Datorlaboration 6

Josef Wilzén och Måns Magnusson

5 maj 2025

Instruktioner

- Denna laboration ska göras i grupper om **två och två**. Det är viktigt för gruppindelningen att inte ändra grupper.
- En av ska vara **navigatör** och den andra **programmerar**. Navigatörens ansvar är att ha ett helhetsperspektiv över koden. Byt position var 30:e minut. **Båda** ska vara engagerade i koden.
- Det är tillåtet att diskutera med andra grupper, men att plagiera eller skriva kod åt varandra är **inte tillåtet**. Det är alltså **inte** tillåtet att titta på andra gruppers lösningar på inlämningsuppgifterna.
- Använd gärna Teams för att ställa frågor. Det finns olika kanaler:
  - **Questions**: Skriv era frågor här. Svar kommer att ges öppet direkt i kanalen. Publicera inte kod till inlämningsuppgifter här (andra kan då se det). Det går bra att skriva frågor om inlämningsuppgifter här så länge ni inte inkluderar kod med lösningar till dessa uppgifter. Det går bra att publicera kod till övningsuppgifter här.
  - **Raise_your_hand**: Skriv här om ni vill ha hjälp men inte ställa er fråga öppet. Skriv något i stil med "Jag vill ha hjälp". Då kommer en lärare att kontaka er när de har tid (i chatten på Teams). Vill flera ha hjälp så bildar de olika kommentarerna en kö, och hjälp kommer att ges i ordning efter kön. En "tumme upp" på kommentaren innebär att läraren har börjat hjälpa den aktuella studenten. Ett "hjärta" på kommentaren innebär att läraren har hjälpt klart studenten.
- Använd inte `å`, `ä` eller `ö` i variabel- eller funktionsnamn.
- Utgå från laborationsmallen, som går att ladda ned **här** (obs ny mall jämfört med tidigare veckor), när du gör inlämningsuppgifterna. Spara denna som `labb[no]_grupp[no].R` , t.ex. labb5_grupp01.R om det är laboration 5 och ni är grupp 01. Ta inte med hakparenteser i filnamnet. Denna fil ska **inte** innehålla något annat än de aktuella funktionerna, namn- och ID-variabler och ev. kommentarer. Alltså **inga** andra variabler, funktionsanrop för att testa inlämningsuppgifterna eller anrop till markmyassignment-funktioner.
- Precis innan inlämning på Lisam, döp om er R-fil till en **`.txt`** fil, detta görs för att kunna skicka in filen till Ouriginal för plagieringskontroll. Exempel: `labb5_grupp01.R` blir då `labb5_grupp01.txt` Ladda upp den filen (som slutar på .txt) på Lisam under rätt inlämning innan deadline.
- Laborationen består av två delar:
  - Datorlaborationen (= övningsuppgifter)
  - Inlämningsuppgifter
- I laborationen finns det extrauppgifter markerade med *. Dessa kan hoppas över.
- Deadline för laboration framgår på LISAM
- **Tips!** Använd "fusklapparna" som finns här. Dessa kommer ni också få ha med på tentan.

Innehåll

Del I

Datorlaboration

Kapitel 1

Introduktion till objektorienterad programmering

Objektorienterad programmering är ett programmeringsparadigm där data vävs ihop med programkod i objekt. Programkoden som är kopplad till ett särskilt objekt kallas metoder. Dessa metoder kan beskrivas som funktioner som bara fungerar för det aktuella objektet. Detta kan ställas mot ett procedurellt programmeringsparadigm där data och metoder inte vävs ihop på samma sätt.

När det gäller statistisk programmering är de vanligaste andra programmen ofta procedurella. Vi har ett datamaterial som vi sedan anropar funktioner för. I exempelvis SAS görs skillnad på datasteg (där data bearbetas) och procedurer (funktioner) som anropas för ett givet datamaterial. R skiljer sig från andra statistikprogram på grund av att det är (mer) objektorienterat.

I R är all data olika former av objekt. De olika objekten har i sin tur olika **klasser**. Objekten kan vara av klasser som `data.frame`, `function`, `numeric`, `matrix` o.s.v. För dessa olika klasser finns det sedan **generiska funktioner** där en och samma funktion gör olika beräkningar/saker beroende på vad det är för klass objektet har. Dessa typer av funktioner specialiserade för enskilda klasser kallas för **metoder**.

