigtsplot for et datasæt.

Eksempel på brug:

```
d <- read.table("minedata.txt")
plot(d, main="Mine data")
```

Kald og parametre:

plot(*datasæt*, ...**)**

Parameter	Betydning
datasæt	Et datasæt. Alle variable (søjler) i datasættet plottes mod hinanden.

Mange parametre man sædvanligvis kan give til **plot** for at styre farver, tekststørrelser osv. kan også gives ved denne type plot.

hist (opgave 46 og 47)

Tegner et histogram.

Eksempel på brug:

```
terning <- function(n) { floor(runif(n,1,7)) }
slag <- replicate(1000, sum(terning(2)))
hist(slag, breaks=1.5:12.5)
```

Eksemplet tegner et histogram over udfaldet af 1000 (simulerede) slag med to terninger.

Kald og parametre:

hist(*datavektor*, ...**)**

Parameter	Betydning
datavektor	En vektor med de data (tal), der skal tegnes et histogram for.
breaks=*værdi*	En specifikation af hvilke eller hvor mange intervaller, data skal opdeles i. Som standard finder R nogle fornuftige intervaller, men hvis data er heltal og man ønsker en søjle per muligt tal kan man med fordel specificere det selv som i eksemplet ovenfor. Man angiver enten en vektor af intervalgrænser (som i eksemplet) eller et (cirka)antal for hvor mange søjler, man vil have.
border=*farve*	Farve for stregerne rundt om blokkene. Standard er sort.
col=*farve*	Fyldfarve for blokkene. Standard er ikke at fylde blokkene.
add=*værdi*	Om **hist** skal starte et nyt plot (**add=TRUE**, standard) eller tegne oven i det aktuelle plot (**add=FALSE**).

Mange af de sædvanlige parametre til **plot** kan også bruges til **hist**, specielt parametrene der bestemmer tekster og akser.

G.2.5 Grafer for funktioner af to variable

persp (afsnit 21.1)

Tegner et 3D overfladeplot.

Eksempel på brug:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
persp(x, y, z, phi=35, theta=30, ticktype="detailed")
```

Kald og parametre:

persp(*x*, *y*, *z*, ...)

Parameter	Betydning
x, *y*	To vektorer koordinator i x hhv. y retningen. Udspænder et "net" af støttepunkter.
z	En matrix med lige så mange rækker som der er elementer i x og lige så mange søjler som der er elementer i y, indeholdende z-værdier for alle støttepunkterne. Typisk fremkommet vha. **outer** som i eksemplet ovenfor.
r=*værdi*	Afstand mellem betragtningspunkt og plot; større værdier giver mindre perspektivisk fortegning. Standard er $\sqrt{3}$.
phi=*værdi*	Betragtningsvinklen, højde i grader over xy planen.
theta=*værdi*	Betragtningsvinklen, drejning om z-aksen i grader.
border=*farve*	Farven på "nettets" linjer. Standard er sort. Angiv **border=NA** for ikke at få tegnet nettet, kun udfyldt "facetterne".
col=*farve*	Farven på "facetterne". Standard er hvid. Angives en vektor med lige så mange elementer som der er facetter bestemmer man farven af den enkelte facet.
shade=*værdi*	Hvis **shade** angives tegnes overfladen som om den blev belyst af en lyskilde i retningen bestemt af **lphi** og **ltheta**. Angiv en værdi mellem 0 og 1, hvor 1 svarer til en punktlyskilde og 0 til diffust lys (samme som ikke at angive **shade**).
lphi=*værdi*	Lyskildens retning, højde i grader over xy planen (hvis man angiver **shade**).
ltheta=*værdi*	Lyskildens retning, drejning om z-aksen i grader (hvis man angiver **shade**).
scale=*værdi*	Om akserne skal skaleres frit (**scale=TRUE**, standard) eller om forholdet mellem enheder på akserne skal bevares (**scale=FALSE**).
expand=*værdi*	Skalering a z-aksen. Standard er 1.
ticktype=*værdi*	Angiver om der skal etiketter med enheder på akserne. Hvis man skriver **ticktype="detailed"** får man enheder på akserne, ellers får man bare pile der angiver retningen af akserne.
nticks=*værdi*	Cirka antal ønskede etiketter på hver akse. Har kun effekt sammen med **ticktype="detailed"**.
box=*værdi*	Om der skal tegnes en kasse (**box=TRUE**, standard) eller ej (**box=FALSE**).
xlab=*tekst*	Tekst på x-aksen.
ylab=*tekst*	Tekst på y-aksen.
zlab=*tekst*	Tekst på z-aksen.
xlim=c(x_1,x_2**)**	Interval for x-aksen. Standard er minimum og maksimum i x.
ylim=c(y_1,y_2**)**	Interval for y-aksen. Standard er minimum og maksimum i y.
zlim=c(z_1,z_2**)**	Interval for z-aksen. Standard er minimum og maksimum i z.
main=*tekst*	Hovedoverskrift.
sub=*tekst*	Underoverskrift.

Udseendet af diverse tekster kan styres med de sædvanlige parametre fra **title** og **axis** – eneste undtagelser er at **col.lab** og **col.axis** ikke lader til at virke.

contour

(afsnit 21.3)

Tegner et niveaukurveplot.

Eksempel på brug:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
contour(x, y, z, nlevels=5)
```

Kald og parametre:

