Datorlaboration 3

Josef Wilzén och Måns Magnusson

4 februari 2016

Instruktioner

- Denna laboration ska göras **en och en**.
- Det är tillåtet att diskutera med andra, men att plagiera eller skriva kod åt varandra är **inte tillåtet**.
- Deadline för labben framgår på **LISAM**
- Laborationen ska lämnas in via **LISAM**.
- Utgå från laborationsfilen, som går att ladda ned **här**, när du gör inlämningsuppgifterna. Spara denna som labb[no]_[liuID].R , tex labb1_josad732.R om det är labb 1. Ta inte med hakparenterser i filnamnet. Denna fil ska laddas upp på LISAM och ska **inte** innehålla något annat än de aktuella funktionerna, namn- och ID-variabler och ev kommentarer. Alltså **inga** andra variabler, funktionsanrop för att testa inläminngsuppgifterna eller anrop till markmyassignment-funktioner.
- Laborationen består av två delar:
  - Datorlaborationen
  - Inlämningsuppgifter
- I laborationen finns det extrauppgifter markerade med *. Dessa kan hoppas över.

Innehåll

Del I

Datorlaboration

Kapitel 1

Programkontroll

En av de centrala delarna för att skriva effektiva och väl fungerande funktioner och kod i R är att kunna styra programmen på ett bra sätt. För detta används så kallad programkontroll. Generellt sett kan man säga att programkontrollen består av två huvudsakliga delar, villkorssatser och loopar.

1.1 Villkorssatser

Villkorssatser används för att kontrollera flödet i programmeringen på ett smidigt sätt och beroende på huruvida ett villkor är uppfyllt eller inte ska programmet göra olika saker. Grunden för villkorststyrning är `if`. Vill vi styra ett program behöver vi med logiska värdet ange vilka delar som ska utföras. Med if utförs dessa OM `if`-satsen är sann (`TRUE`), annars utförs den inte. Vi kan sedan använda `else` för de fall då uttrycket i if är falskt (`FALSE`).

Villkorssatser bygger helt på logiska värden i R som behandlades tidigare under laborationen om Logik i R.

1. Skapa `if`-satsen nedan. Pröva att ändra värdet på `x` på lämpligt sätt och se hur resultatet av `if`-satsen ändras.

```
x <- -100
if (x < 0) print("Hej!")

[1] "Hej!"

if (x > 0) print("Hej hej!")
```

2. För att kunna göra fler beräkningar i en `if`-sats måste { } användas. Kör koden nedan. Pröva olika värdet på `x`.

```
x<- -20
if(x < 0){
        print("Negativt x")
        a <- pi + 23
        print(a)
}

[1] "Negativt x"
[1] 26.142

if(x >= 0){
        a <- 100
}
print(a)
```

```
[1] 26.142
```

3. Alla logiska värden kan användas - så länge det är ett enda logiskt värde.

```
if(TRUE) x <- "Lotta"
if(FALSE) x <- "Lisa"
if(x == "Lotta"){
        print("Hej Lotta!")
}

[1] "Hej Lotta!"

if(!TRUE) x <- "Lisa"
if(x == "Lisa"){
        print("Hej Lisa!")
}
```

4. Nästa steg är att lägga till en `else`-sats. Testa nu att köra följande `if else`-sats (testa med olika värden för `x`)

```
x <- 100
if(x < 0){
        a <- 1
        print("Negativt x")
} else {
        a <- 2
        print("Positivt eller noll")
}
a
```

5. Pröva att göra en `if-else`-sats som skriver ut ‘‘`Male`’’ om värdet `x` är ‘‘`M`’’ och ‘‘`Female`’’ om värdet `x` är ‘‘`F`’’.

6. Det går också att göra flera logiska tester med fler `if - else`. Testa nu att köra en `if - else if - else` - sats med flera nivåer:

```
if(x==0){
        print("x är noll")
} else if(x < 0){
        print("x är negativ")
} else {
        print("x är positiv")
}
```

7. Skapa variabeln `cool_kvinna`. Skapa en `if - else if - else` sats som skriver ut födelseåret om vi anger förnamnet som ett textelement. Anges något annat namn/text ska programmet returnera `NA`.

   (a) Amelia Earhart (1897)

   (b) Ada Lovelace (1815)

   (c) Vigdis Finnbogadottir (1930)

1.1.1 * Extraproblem

1. Skapa ett program som med en villkorssats som skriver ut namnet på en av kvinnorna ovan om rätt födelseår har angetts. Om inget korrekt födelseår har angett ska programmet skriva ut ‘‘`Hillary Clinton`’’.

1.2 Loopar

En av de mest centrala verktygen för all programmering är användandet av loopar. Dessa används för att utföra upprepande uppgifter och är en central del i att skriva välfungerande program.