Det finns tre olika system för objektorientering i R. Det enklaste (och vanligaste) systemet för objektorientering kallas S3 och kan beskrivas som en lättviktsvariant av objektorienterad programmering. För en fördjupning i objektorienterad programmering i R (och de andra objektorienterade systemen i R) rekommenderas Advanced R av Hadley Wickham.

1.1 Klasser och objekt

I R:s system S3 används attributet `class()` för att både undersöka ett objekts klass och för att tillskriva ett objektet en egen klass.

1. Använd följande kod för att skapa objekt och undersöka dess klasser.

```
a <- c(1,2,5)
class(a)
b <- matrix(a)
class(b)
c <- data.frame(a)
class(c)
```

2. För att tillskriva ett objekt en given klass använder vi också `class()`. Nedan skapar jag en klass `student`.

```
stud_Kalle <- list()
class(stud_Kalle) <- "student"
str(stud_Kalle)
```

3. Ofta när vi skapar nya objekt vill vi ha konstruktorfunktioner, funktioner som skapar våra objekt. Exempel på detta är `data.frame()`, `matrix()` och `factor()`. Vill vi skapa en konstruktorfunktion för vår klass `student`.

```
student <- function(name, sex, grade){
        p <- list(name, sex, grade)
        class(p) <-"student"
        return(p)
}
kalle <- student("Kalle", "Man", "Pass")
class(kalle)

[1] "student"

print(kalle)

[[1]]
[1] "Kalle"

[[2]]
[1] "Man"

[[3]]
[1] "Pass"

attr(,"class")
[1] "student"
```

4. De olika delarna eller datat i klassen brukar kallas fields, eller fält. I klassen student ovan är `name`, `sex` och `grade` fält.

5. För att undersöka om ett objekt är av en specifik klass använder vi `inherits()`.

```
inherits(kalle, "student")

[1] TRUE
```

1.2 Generiska funktioner och metoder

För varje klass finns det (oftast) så kallade generiska funktioner. Eller funktioner som fungerar på olika sätt för olika klasser. Vi har redan stött på ett flertal sådana funktioner som `summary()`, `print()` och `plot()`. Den generiska funktionen anropar sedan specifika metoder - beroende på objektets klass.

1. För att undersöka om en funktion är en generisk funktion är det enklast att studera källkoden för funktionen. Vi kan exempelvis titta på funktionen `mean()`.

```
mean

function (x, ...)
UseMethod("mean")
<bytecode: 0x14affe218>
<environment: namespace:base>
```

2. Som framgår ovan är funktionen `mean()` en generisk funktion då det enda funktionen gör är att anropa metoden för den aktuella klassen. Vi kan se vilka metoder den generiska funktionen `mean()` har med `methods()`.

```
methods(mean)

[1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
see '?methods' for accessing help and source code
```

3. I fallet ovan ser vi att den generiska funktionen `mean()` kommer anropa olika metoder (funktioner) för olika klasser. Det som definierar en metod i R är att funktionsnamnet har följande struktur `[generiskt funktionsnamn].[klass]`. I R är dessa metoder i övrigt bara vanliga funktioner. De klasser som finns för `mean()` är olika klasser för tider och datum med undantag för klassen `default`. Metoden `mean.default` är den funktion som används om ingen metod finns för den specifika klassen (ex. en numerisk vektor).
4. Undersök för vilka klasser `print()` och `summary()` har metoder.
5. Vi kan också anropa dessa funktioner direkt om vi vill.

```
mean.default(1:3)

[1] 2
```

6. Undersök för vilka klasser `plot()` har metoder. Testa sen koden nedan.

```
data("trees")
class(trees$Girth)
plot(trees$Girth,trees$Volume)
class(trees)
plot(trees)
data("AirPassengers")
class(AirPassengers)
plot(AirPassengers)
plot.ts(AirPassengers)
plot.default(AirPassengers)
```

1.3 Skapa egna generiska funktioner och metoder

Som ett första steg om vi har skapat en egen klass kanske vi vill skapa metoder för vanliga generiska funktioner som `print()`.