`contour(`*x*`,` *y*`,` *z*`, ...)`

Parameter	Betydning
x, *y*	To vektorer koordinator i *x* hhv. *y* retningen. Udspænder et "net" af støttepunkter.
z	En matrix med lige så mange rækker som der er elementer i *x* og lige så mange søjler som der er elementer i *y*, indeholdende *z*-værdier for alle støttepunkterne. Typisk fremkommet vha. `outer` som i eksemplet ovenfor.
`nlevels=`*værdi*	Angiver *cirka* hvor mange niveauer, man vil have (standard er 10). R bestemmer ud fra dette nogle "pæne" niveauer.
`levels=`*niveauer*	Eksplicit angivelse af de niveauer, man ønsker (i en vektor).
`labels=`*etiketter*	Angivelse af etiketter, der skal skrives på niveaukurverne. Som standard skrives niveauernes *z*-værdier.
`drawlabels=`*værdi*	Om etiketter skal skrives på kurverne (`drawlabels=TRUE`, standard) eller ej (`drawlabels=FALSE`).
`method=`*værdi*	Hvordan etiketter skal indpasses på kurverne. Enten `"simple"` (i kanten, oven på konturerne), `"edge"` (i kanten "indsat" i konturerne) eller `"flattest"` (standard, i den fladeste del af konturerne uden at de overlapper).
`add=`*værdi*	Om `contour` skal starte et nyt plot (`add=TRUE`, standard) eller tegne oven i det aktuelle plot (`add=FALSE`).

Parametre til at bestemme hvordan linjerne tegnes er som for `points`; tekster og akser kan også styres med de sædvanlige parametre til `plot` (og `title`/`axis`).

image

(afsnit 21.2)

Tegner et *xy* "kort" hvor farven bestemmes af *z*-værdien i hvert punkt.

Eksempel på brug:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
image(x, y, z, col=heat.colors(256))
```

Kald og parametre:

`image(`*x*`,` *y*`,` *z*`, ...)`

Parameter	Betydning
x, *y*	To vektorer koordinator i *x* hhv. *y* retningen. Udspænder et "net" af støttepunkter.
z	En matrix med lige så mange rækker som der er elementer i *x* og lige så mange søjler som der er elementer i *y*, indeholdende *z*-værdier for alle støttepunkterne. Typisk fremkommet vha. `outer` som i eksemplet ovenfor.
`col=`*farver*	En vektor af farver der anvendes som "farveskala". Typisk bruges funktioner som `heat.colors` til at generere farverne, eller man kan opnå finere kontrol med `rgb`/`gray`.
`add=`*værdi*	Om `image` skal starte et nyt plot (`add=TRUE`, standard) eller tegne oven i det aktuelle plot (`add=FALSE`).

Parametre til at styre tekster og akser osv. er de sædvanlige fra `plot` (og `title`/`axis`).

G.3 Datasæt og statistik

G.3.1 Indlæsning/oprettelse af datasæt

`read.table` (afsnit 10.1)

Indlæser et datasæt fra en tekstfil.

Eksempel på brug:

```
data <- read.table("Markforsøg.txt", header=TRUE, dec=",")
summary(data)
```

Kald og parametre:

`read.table(`*filnavn*`, ...)`

Parameter	Betydning
filnavn	Navnet på tekstfilen, relativt til det aktuelle katalog (eller man kan angive den fuldstændige sti).
`header=`*værdi*	Angiv `header=TRUE` hvis filen indeholder en linje med kolonneoverskrifter. Ellers angiv `header=FALSE` (eller udelad `header`).
`dec=`*tegn*	Hvilket tegn der er decimalkomma i tal. Angiv `dec=","` for dansk decimalkomma. Standard er det engelske decimalpunktum `"."`.
`sep=`*tegn*	Hvilket tegn der adskiller kolonner, fx `sep=","` eller `sep=";"`. Standard er blanktekst (et eller flere mellemrum og/eller tabulatortegn).
`quote=`*tegn*	Hvilke(t) tegn der anvendes som anførselstegn om tekster. Standard er "`"`" (gåseøjne).
`na.strings=`*tekst(er)*	Hvilke(n) tekst(er) i filen der skal fortolkes som "manglende værdi" (NA). Fx `na.strings="."` hvis manglende værdier er markeret med punktum eller `na.strings=c(".", "-")` hvis både punktummer og "-" anvendes.
`as.is=`*kolonner*	Anføres hvis værdier i kolonner indeholdende tekst skal bevares som tekst og ikke konverteres til "faktorer". Enten `as.is=TRUE` for alle kolonner eller fx `as.is=c(3,5)` for kun tredje og femte kolonne.

`read.csv` (afsnit 10.1.2)

Indlæser et datasæt fra en tekstfil i amerikansk/engelsk kommasepareret format.

Eksempel på brug:

```
data <- read.csv("Experiment5.csv")
summary(data)
```

Kald og parametre:

`read.csv(`*filnavn*`, ...)`

— synonym for: `read.table(`*filnavn*`, sep=",", header=TRUE, ...)`

Se skemaet for `read.table` på side 174 for mulige parametre.

`read.csv2` (afsnit 10.1.2)

Indlæser et datasæt fra en tekstfil i dansk kommasepareret format.

Eksempel på brug:

```
data <- read.csv2("MitForsøg.csv")
summary(data)
```

Kald og parametre:

`read.csv2(`*filnavn*`, ...)`

— synonym for: `read.table(`*filnavn*`, sep=";", dec=",", header=TRUE, ...)`

Se skemaet for `read.table` på side 174 for mulige parametre.

`textConnection` (afsnit 10.4)

Læser fra en tegnstreng (tekst) som om den var indholdet af en tekstfil. Kan bruges til at indlejrer en tekstfil direkte i et R-script.

Eksempel på brug:

```
d <- read.table(header=TRUE, dec=",", textConnection(
"Tid Kontrol Vaekst
30 6,45 6,07
45 6,32 5,94
1440 5,43 5,34"))
```

Kald og parametre:

`textConnection(` *tekst* `)`

Parameter	Betydning
tekst	Teksten, der skal "læses" fra.

`data.frame` (afsnit 20.2)

Opretter et datasæt med angivne søjler og værdier.

Eksempel på brug:

```
d <- data.frame(Tid=c(5, 10, 15, 20, 30, 40, 50, 60),
                pH=c(5.3, 5.3, 5.4, 5.5, 5.7, 5.9, 6.3, 6.4))
```

Kald og parametre:

`data.frame(` *navn=værdier*`, ...)`

Parameter	Betydning
navn=værdier	Hver søjle oprettes med søjlenavn og en vektor af værdier. Vektorerne for de forskellige søjler skal være lige lange.

G.3.2 Eksport af data

`write.table` (afsnit 10.5)

Eksporterer et datasæt til en tabel.

Eksempel på brug:

```
write.table(d, "data.txt")
```

Kald og parametre:

`write.table(` *datasæt*`,` *filnavn*`, ...)`

Parameter	Betydning
datasæt	Det datasæt, der skal eksporteres.
filnavn	Navnet på tekstfilen, der skal skrives. Relativt til det aktuelle katalog. Man kan også angive `""` for at skrive data i R Console i stedet.

Uden andre parametre end ovenstående skrives en fil der kan indlæses direkte med `read.table` uden andre parametre end filnavnet. Nedenstående kan dog bruges til at justere formatet.

`sep="`*tegn*`"`	Separatortegn til at adskille søjler. Standard er mellemrum.
`dec="`*tegn*`"`	Decimalkommategn. Standard er punktum.
`na="`*tekst*`"`	Hvad der skal skrives for manglende værdier (NA). Standard er "NA".
`quote=`*værdi*	Om overskrifter og værdier, der er tekster, skal i gåseøjne (`quote=TRUE`, standard) eller ej (`quote=FALSE`).
`col.names=`*værdi*	Om søjleoverskrifter skal skrives (`col.names=TRUE`, standard) eller ej (`col.names=FALSE`). Alternativt en tekstvektor med søjleoverskrifter.
`row.names=`*værdi*	Om rækkenavne skal skrives (`row.names=TRUE`, standard) eller ej (`row.names=FALSE`). Alternativt en tekstvektor med rækkenavne.

write.csv (afsnit 10.5)

Eksporterer et datasæt i amerikansk/engelsk kommasepareret format.

Eksempel på brug:

```
write.csv(data, "Output.csv", row.names=FALSE)
```

Kald og parametre:

```
write.csv(datasæt, filnavn, ...)
```

— synonym for: `write.table(datasæt, filnavn, sep=",", dec=".", ...)`

Se skemaet for `write.table` på side 175 for mulige parametre.

write.csv2 (afsnit 10.5)

Eksporterer et datasæt i dansk kommasepareret format.

Eksempel på brug:

```
write.csv2(data, "MitForsøg.csv", row.names=FALSE)
```

Kald og parametre:

```
write.csv2(datasæt, filnavn, ...)
```

— synonym for: `write.table(datasæt, filnavn, sep=";", dec=",", ...)`

Se skemaet for `write.table` på side 175 for mulige parametre.

G.3.3 Søjler fra datasæt som variable

attach (afsnit 10.7)

Opretter kopier af søjlerne fra et datasæt som variable som man kan tilgå uden `$`-notation.

Eksempel på brug:

```
attach(d)
plot(Tid, Kontrol)
```

Kald og parametre:

```
attach(x)
```

Parameter	Betydning
x	Datasæt, hvis søjler man vil have adgang til.

detach (afsnit 10.7)

Fjerner variable tidligere tilføjet med `attach`.

Eksempel på brug:

```
detach(d)
```

Kald og parametre:

```
detach(x)
```

Parameter	Betydning
x	Datasæt tidligere tilføjet med `attach`.

G.3.4 Udtagning af deldatasæt

`head` (afsnit 10.6)

Udtager de første rækker af et datasæt eller en matrix, eller de første elementer af en vektor.

Eksempel på brug:

```
head(d, 10)
```

Kald og parametre:

```
head(x)
head(x, n)
```

Parameter	Betydning
x	Datasæt/matrix/vektor, der skal udtages fra.
n	Antal rækker eller elementer; standard er 6. Hvis man angiver et negativt tal $-n$ udtages alle rækker/elementer undtagen de sidste n.

`tail` (afsnit 10.6)

Udtager de sidste rækker af et datasæt eller en matrix, eller de sidste elementer af en vektor.

Eksempel på brug:

```
tail(d)
```

Kald og parametre:

```
tail(x)
tail(x, n)
```

Parameter	Betydning
x	Datasæt/matrix/vektor, der skal udtages fra.
n	Antal rækker eller elementer; standard er 6. Hvis man angiver et negativt tal $-n$ udtages alle rækker/elementer undtagen de første n.

`subset` (afsnit 20.3)

Udtager de rækker fra et datasæt, der opfylder en given betingelse.

Eksempel på brug:

```
subset(d, Kontrol>Vaekst)
```

Kald og parametre:

```
subset(datasæt, betingelse)
```

Parameter	Betydning
datasæt	Datasættet, der skal udtages en del af.
betingelse	En betingelse (et logisk udtryk) der skal være opfyldt (give `TRUE`) for en given række for at den udtages. Man kan anvende variabelnavne fra datasættet direkte uden $-notation.

G.3.5 Statistik og regression

Funktionerne i dette skema virker alle på såvel vektorer som matricer og datasæt:

Funktion	Betydning
`mean(v)`	Middelværdi (gennemsnit) af elementerne i `v`
`median(v)`	Medianen af elementerne i `v` (virker ikke på helt datasæt)
`sd(v)`	Standardafvigelse af elementerne i `v`
`var(v)`	Varians af elementerne i `v`
	Nedenstående funktioner kan alle kaldes med mere end én parameter, fx `max(x,y,z)`; i dette tilfælde virker de som hvis tallene fra alle parametrene var kombineret i én vektor `v`.
`max(v)`	Maksimum af elementerne i `v`
`min(v)`	Minimum af elementerne i `v`
`range(v)`	Svarer til `c(min(v), max(v))`
`sum(v)`	Summen af elementerne i `v`
`prod(v)`	Produktet af elementerne i `v`

`summary` (afsnit 10.6)

Giver en oversigt over et datasæt, (søjlerne i) en matrix eller en vektor. For søjler eller vektorer der indeholder tal giver det min, max, gennemsnit og kvartiler. For andre datatyper giver det andre former for oversigt.

Eksempel på brug:

```
data <- read.table("Markforsøg.txt", header=TRUE, dec=",")
summary(data)
```

Kald og parametre:

```
summary(x)
```

Parameter	Betydning
x	Datasæt, vektor eller matrix.

`lm` (afsnit 13)

Tilpasser en *lineær model* til data; dvs. foretager lineær regression.

Eksempel på brug:

```
plot(d$Tid, d$Kontrol, ylim=c(5,7), xlab="min", ylab="pH")
reg <- lm(Kontrol ~ Tid, data=d)
abline(reg, col="red")
```

Kald og parametre:

```
lm(formel, ...)
```

Parameter	Betydning
formel	R formel der angiver modellen – hvad der er den afhængige og de(n) uafhængige variabel/variable. Se afsnit 13 og eventuelt appendiks C.
`data=`*datasæt*	Et datasæt, variable skal tages fra. Gør at man kan bruge søjler fra dette datasæt direkte i formlen uden at skulle bruge $-notation.

`I`

(appendiks C)

En speciel funktion der bruges i R formler til at markere at et deludtryk skal fortolkes "som det er", specielt at operatorer som `*` og `^`, der ellers har speciel betydning i formler, har deres sædvanlige betydning som regneoperatorer.

Eksempel på brug:

```
preg <- lm(Kontrol ~ Tid + I(Tid^2), data=d)
```

Kald og parametre:

`I(`*udtryk*`)`

Parameter	Betydning
udtryk	Det deludtryk, der skal "beskyttes".

`predict`

(appendiks C)

Beregner værdier af den afhængige variabel i en model ud fra de(n) uafhængige variabel/variable. Kan fx. bruges til at beregne y-værdier på en regressionslinje.

Eksempel på brug:

```
reg <- lm(Kontrol ~ Tid, data=d)
predict(reg, data.frame(Tid=c(50,55,60)))
```

Kald og parametre:

`predict(`*model*`,` *data*`)`

Parameter	Betydning
model	En model; fx resultatet af et tidligere kald af `lm`.
data	Et datasæt der minimum indeholder variable (søjler) svarende til de(n) uafhængige variable/variable, der indgår i modellen. For en almindelig lineær regression er dette den variabel, der står til højre for `~` i formlen givet til `lm`. Datasættet kan enten være et almindeligt datasæt indlæst tidligere eller kan konstrueres til lejligheden med `data.frame` som i eksemplet ovenfor.

G.4 Vektorer og matricer

G.4.1 Oprettelse af vektorer og matricer

`c` (afsnit 14.1)

Konstruerer en vektor med specificerede elementer. Kan også sætte associationslister sammen.

Eksempel på brug:

```
c(7, 9, 13)
```

Kald og parametre:

```
c(...)
```

Parameter	Betydning
...	Alle argumenter kombineres til en vektor — med mindre et eller flere elementer er associationslister, i hvilket tilfælde resultatet bliver en associationsliste.

`rep` (afsnit 14.1)

Konstruerer en vektor med gentagne elementer.

Eksempel på brug:

```
tyve.ettaller <- rep(1, 20)
```

Kald og parametre:

```
rep(x,n)
rep(x, ...)
```

Parameter	Betydning
x	Element(er), der skal gentages. Kan være en vektor.
n	Antal gentagelser. Enten et enkelt tal, i hvilket tilfælde hele *x* gentages *n* gange, eller en vektor af samme længde som *x*, i hvilket tilfælde hvert element i *x* gentages det antal gange der står på den tilsvarende plads i *n*.
`each=`*m*	Angivelse af at elementerne i *x* skal deltages elementvis, hver *m* gange.
`len=`*længde*	Den ønskede længde af resultatvektoren. Fx vil `rep(c(1, 2, 3), len=5)` give vektoren (1,2,3,1,2).

`seq` (afsnit 14.1)

Giver en vektor med en regelmæssig sekvens af tal.

Eksempel på brug:

```
seq(0, 100, by=10)
```

Kald og parametre:

```
seq(fra, til, ...)
```

Parameter	Betydning
fra, *til*	De tal, sekvensen skal gå fra og til, altså de to tal i enderne af sekvensen. (Hvis trinlængden (`by`) ikke går op i intervallet vil sekvensen dog slutte med det sidste trin før *til* i stedet.)
`len=`*længde*	Antallet af elementer, der ønskes i resultatet. Beregnes som standard ud fra *fra*, *til* og `by`.
`by=`*trin*	Trinlængden, dvs. hvor meget sekvensen skal springe fra et tal til det næste (med fortegn). Standard er ± 1.
`along.with=`*vektor*	Der ønskes et resultat med samme antal elementer som i *vektor*.

`matrix` (afsnit 16.1)

Opretter en matrix.

Eksempel på brug:

```
M <- matrix(c(5.3, 4.19, 2.7, 1.3), 2)
```

Kald og parametre:

`matrix(`*vektor*`,` *rækker*`, ...)`

Parameter	Betydning
vektor	En vektor af tal, der skal indgå i matricen. Fylder som standard matricen søjlevis, se `byrow` nedenfor.
rækker	Antal rækker, matricen skal have.
`ncol=`*søjler*	Man kan angive antal søjler (og i dette tilfælde udelade antal rækker). Hvis både antal søjler og rækker angives, og der ikke er tilstrækkelig med elementer i den givne *vektor* til at fylde matricen, anvendes genbrugsreglen.
`byrow=`*værdi*	Angiv `byrow=TRUE` for at fylde matricen rækkevis i stedet for søjlevis.

`diag` (afsnit 16.1)

Opretter en diagonalmatrix (en matrix, hvor alle elementer uden for diagonalen er nul).

Eksempel på brug:

```
E <- diag(1, 3)
```

Kald og parametre:

`diag(`*vektor*`,` *rækker*`)`
`diag(`*rækker*`)`

Parameter	Betydning
vektor	En vektor af tal, der skal udgøre diagonalen i matricen. Standard er 1 (giver en enhedsmatrix).
rækker	Antal rækker (og søjler), matricen skal have. Hvis der ikke er tilstrækkelig med elementer i den givne *vektor* anvendes genbrugsreglen, så i eksemplet ovenfor dannes en 3×3 enhedsmatrix, hvilket også kunne være gjort med `diag(3)`.

`cbind` (afsnit 16.3)

Sætter vektorer/matricer sammen, søjlevis.

Eksempel på brug:

```
M <- cbind(M, v)
```

Kald og parametre:

`cbind(...)`

Parameter	Betydning
...	Vektorer og/eller matricer, der skal sættes sammen. Vektorer forlænges efter genbrugsreglen om nødvendigt.

`rbind` (afsnit 16.3)

Sætter vektorer/matricer sammen, rækkevis.

Eksempel på brug:

```
M <- rbind(x, y)
```

Kald og parametre:

`rbind(...)`

Parameter	Betydning
...	Vektorer og/eller matricer, der skal sættes sammen. Vektorer forlænges efter genbrugsreglen om nødvendigt.

outer

(afsnit 21)

Laver en "ydre produkt" matrix fra to vektorer.

Eksempel på brug:

```
f <- function(x,y) { cos(x)*sin(2*y) }
x <- seq(0, 2*pi, len=50)
y <- seq(-pi, pi, len=50)
z <- outer(x, y, f)
persp(x, y, z)
```

Kald og parametre:

`outer(`*x*`,` *y*`,`*funktion*`)`

Parameter	Betydning
x, *y*	To vektorer. Den resulterende matrix får en række for hvert element i x og en søjle for hvert element i y.
funktion	En funktion $f(x, y)$ af to variable. Værdien i række i, søjle j bliver $f(x_i, y_j)$. Hvis man ikke angiver nogen funktion bliver værdien i række i, søjle j til produktet $x_i y_j$.

data.matrix

Konverterer et datasæt til en matrix.

Eksempel på brug:

```
d <- read.table("matrix.txt")
M <- data.matrix(d)
```

Kald og parametre:

`data.matrix(`*datasæt*`)`

Parameter	Betydning
datasæt	Det datasæt, der skal konverteres til en matrix.

G.4.2 Element- eller række-/søjlevis anvendelse af funktioner

sapply

(afsnit 14.3)

Anvender en funktion elementvis på en vektor.

Eksempel på brug:

```
f <- function(x) { integrate(sin, 0, x*pi/100)$value }
sapply(0:100, f)
```

Kald og parametre:

`sapply(`*vektor*`,` *funktion*`,` ...`)`

Parameter	Betydning
vektor	En vektor, hvis elementer funktionen skal anvendes på.
funktion	En funktion, der skal anvendes elementvis på vektorens elementer. Dette vil typisk være en funktion som ikke i forvejen virker elementvis på vektorer, for eksempel fordi den kun kan tage et enkelt tal som argument (som i eksemplet ovenfor) eller fordi den virker på vektorer som et hele (for fx `sum` eller `c`). Resultatet af `sapply` vil blive en vektor med lige så mange elementer som den givne hvis funktionen giver et enkelt element (fx et tal) som resultat. Hvis funktionen derimod giver en vektor som resultat vil resultatet af `sapply` blive en matrix hvor resultaterne fra funktionen er sat sammen søjlevis.
...	Eventuelle ekstra argumenter til funktionen (hvis den tager mere end ét argument).

apply

Anvender en funktion søjle-, række- eller elementvis på en matrix. Kan også anvendes på datasæt.

Eksempel på brug:

```
apply(M, 2, sum)
```

(Beregner søjlesummer for `M`.)

Kald og parametre:

`apply(`*matrix*`,` *led*`,` *funktion*`, ...)`

Parameter	Betydning
matrix	En matrix, hvis søjler/rækker/elementer funktionen skal anvendes på.
led	Hvis man angiver 1 anvendes funktionen rækkevis, dvs. på matricens rækker efter tur, og resultaterne sættes sammen enten til en vektor, hvis de er enkelte tal, eller rækkevis til en ny matrix, hvis de er vektorer. Tilsvarende anvendes funktionen søjlevis hvis man angiver 2. Endelig kan man angive `c(1,2)` eller `1:2` hvilket betyder elementvis anvendelse, lige som `sapply` på vektorer. I eksemplet ovenfor er angivet 2 og derfor anvendes funktionen `sum` på hver søjle i `M` efter tur og resultatet vil blive en vektor med summerne af matricens søjler. Havde man angivet 1 i stedet for 2 ville man have fået summerne af rækkerne.
funktion	Den funktion, der skal anvendes.
...	Eventuelle ekstra argumenter til funktionen (hvis den tager mere end ét argument).

G.4.3 Dimensioner

length (afsnit 14.3)

Giver antal elementer i en vektor eller associationsliste.

Eksempel på brug:

```
v[ length(v) ]
```

(Giver sidste element i `v`.)

Kald og parametre:

`length(`*v*`)`

Parameter	Betydning
v	Vektoren (eller associationslisten) hvis antal elementer man ønsker.

dim (afsnit 16.7)

Giver dimensionerne af en matrix eller et datasæt som en vektor `c(`*n*`,`*m*`)` hvor *n* er antal rækker og *m* antal søjler.

Kald og parametre:

`dim(`*x*`)`

Parameter	Betydning
x	Matrix eller datasæt, dimensionerne ønskes for.

ncol (afsnit 16.7)

Giver antal søjler i en matrix eller et datasæt.

Kald og parametre:

`ncol(`*x*`)`

Parameter	Betydning
x	Matrix eller datasæt, antal søjler ønskes for.

nrow (afsnit 16.7)

Giver antal rækker i en matrix eller et datasæt.

Kald og parametre:

`nrow(`*x*`)`

Parameter	Betydning
x	Matrix eller datasæt, antal rækker ønskes for.

G.4.4 Navne

names (afsnit 14.5)

Giver eller ændrer navnene på vektorelementer.

Eksempler på brug:

```
names(x)

names(x) <- c("Et", "To", "Tre")

names(x)[3] <- "Nyt navn"
```

Kald og parametre:

`names(`*x*`)`
`names(`*x*`) <-` *navne*
`names(`*x*`)[`*indeksudtryk*`] <-` *navne*

Parameter	Betydning
x	Vektor.
navne	En vektor af navne, elementerne i *x* skal have. Angiv **NULL** for at fjerne alle navne fra *x*.
indeksudtryk	Angiver hvilke(t) element(er) i *x*, der skal have ændret navn (angives kun hvis ikke alle *x*'s elementer skal have ændret navn).

colnames (afsnit 16.8)

Giver eller ændrer søjlenavne for en matrix. Svarer fuldstændig i syntaks og brug til **names** (se skema ovenfor).

Eksempel på brug:

```
colnames(V) <- 1992:2002
```

Ovenstående ændrer søjlenavnene i en matrix **V** med 11 søjler til årstallene 1992 til 2002.

rownames (afsnit 16.8)

Giver eller ændrer rækkenavne for en matrix. Svarer fuldstændig i syntaks og brug til **names** (se skema ovenfor).

Eksempel på brug:

```
rownames(V) <- c("Klasse I", "Klasse II")
```

Ovenstående ændrer rækkenavnene i en matrix **V** med 2 rækker til teksterne "Klasse I" og "Klasse II".

G.4.5 Matrixalgebra

`t` (afsnit 16.1)

Transponerer en matrix (bytter om på rækker og søjler).

Kald og parametre:

```
t(M)
```

Parameter	Betydning
M	Matrix, der skal transponeres.

`det` (afsnit 16.5)

Udregner determinanten for en matrix.

Kald og parametre:

```
det(M)
```

Parameter	Betydning
M	Kvadratisk matrix, determinanten skal udregnes for.

`solve` (afsnit 16.5)

Inverterer en matrix eller finder løsning på en matrixligning $AX = B$.

Eksempler på brug:

```
Ainv <- solve(A)
loes <- Ainv %*% B

loes <- solve(A, B)
```

Kald og parametre:

```
solve(M)
solve(A, B)
```

Parameter	Betydning
M	Matrix, der skal inverteres. Resultatet er M^{-1}.
A, B	Matricen A og vektoren/matricen B i matrixligningen $AX = B$, der skal løses. Resultatet er X, dvs. $A^{-1}B$.

`eigen` (afsnit 16.6)

Giver egenværdier og egenvektorer for en matrix.

Kald og parametre:

```
eigen(M)
```

Parameter	Betydning
M	Matrix, hvis egenværdier og -vektorer ønskes.

G.5 Andre funktioner

G.5.1 Programmering

`list` (afsnit 20.1)

Skaber en associationsliste.

Eksempel på brug:

```
koordinater <- list(x=1:20, y=f(1:20))
```

Kald og parametre:

```
list(...)
```

Parameter	Betydning
...	Listens elementer angives enten som par af formen *navn=værdi* eller blot værdier uden tilknyttede navne, de enkelte elementer adskilt med kommaer.

`ifelse` (afsnit 19.3)

Sammensætter ud fra en vektor af sandhedsværdier en resultatvektor blandet af elementerne fra to vektorer, hvor sandhedsværdierne angiver hvilken af de to hvert element skal tages fra. Kan betragtes som et `if ... else` udtryk der virker elementvis på vektorer.

Eksempel på brug:

```
mindste <- ifelse(x<y, x, y)
```

(Giver en vektor hvor der på hver plads `mindste[`*i*`]` står det mindste af tallene `x[`*i*`]` og `y[`*i*`]`.)

Kald og parametre:

```
ifelse(s,x,y)
```

Parameter	Betydning
s	Vektor af sandhedsværdier.
x	Vektor med værdier, der bruges på de pladser hvor *s* er sand (`TRUE`).
y	Vektor med værdier, der bruges på de pladser hvor *s* er falsk (`FALSE`).

`invisible`

Gør en værdi "usynlig" i R Console. Anvendes hovedsagelig til at skjule returværdier fra funktioner hvor man sjældent ønsker at se returværdien men en gang imellem har brug for at gemme den eller regne videre på den.

Kald og parametre:

```
invisible(x)
```

Parameter	Betydning
x	Et udtryk eller en værdi, der ønskes gjort "usynlig". Værdien af `invisible(`*x*`)` er *x*, men `invisible` gør at værdien ikke vises i R Console.

`is.null`

Undersøger om en værdi er `NULL`. Man kan ikke i en betingelse tjekke om et objekt *x* er `NULL` med udtrykket *x*`==NULL`, man er nødt til at skrive `is.null(`*x*`)` i stedet.

Anvendes typisk i funktioner for at se om en formel parameter har fået en værdi (hvis man samtidig har specificeret parametrens standardværdi til at være `NULL`).

Kald og parametre:

```
is.null(x)
```

Parameter	Betydning
x	Et udtryk eller en værdi, der skal sammenligned med `NULL`.

paste

Sammensætter en tegnstreng af flere dele. Især nyttig til fx overskrifter i plot eller konstruktion af filnavne.

Eksempel på brug:

```
mit.plot <- function(a) {
     f <- function(x) { a*x^2 - 3*x + 2 }
     overskrift <- paste("Funktionen f med a=", a, sep="")
     plot(f, 0, 10, main=overskrift)
}
mit.plot(1)
mit.plot(5)
```

I ovenstående eksempel bruges **paste** til at sætte en parameterværdi ind i en plotoverskrift. Funktionen **mit.plot** laver en graf for funktionen $f(x) = ax^2 - 3x + 2$ for $x \in [0, 10]$, hvor a er en parameter. Overskriften (**main** i **plot**) indeholder den værdi af a, der er brugt. Kaldet **mit.plot(1)** giver et plot med overskriften "Funktionen f med a=1" mens **mit.plot(5)** giver et plot med overskriften "Funktionen f med a=5".

Kald og parametre:

paste(*dele*, ...**)**

Parameter	Betydning
dele	De dele (tekster og/eller tal) som skal sættes sammen til en samlet tekst. Hvis man giver vektorer får man en vektor som resultat, fx giver **paste(c("a","b"), 1:2)** resultatet **c("a 1","b 2")**.
sep=*tekst*	Tekst, der indsættes mellem de enkelte dele. Standard er **sep=" "** (et mellemrum), så hvis man vil have værdierne klistret "tæt" sammen som i eksemplet ovenfor må man angive **sep=""**.
collapse=*tekst*	Hvis man angiver en værdi for **collapse** bliver resultatet en enkelt tekst, også selvom delene er vektorer. Teksten angivet for **collapse** sættes ind mellem resultaterne som adskillelse, fx giver **paste(c("a","b"), 1:2, collapse=":")** resultatet **"a 1:b 2"**.

G.5.2 Interaktion

print

(afsnit 15.1)

Udskriver en værdi i R Console. Nyttig til fx at følge med i udførelsen af løkker under fejlfinding.

Eksempel på brug:

```
y <- 1
for (x in 2:6) { y <- y*x ; print(c(x,y)) }
```

Kald og parametre:

print(*værdi*, ...**)**

Parameter	Betydning
værdi	Den værdi, der skal udskrives. Formatet bestemmes af hvilken type værdi, det drejer sig om (vektor, matrix, datasæt...).
Nedenstående parametre respekteres ved de fleste typer værdier.	
digits=*værdi*	Antal betydende cifre i tal. En værdi mellem 1 og 22, omend mere end 15 betydende cifre som regel er meningsløst. Som standard anvendes det antal betydende cifre, der er valgt med **options** (som regel 7 betydende cifre, se skema side 189).
quote=*værdi*	Om tekster skal skrives med gåseøjne omkring (**quote=TRUE**, standard) eller ej (**quote=FALSE**).

cat

(afsnit 19.5.4)

Udskriver værdier i R Console i mere "råt" format end `print`. Kombinere i nogen grad funktionaliteten af `print` og `paste`. Nyttig til dialog med brugeren i R Console (sammen med `readline`).

Eksempel på brug:

```
cat("Resultatet af beregningen er:", 2+2, "\n")
```

Kald og parametre:

`cat(`*værdi(er)*`, ...)`

Parameter	Betydning
værdi(er)	En eller flere (tal og/eller tekst) værdier, der skal udskrives i R Console. Bemærk, at `cat` ikke laver linjeskift, så alle linjeskift der ønskes skal indsættes eksplicit med tegnet `"\n"`.
`sep=`*tekst*	Tekst, ser skal indsættes mellem de værdier, der udskrives. Standard er `sep=" "` (mellemrum), så for at få værdierne sat helt tæt sammen må man bruge `sep=""`.

flush.console

Sikrer at R Console er opdateret med seneste output fra `print` og `cat`. Af effektivitetshensyn går output til R Console gennem en buffer og `flush.console` synkroniserer selve R Console vinduet med bufferen. Kan være nødvendigt i interaktive R-scripts.

Kald og parametre:

```
flush.console()
```

readline

(afsnit 19.5.4)

Læser en linjes tekst indtastet i R Console. Venter på at brugeren indtaster noget tekst og taster ↵. Giver det indtastede som resultat. Bemærk at resultatet altid er en tekst, så hvis det skal tolkes som et tal må det efterfølgende konverteres med `as.numeric`.

Eksempel på brug:

```
indtastning <- readline("Indtast et tal: ")
tal <- as.numeric(indtastning)
```

Kald og parametre:

`readline(`*ledetekst*`)`

Parameter	Betydning
ledetekst	Denne tekst skrives i R Console før R giver sig til at vente på indtastningen. Kan udelades, hvilket svarer til at angive en tom tekst.

as.numeric

(afsnit 19.5.4)

Konverterer en værdi, der ikke er et tal, til et tal. Anvendes typisk til at lave tekstværdier (tegnstrenge) om til tal, altså fx lave teksten `"123"` om til tallet `123`.

Eksempel på brug:

```
indtastning <- readline("Indtast et tal: ")
tal <- as.numeric(indtastning)
```

Kald og parametre:

`as.numeric(`*x*`)`

Parameter	Betydning
x	De(n) værdi(er), der skal konverteres til et tal. Hvis *x* er en vektor får man en vektor af tal.

`file.choose`
Lader brugeren vælge en fil (i en sædvanlig fildialog) og returnerer navnet på filen.

Eksempel på brug:

```
data <- read.csv2( file.choose() )
summary(data)
```

Kald og parametre:

```
file.choose()
```

`Sys.sleep`
Indfører en pause i programudførelsen, for eksempel hvis man vil opbygge en graf trin for trin i en animation.

Kald og parametre:

```
Sys.sleep(x)
```

Parameter	Betydning
x	Antal sekunders pause (kommatal er muligt, så man kan angive brøkdele af sekunder).

G.5.3 Diverse

`source` (afsnit 10.4)
Kører et R-script direkte fra en fil.

Eksempel på brug:

```
source("mitscript.R")
```

Kald og parametre:

```
source(filnavn)
```

Parameter	Betydning
filnavn	Navnet på den `.R`-fil, der skal læses fra. Relativt til det aktuelle katalog.

options

(afsnit 2.1)

Ændrer/viser generelle indstillinger for R.

Eksempel på brug:

```
gemte.op <- options(digits=10)
exp(17)
options(gemte.op)
```

Kald og parametre:

```
options("navn")          (forespørg på indstilling)
options(navn=værdi)      (ændring af indstilling)
options(liste)           (ændring af flere indstillinger)
```

Parameter	Betydning
navn	Navnet på indstillingen der spørges på eller ændres. Se R's hjælp for de mange muligheder. Der kan spørges på eller flere indstillinger ad gangen ved at angive flere navne, fx `options("digits","locatorBell")` eller `options(digits=8, locatorBell=FALSE)`.
værdi	Ny værdi af indstillingen. Bemærk at `options` ved ændring af indstillinger returnerer en (usynlig) liste med de gamle værdier; denne kan man gemme i en variabel og så senere anvende med `options` for at genetablere de gamle indstillinger, jævnfør eksemplet ovenfor.
liste	En liste af indstillinger med deres nye værdier. Er som regel fremkommet som resultat fra et tidligere kald af `options` og bruges til retablering af gamle indstillinger, jævnfør eksemplet ovenfor.

Der er over 50 forskellige indstillinger der kan ændres og/eller vises med `options`. Herunder kun et par af dem.

`digits=`*antal*	Antal betydende cifre som tal vises med. Har ingen indflydelse på den interne regnenøjagtighed, kun visningen. Det er muligt at angive mellem 1 og 22, men over 15 er meningsløst på de fleste PC'er. Standard er 7.
`locatorBell=`*værdi*	Om `locator` skal komme med en bip-lyd hver gang man har udpeget et punkt (`locatorBell=TRUE`, standard) eller ikke (`locatorBell=FALSE`).

ls

(afsnit 3)

Giver en liste (tekstvektor) med navnene på alle variable, man har defineret.

Kald og parametre:

```
ls()
```

rm

(afsnit 3)

Sletter en eller flere variable.

Eksempler på brug:

```
rm("a","b")
```

(Fjerner variablene `a` og `b`.)

```
rm(list=ls())
```

(Fjerner alle variable, man har defineret.)

Kald og parametre:

```
rm(...)
```

Parameter	Betydning
...	Navne på de variable, der skal slettes, eventuelt med gåseøjne omkring.
`list=`*navne*	En tekstvektor med navne på variable, der skal slettes.

Indeks

Sidehenvisninger i *kursuv* er til skemaerne i oversigten over R-funktioner i appendiks G. Andre sidehenvisninger for R funktioner og parametre angiver hvor funktionerne forklares i hovedteksten.