1.2.1 `for` - loop

1. En `for`-loop har ett loop-element (`i`) och en loop-vektor (t.ex. `1:10`). I koden nedan är `i` loop-elementet och `1:10` är vektorn som loopas över. Testa att köra koden.

```
for(i in 1:10){
        x<-i+3
        print(x)
}

for(i in 1:10) print(i+3)

y <- 0
for(i in 1:10) {
    y <- y + i
}
```

2. Testa att ändra `1:10` till `1:5` och `5:1`. Vad händer nu? Testa att använda loop-vektorn `seq(1, 6, by=2)`

3. Skriv en `for`-loop som skriver ut texten `Övning ger färdighet` 20 gånger med `print()`.

4. Testa nu att köra koden nedan. Vad händer? Testa att ändra på vektorn `minVektor` till lämplia värden. Vilka värden ska `minVektor` ha om du vill bara skriva ut de tre sista orden?

```
minaOrd <- c("campus","sal","kravall","tenta","senare","konjunktur")
minVektor<-1:5

for(i in 1:length(minaOrd)){
        print(minaOrd[i])
}

for(i in minVektor){
        print(minaOrd[i])
}

for(ord in minaOrd){
        print(ord)
}
```

5. En bra funktion för att skapa loop-vektorer är funktionen `seq_along()`. Den skapar en loop-vektor på samma sätt som `1:length(minaOrd)`. Dock blir det tydligare i koden vad loopen gör (sequence along `minaOrd`).

```r
for(i in 1:length(minaOrd)){
        print(i)
}

for(i in seq_along(minaOrd)){
        print(i)
}
```

6. Det går också att använda en loop för att iterera över element i en lista.

```r
myList <- list("Hej",3:8,c("Lite mer text", "och lite nuffror"), 4:12)
for (element in myList){
        print(element)
}
```

7. Pröva att skriva en `for`-loop som:

   (a) Summerar talen 0 till 200

   (b) Skriver ut "I love R!" 20 gånger

   (c) Skriver ut talen 1 till 20 och den kumulativa summan från 1 till 20

1.2.2 * Extraproblem

1. Skriv en `for`-loop som skriver ut alla heltal som är jämt delbara med 13 som finns mellan 1 och 200 med hjälp av en loop och villkorssats. [**Tips!** `?%%`]

2. Skriv en `for`-loop som skriver ut alla heltal som är jämt delbara med 3 som finns mellan 1 och 200. Förutom att skriva ut dessa tal ska de även sparas i en vektorn `delatMedTre`. Men bara de tal som är **udda** ska vara med. Använd en villkorssats för att göra den förändringen. Om ett av talen är jämt, så skriv ut texten `''Intresserar mig inte''` till skärmen. [**Tips!** `?%%`]

1.2.3 Nästlade `for`-loopar

1. Följande kod är ett exempel på en nästlad loop för att loopa över flera index (exempelvis rader och kolumner). Denna loop är nästlad i två nivåer. I teorin kan vi nästla en loop i hur många nivåer vi vill. Men ju fler nivåer, desto svårare är det att kunna läsa koden och följa vad som sker i programmet.

```r
for (i in 1:2){
        for (j in 1:3){
                print(i)
                print(j)
        }
}
```

2. Vi ska nu pröva att summera elementen i två matriser med en nästlad `for`-loop.

```r
A <- matrix(1:6,ncol=3)
B <- matrix(1:6,ncol=3)
C <- matrix(0, ncol=3, nrow=2)
for (i in 1:2){
        for (j in 1:3){
                print(A[i,j])
```

```
                print(B[i,j])
                C[i,j] <- A[i,j] + B[i,j]
                print(C[i,j])
        }
}
```

3. Ändra koden ovan för matriser som är av storlek 3×3. Testa med följande två matriser. Hur behöver du ändra koden för att det ska fungera?

```
A <- matrix(1:9,ncol=3)
B <- matrix(10:8,ncol=3)
```

1.2.4 `while` loopar

1. En `while`-loop loopar så länge villkoret är sant och inte ett bestämt antal gånger som `for`-loopar. På detta sätt liknar det en `if`-sats fast som loop. Testa koden nedan med några olika värden på `x`.

```
x<-1
while(x<10){
  print("x is less than 10")
  x<-x+1
}
```

2. Om inte `while`-loopar skrivs på rätt sätt kan de loopa i "oändlighet". Vad är viktigt att tänka på i while-loop används för att undvika detta?
Obs! Om du testar koden nedan vill du nog avbryta.
I R-studio: trycka på stop-knappen i kanten på console - fönstret eller med menyn "Session"→"Interrupt R".
Om du kör vanliga R: tryck "ctrl+C" .

```
x<-1
while(x<10){
  print("x is less than 10")
  x<-x-1
  print(x)
}
```

3. Skriv en `while` - loop som:

   (a) Skriver ut talen 1 till 35

   (b) Summerar talen 5 till 200

   (c) Skriver ut "I love R!" 20 gånger

   (d) Skriver ut talen 1 till 20 och den kumulativa summan från 1 till 20

   (e) Skriver ut alla jämna tal mellan 1 och 20. [**Tips!** `?%%`]

1.2.5 Kontrollstrukturer för loopar och `repeat{}`

För att kontrollera loopar finns det två huvudsakliga kontrollstrukturer.