1. Att skapa egna generiska funktioner görs på följande sätt. Först skapar vi den generiska funktionen.

```
min_gen <- function(x) UseMethod("min_gen")
```

2. Nästa steg blir att skapa en metod för respektive klass.

```
min_gen.student <- function(x) print("Min studentklass.")
min_gen(kalle)

[1] "Min studentklass."
```

3. Vi kan också lägga till en default-metod om vi vill som hanterar de situationer då funktionen inte anropas för vår studentklass.

```
min_gen.default <- function(x) print("En annan klass.")
min_gen(1:5)

[1] "En annan klass."
```

4. Detta gör att vi också kan använda andra generiska funktioner om vi definierar en metod för denna klass. Vill vi lägga till en egen metod till `print()` för vår klass student gör vi på följande sätt:

```
print.student <- function(x){
  cat("My name is ", x[[1]], ". I got a ", x[[3]], ".", sep="")
}
print(kalle)

My name is Kalle. I got a Pass.

print.student(kalle)

My name is Kalle. I got a Pass.

print.default(kalle)

[[1]]
[1] "Kalle"

[[2]]
[1] "Man"

[[3]]
[1] "Pass"

attr(,"class")
[1] "student"
```

1.4 * Extraproblem

1. Nu ska en S3 klass skapas som du kallar `account` och som har fälten `changes` och `owner`. Fältet `changes` ska vara en data.frame med variablerna `time` och `amount`. Fältet `owner` ska vara en character-vektor av längd 1. Syftet är att skapa en klass som representerar för en persons bankkonto.

   (a) Skapa nu en konstruktor för klassen `account`, som heter `account(changes=,owner=)` och som testar att villkoren i 1 är uppfyllda. Testa sedan att skapa ett objekt av klassen `account:`

   ```
   x<-account(changes=data.frame(time="20:01",amount=1000),owner="Elin")
   x
   class(x)
   ```

   (b) Skapa nu två generiska funktioner `deposit()` och `withdraw()` som lägger till information om uttag och insättning i `changes`. Funktionen `deposit()` ska lägga till ett positivt numeriskt värde (insättning) i `amount` och `withdraw()` ska lägga till ett negativt värde (uttag). När `deposit()` eller `withdraw()` används ska också tidpunkten för detta sparas. [**Tips!** `Sys.time()`]

(c) Korrigera nu din metod `withdraw()`. Det ska bara vara tillåtet att göra ett uttag om det redan finns pengar på kontot, d.v.s. kontot får aldrig som helhet vara negativt.

Kapitel 2

Grundläggande linjär algebra

R har en hel del funktioner för att arbeta med matriser. Det som skiljer matriser från data.frames i R är att matriser endast kan ha en atomär klass/variabeltyp, d.v.s. logiska matriser, numeriska matriser och textmatriser. I denna del kommer vi att fokusera på numeriska matrsier och klassisk linjär algebra i R.

2.1 Skapa matriser

Följande funktioner är av intresse för att skapa matriser.

1. För att skapa en numerisk matris använder vi `matrix()`. Där vi kan ange data och matrisens dimensioner.

```
A <- matrix(data=1:20,nrow=4,ncol=5)
A

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

2. Tänk på att om vi indexerar en rad eller column i matrisen reduceras detta till en vektor i R, d.v.s. vi vet inte om det är en rad- eller kolumnvektor. Pröva följande kod.

```
X<-matrix(1:20,4,5)
X[,1]
X[,1,drop=FALSE]
```

3. Ibland vill vi snabbt kunna skapa diagonalmatriser. För detta används funktionen `diag()`

```
A <- diag(1:3)
A
```

Funktion	i R
Skapa matris	`matrix()`
Skapa diagonal/enhetsmatris	`diag()`
Triangulära matriser	`upper.tri(), lower.tri()`

Tabell 2.1: Skapa matriser i R

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

4. På ett liknande sätt kan vi skapa en godtycklig enhetsmatris med `diag()` på följande sätt.

```
A <- diag(4)
A

     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

5. Har vi redan en matris kan vi använda `diag()` för att plocka ut diagonalelementen från matrisen.

```
A <- matrix(1:16, ncol=4)
diag(A)

[1]  1  6 11 16
```

6. Använd funktionen `diag()` för att:

   (a) Skapa en enhetsmatris av storlek 12.

   (b) Skapa en diagonalmatris som har värdena $2, 3, 5, 7, 1, 2$ på diagonalen.

7. Ibland vill vi skapa en över eller undertriangulär matris. För detta kan vi använda funktionerna `upper.tri()` eller `lower.tri()`. Dessa funktioner skapar en logisk matris som kan användas för att indexera de triangulära elementen.

```
A <- matrix(0, ncol=3, nrow=3)
A[upper.tri(A, diag = FALSE)] <- c(1,2,3)
A

     [,1] [,2] [,3]
[1,]    0    1    2
[2,]    0    0    3
[3,]    0    0    0
```

8. Använd trianguleringsfunktionerna för att skapa följande matris.

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    2    4    0
[3,]    3    5    6
```

2.1.1 Blockmatriser

Vi kan även självklart arbeta med blockmatriser för att skapa större matriser. Exempel på blockmatriser är

$$\mathbf{A} = \begin{pmatrix} \mathbf{B} \\ \mathbf{C} \end{pmatrix}, \mathbf{A} = \begin{pmatrix} \mathbf{C} & \mathbf{B} \end{pmatrix} \text{ och } \mathbf{A} = \begin{pmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{F} \end{pmatrix}$$

där **B**, **C**, **D** och **F** är matriser med lämpliga dimensioner.

1. För att sätta samman två matriser kolumnvis används `cbind()`.

```
A <- diag(3)
B <- matrix(1:9, ncol=3)
cbind(A, B)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    1    4    7
[2,]    0    1    0    2    5    8
[3,]    0    0    1    3    6    9
```

2. För att sätta samman två matriser kolumnvis används `rbind()`.

```
rbind(A, B)
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
[4,]    1    4    7
[5,]    2    5    8
[6,]    3    6    9
```

3. Skapa följande matris genom att använda blockmatriser.

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    2    1    0    0    0    0
[3,]    3    2    1    0    0    0
[4,]    0    0    0    2    3    4
[5,]    0    0    0    0    3    4
[6,]    0    0    0    0    0    4
```

2.2 Matrisalgebra

De flesta matrisoperationer finns redan installerat i R från början.

1. Addition och subtraktion sker elementvis.

```
A <- matrix(c(1,1,1,1,1,2,1,3,4),ncol=3)
B <- matrix(c(-1,2,2,-2,-2,1,1,1,-1),ncol=3)
A + B
```

```
     [,1] [,2] [,3]
[1,]    0   -1    2
[2,]    3   -1    4
[3,]    3    3    3

B - A

     [,1] [,2] [,3]
[1,]   -2   -3    0
[2,]    1   -3   -2
[3,]    1   -1   -5
```

2. Matrismultiplikation görs med `%*%`.

```
A %*% B

     [,1] [,2] [,3]
[1,]    3   -3    1
[2,]    7   -1   -1
[3,]   11   -2   -1
```

3. Vill vi transponera vår matris använder vi `t()`.

```
t(B)

     [,1] [,2] [,3]
[1,]   -1    2    2
[2,]   -2   -2    1
[3,]    1    1   -1
```

4. Matrisinversen för en kvadratisk matris B betecknas som B^{-1}, och definars som

$$BB^{-1} = E$$

där E är en enhetsmatris av lämplig storlek. För att beräkna inversen av en matris används `solve()`.

```
B

     [,1] [,2] [,3]
[1,]   -1   -2    1
[2,]    2   -2    1
[3,]    2    1   -1

solve(B)

          [,1]    [,2] [,3]
[1,] -0.33333 0.33333    0
[2,] -1.33333 0.33333   -1
[3,] -2.00000 1.00000   -2

E<-B%*%solve(B)
print(E)
```

```
          [,1] [,2] [,3]
[1,] 1.0000e+00    0    0
[2,] 0.0000e+00    1    0
[3,] 2.2204e-16    0    1

# när vi gör numeriska beräkningar uppstår ofta små beräkningsfel:
round(E,8)

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

5. **Notera**: När vi gör numeriska beräkningar i programmering (t.ex. numerisk linjär algebra), så uppstår det ofta små numeriska fel. Detta kan göra att värden som ”ska vara 0” inte är det, utan är mycket små tal istället. Oftast spelar det inte någon roll praktiskt, om vi avrundar vårt resultat till t.ex. åtta decimaler så blir det ofta 0, vilket är tillräckligt bra i flesta situationer. Dock så kan vi inte alltid testa om en variabel är noll med `x==0`, då `x` kanske är $-3.342 \cdot 10^{-13}$. Då får vi göra test av typen: `abs(x)<=1e-8`. Testa koden nedan:

```
?all.equal
set.seed(35)
G<-matrix(rnorm(n = 20),2,10)
H<-G+1e-6
all.equal(G,H)
all(abs(G-H)<=1e-5)
all(abs(G-H)<=1e-6)

H<-G+1e-7
all.equal(G,H)
all(abs(G-H)<=1e-7)

H<-G+1e-8
all.equal(G,H)
all(abs(G-H)<=1e-8)
```

6. `solve()` kan även användas för att lösa linjära ekvationssystem. Testa `?solve()`. Säg att vi vill lösa följande ekvationssystem:

$$\begin{aligned} -x_1 - 2x_2 + x_3 &= 1 \\ 2x_1 - 2x_2 + x_3 &= 2 \\ 2x_1 - x_2 - x_3 &= 3 \end{aligned}$$

Eller uttryck med matriser:

$$B = \begin{pmatrix} -1 & -2 & 1 \\ 2 & -2 & 1 \\ 2 & 1 & -1 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

$$Bx = b$$

För att räkna ut vilket värde som x har kan solve användas enligt nedan:

```
print(B)

     [,1] [,2] [,3]
[1,]   -1   -2    1
[2,]    2   -2    1
[3,]    2    1   -1
```

```r
b<-c(1,2,3)
print(b)

[1] 1 2 3

x<-solve(a=B,b=b)
print(x)

[1]  0.33333 -3.66667 -6.00000
```

7. Testa att ändra några värden i matrisen B och vektorn b, och lös det nya ekvationssystemet. Notera dock att vissa ekvationssystem saknar unika lösningar.
8. Använd matriserna $\mathbf{A}$ och $\mathbf{B}$ ovan. Skapa även följande matriser $\mathbf{C}$ och $\mathbf{D}$.

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \mathbf{D} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

9. Beräkna följande blockmatris i R

$$\mathbf{X} = \begin{bmatrix} (\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & -(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & \mathbf{D}^{-1} + \mathbf{D}^{-1}\mathbf{C}(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}\mathbf{B}\mathbf{D}^{-1} \end{bmatrix}$$

10. Beräkna följande blockmatris i R

$$\mathbf{Y} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1}$$

11. $\mathbf{X}$ och $\mathbf{Y}$ ska vara identiska. Detta är ett sätt att invertera matriser på ett enklare sätt genom att invertera delar av matrisen. Testa om $\mathbf{X}$ och $\mathbf{Y}$ är identiska med koden nedan.

```r
tol_val<-1e-8
all(abs(X-Y)<=tol_val)

all.equal(X,Y)
```

2.2.1 Matrisegenskaper

1. Vill vi ta reda på en matris dimensioner använder vi `dim()`. Då returneras matrisens dimensioner som en integervektor av längd 2.

```r
dim(A)

[1] 3 3
```

2. Vill vi beräkna determinanten för en given matris använder vi `det()`.

```r
det(A)

[1] -2
```

3. Beräkna följande determinanter.

$$\det\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \det\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}, \det(\mathbf{I}_5)$$

där $\mathbf{I}_5$ är identitetsmatrisen av storlek 5.

2.3 Egenvärden och egenvektorer

I R används funktionen `eigen()` för att beräkna både egenvärden och egenvektorer.

1. Med följande kod kan vi beräkna egenvärdena för följande.

```
A <- matrix(c(3,-2,2,-2),ncol=2)
egen <- eigen(A)
egen

eigen() decomposition
$values
[1]  2 -1

$vectors
         [,1]     [,2]
[1,]  0.89443 -0.44721
[2,] -0.44721  0.89443
```

2. Funktionen `eigen()` returnerar en lista med egenvärdena (i fallande ordning) och egenvektorerna för respektive egenvärde som kolumner i matrisen med listnamnet `vectors`.

```
egen$values

[1]  2 -1

egen$vectors

         [,1]     [,2]
[1,]  0.89443 -0.44721
[2,] -0.44721  0.89443
```

3. För matrisen A ovan, kontrollera att definitionen för egenvärden och egenvektorer stämmer. D.v.s.

$$\mathbf{Ax} = \lambda\mathbf{x}$$

där λ är ett av egenvärdena och $\mathbf{x}$ är egenvärdets egenvektor. Kontrollera på detta sätt båda egenvärdena.

2.4 Paketet Matrix

Paketet `Matrix` är ett paket som används för att utföra numerisk linjär algebra. Paketet innehållet många specialfunktioner som relaterar till matriser och är *snabbare* än grundfunktionerna i R för linjär algebra.

1. Ladda in paketet i din session.
2. Testa att köra koden nedan.

```
?Matrix
a<-Matrix(1:10,5,2)
a
Matrix(1:10)
```

3. Matriser från `Matrix()` är av en andra klasser jämfört med matriser skapade med `matrix()`. Olika metoder är implemnterade för de olika klasserna. Därför är det viktigt att kunna konvertera mellan klasserna, beroende på sammanhanget. Testa koden nedan.

```
b<-matrix(11:20,5,2)
class(a)
class(b)
str(a)
str(b)
a2<-as.matrix(a)
b2<-Matrix(b)
class(a2)
class(b2)
```

4. De flesta metoderna för linjär algebra finns implementerade för klasserna i paketet `Matrix`. Skapa nu några matriser med funktionen `Matrix()`. Testa sen några av de vanliga matrisfunktionerna från 2.1 till 2.3 på dessa matriser.

2.5 * Extraproblem

1. Skapa ett linjärt ekvationssystem med fyra och sex ekvationer och lös dem sedan med `solve()`.
2. Skapa matrisen `A`,`B` och `C` nedan.

```
A<-matrix(1:25,5,5)
B<-matrix(11:25,5,3)
C<-matrix(c(5,2,1,3,4,5,-2,-2,1),ncol=3)
```

3. Gör följande beräkningar.
   (a) $\mathbf{A}^\intercal\mathbf{A}$
   (b) $\mathbf{B}\mathbf{B}^\intercal\mathbf{A}$
   (c) $(\mathbf{B}^\intercal\mathbf{B})^{-1}$
   (d) $\mathbf{ABC}$
4. Funktionen `generate_matrix()` nedan skapar slumpmässiga kvadratiska matriser med hjälp av `sample()`. Skapa och kör `generate_matrix()` så att den finns tillgänglig i din workspace.

```
generate_matrix<-function(mat_dim=5, numbers=10, seed=12345){
        set.seed(seed)
        my_size<-mat_dim^2
        temp <- sample(x=numbers, size=my_size,replace=TRUE)
        mat<-matrix(temp,mat_dim, mat_dim)
        return(mat)
}
```

5. Kör koden nedan. Vad innebär resultatet från funktionen `kappa()`? [**Tips** `?kappa`]

```
A<-generate_matrix(mat_dim=10,numbers=-10:10,seed=398)
B<-generate_matrix(mat_dim=100,numbers=-10:10,seed=872)
C<-generate_matrix(mat_dim=1000,numbers=-10:10,seed=812)
dim(A)
```

```
dim(B)
dim(C)
kappa(A)
kappa(B)
kappa(C)
Ainv<-solve(A)
Binv<-solve(B)
Cinv<-solve(C)
det(A)
det(B)
det(C)
```

6. Hur påverkar κ beräkningarna av matrisinverser?

Kapitel 3

Tid och datum med `lubridate`

Att arbeta med datum och tid i R innebär att vi behöver arbeta i två steg. Först behöver vi läsa in datumet i ett korrekt datumformat med paketet `lubridate` och sedan kan vi använda det för beräkningar.

3.1 Läsa in datum med `lubridate`

Det finns ett antal inläsningsfunktioner för att konvertera textvektorer till datumvektorer.

Ordning i textvektor	Inläsningfunktion
year, month, day	`ymd()`
year, day, month	`ydm()`
month, day, year	`mdy()`
day, month, year	`dmy()`
hour, minute	`hm()`
hour, minute, second	`hms()`
year, month, day, hour, minute, second	`ymd_hms()`

Källa: Grolemund and Wickham (2011) Dates and time made easy with lubridate

1. Ladda in paktet `lubridate` i den aktuella R-sessionen. [**Tips!** `library()`]
2. För att läsa in datum kan vi exempelvis göra på följande sätt:

```
library(lubridate)


Attaching package: 'lubridate'
The following objects are masked from 'package:base':
   date, intersect, setdiff, union

ymd("2012-10-10")

[1] "2012-10-10"
```

3. Konvertera ditt födelsedatum som ett datum i R (kalla variabeln `birth`), pröva med `ymd()` och `mdy()`.
4. Pröva funktionen `now()` och `today()`. Vad gör de?
5. Skapa en textvektor med minst 3 textelement med godtyckliga datum. Konvertera dessa till R med en av inläsningfunktionerna ovan.

6. Vill vi plocka ut en viss information ur ett datum kan vi göra det med följande funktioner: `year()`, `month()`, `week()`, `yday()`, `mday()` och `wday()`. Pröva dessa funktioner på din födelsedag. Vad får du för resultat av respektive funktion?

7. Dessa funktioner kan också användas för att ändra datumvariabler.

```
birth <- ymd("2005-03-22")
month(birth) <- 1
wday(birth) <- 1
```

8. Vad innebär förändringarna ovan med avseende på din födelsedag?

3.2 Räkna med datum

3.2.1 Intervall

När vi arbetar med datum finns det tre former av utsträckning i tid att hålla reda på. Först har vi tidpunkter (instants). Det är punkter i tiden, exempelvis ett datum, sedan har vi tidsintervall (intervals) duration (duration) och period (period). Intervallen är datumintervallen mellan två tidpunkter. För att skapa ett intervall gör vi på följande sätt i R:

```
date1 <- ymd("2012-10-10")
date2 <- ymd("2014-11-03")
myInterval <- interval(start=date1, end=date2)
```

1. Skapa ett intervall-objekt `myTime` som börjar vid din födelsedag och slutar idag.

2. Använd den slumpmässiga vektor med datum du skapat ovan och skapa en vektor med tidsintervall.

3.2.2 Duration

Duration och period är istället för intervall definierade som en tidperiod utan tydliga tidpunkter. Om vi mäter en period i sekunder får vi ett sätt att mäta perioder som är oberoende av vilka datum vi talar om. Detta är definitionen av duration i R och för att skapa dessa tidsintervall gör vi exempelvis på följande sätt.

```
as.duration(myInterval)

[1] "65145600s (~2.06 years)"

dseconds(20)

[1] "20s"

dhours(1)

[1] "3600s (~1 hours)"

ddays(4)

[1] "345600s (~4 days)"

birth + ddays(1000)

[1] "2007-10-13"
```

Då tiden hela tiden utgår från sekunder är det enkelt att räkna exakt hur många dagar det går på en viss tidsperiod genom att bara dividera med `ddays(1)`.

1. Räkna ut följande:
   (a) Hur många dagar som finns i `myTime`
   (b) Hur många veckor (som 7-dagarsperioder) som finns i `myTime`
   (c) Hur många år som finns i `myTime`
2. Eftersom alla tidsintervall i duration är konstanta måste år ges ett fixt antal dagar. Räkna ut hur många dagar en `dyears(1)` är.

3.2.3 Period

Den sista typen av tidsintervall är det vi ofta i vanligt tal menar med datumintervall, d.v.s. hur många dagar, veckor eller månader som gått under en given period. Detta sätt att betrakta tid gör att vi kan lägga till olika långa tidsperioder, beroende på vad vi lägger till. Lägger vi till en månad till ett datum i februari blir det en kortare tidsperiod (sett som duration) än och vi lägger till en månad i juli.

Detta sätt innebär att en period håller koll på de olika tidsperioderna separat.

```
myPeriod <- as.period(myInterval)
myPeriod

[1] "2y 0m 24d 0H 0M 0S"

myPeriod / weeks(1)

[1] 107.79

myPeriod %/% weeks(1)

[1] 107
```

Med perioder blir det lite svårare att beräkna hur långa vissa tidpunkter är (eftersom det faktisk beror på vilken period vi faktiskt talar om). Detta gör att lubridate uppskattar tidperioderna efter hur många "hela" tidsperioder vi har i vår period. Dock kan vi använda perioder och heltalsdivision (%/%) för att beräkna hela perioder för olika intervall.

```
myPeriod / weeks(1)

[1] 107.79

myPeriod %/% weeks(1)

[1] 107

myInterval / weeks(1)

[1] 107.71

myInterval %/% weeks(1)

[1] 107
```

1. Pröva att beräkna följande baserat på `myTime` (pröva både med och utan heltalsdivision):
   (a) Hur många dagar du levt.
   (b) Hur många månader du levt.
   (c) Hur många veckor du levt.
   (d) Hur många år du levt.

3.3 Sekvenser med datum

Funktionen `seq()` är en generisk funktion, som kan användas för att skapa sekvenser med datum. Detta är användbart om vi t.ex. vill skapa en vektor som innehåller alla datum för tio år.

```
?seq.Date()
methods(seq)
```

1. Testa koden nedan. Vad händer?

```
seq(from = ymd('2012-04-07'),to = ymd('2014-03-22'),by='weeks')
seq(ymd('2012-04-07'),ymd('2014-03-22'), by = '1 week')
seq(ymd('2012-04-07'),ymd('2014-03-22'), by = '2 week')

seq(ymd('2012-04-07'),ymd('2014-03-22'),by='days')
seq(ymd('2012-04-07'),ymd('2014-03-22'),by='15 days')

seq(ymd('2012-04-07'),ymd('2014-03-22'),by='months')

seq(ymd('2012-04-07'),ymd('2014-03-22'),by='years')
```

2. Skapa följande sekvenser:
   (a) Alla dagar mellan 2014-01-20 till 2017-03-28
   (b) Varanan dag mellan 2014-01-20 till 2014-03-28, med start på den första dagen
   (c) Med datumet för alla fredagar under 2020
   (d) Med datumen för var fjärde vecka under hela 2019, med start 2019-01-01.

3. Tidserieplottar: Ibland vill vi göra plottar med data som varierar över tid. Då kommer sekvenser med datum väl till pass. Funktionen `plot()` kan anpassa sig automatiskt om x-variabeln är av datumklass. Testa koden nedan, och notera x-axeln i plottarna i båda fallen. Detta är användbart i samband med projektet.

```
# skapa tidseriedata:
x1<-seq(ymd("2017-01-01"),ymd("2020-12-31"),by="days")
N<-length(x1) X<-scale(cbind(1:N,(1:N)^2,(1:N)^3))
set.seed(403)
y<-130+2*X[,1]-2*X[,2]-10*X[,3]+rnorm(n = N,sd=1)

class(x1)
class(y)

# tidserieplot
plot(x1,y,t="l")
# vilken skala har x-axeln?

# lite mindre data:
plot(x1[1:365],y[1:365],t="l")
# vilken skala har x-axeln?

x2<-1:N
class(x2)
plot(x2,y,t="l")
# vilken skala har x-axeln?
```

3.4 Mer övningar

1. Skapa fyra vektorer: En som är en instant, en som är av typen interval, en av typen duration och en av typen period. Ni bestämmer själv vilka datum som variablerna ska innehålla. Testa sen att göra minst tre av de beräkningar som finns beskrivna i tabell 6 i artikeln om `lubridate`.

2. Skapa datumet "2010-04-23 12:33:45" med funktionen `ymd_hms()` och döp den till `testTimes`. Gör följande beräknigar:

   (a) Välj ut året med `year()`

   (b) Välj ut dagen med `day()`

   (c) Välj ut timmen med `hour()`

   (d) Välj ut sekunden med `second()`

   (e) Kolla i artikeln om lubridate hur ni kan göra avrundningar under sektion 6. Avrunda till:

      i. Nedåt till år

      ii. Uppåt till dag

      iii. Närmste heltalsminuten

   (f) Ändra nu följande saker i `testTimes`.

      i. Året till 1876

      ii. Sekunden till 21

      iii. Månaden till september.

3. Vill du ha mer övning på datum och tider?

   (a) Gå igenom koden här: vignette

   (b) Gå igenom koden här: Boken R for Data Science: 16 Dates and times

Det var allt för denna laboration!