Kontrollstruktur	Betydelse
`next()`	Hoppa vidare till nästa iteration i loopen
`break()`	Avbryt den aktuella loopen

Dessa två sätt att kontrollera en loop är mycket värdefulla och gör det möjligt att avsluta en hel loop i förtid (`break`) eller hoppa över beräkningar för den nuvarande iterationen (`next`).

1. Nedan är ett exempel på kod som använder kontrollstrukturen `next`. Innan beräkningar i loopen görs prövar vi med en villkorssats om beräkningen är möjlig.
   Pröva koden och pröva sedan att ta bort `next` och se vad som händer.

```
myList <- list("Hej",3:8,c("Lite mer text", "och lite nuffror"), 4:12)

for (element in myList){
        if(typeof(element) != "integer"){ next() }
        print(mean(element))
}
```

2. Använd nu `next()` för att skriva ut alla tal mellan 13 och 200 som är jämt delbara med 13. [**Tips!** %%]

3. På samma sätt som `next` kan användas för att begränsa vissa beräkningar kan `break` avsluta en `for`-loop när exempelvis en beräkning är tillräckligt bra. Det blir då en form av `while` loop fast med ett begränsat antal iterationer. `while` loopen i uppgift 1 på sida 8 kan på detta skrivas om med `break` på följande sätt. Pröva denna kod och experimentera lite med `x`.

```
x<-1
for (i in 1:20) {
  if( x > 10 ) break()
  print("x is less than 10")
  x<-x+1
}

[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
[1] "x is less than 10"
```

4. Skriv en `for` - loop som itererar över loop vektorn `1:100`. Använd `break` för att...

   (a) Skriver ut talen 1 till 35

   (b) Summerar talen 1 till 20

   (c) Skriver ut “I love R!” 20 gånger

En sista typ av loop som kan användas är `repeat{}`. Till skillnad från for och while-loopar kommer denna struktur fortsätta iterera till dess att den stöter på ett `break`. Precis som med `while`-loopar kan detta innebära att programmet aldrig avslutas.

1. Nedan är ett exempel på kod som använder `repeat{}`

```r
x<-1
repeat {
  x <- x + 1
  print(x)
  if( x > 5 ) break()
}
```

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

2. Skapa en `repeat`-loop som...
   (a) Skriver ut talen 1 till 35
   (b) Summerar talen 1 till 20
   (c) Skriver ut "I love R!" 20 gånger

1.2.6 * Extraproblem

1. Skapa med `repeat`, `next` och `break` kod som gör följande:
   (a) Skriver ut alla jämna tal mellan 3 och 17.
   (b) Beräknar och skriver ut resultat i varje steg av den kumulativa summan från 10 till 20.
2. Skapa en egen `sum()` funktion och `mean()` som med en `for`-loop beräknar summan och medelvärdet för en godtycklig numerisk vektor.

1.3 Avbryta funktioner och generera varningar

Ibland vill vi att ett program ska avbrytas om vissa villkor inte är uppfyllda. Det kan vara att argument till en funktion inte är korrekt eller att resultat som beräknats är felaktiga. För att avbryta ett R-program använder vi `stop()`.

1. Skapa följande funktion (som avbryts om `x>10`) och pröva lite olika värden på `x`.

```r
test_funktion <- function(x){
  if(x>10) stop()
  return("Yay!")
}
```

2. Det går också att generera **felmeddelanden** med `stop()`. Detta kan vara bra för att kunna identifiera var programmet var tvungen att avbrytas. Pröva att lägga till felmeddelandet nedan.

```r
test_funktion <- function(x){
  if(x>10) stop("x > 10 juh!")
  return("Yay!")
}
```

Ibland kan det vara så att vi inte vill avbryta ett pågående program utan att vi istället bara skulle vilja varna för att det kan vara något fel. Det görs med funktionen `warning()`.

1. Skapa följande funktion (som avbryts om `x>10`) och pröva lite olika värden på `x`.

```
test_funktion <- function(x){
  if(x>10) warning()
  return("Yay!")
}
```

2. Med `warning()` kan vi också ange varningsmeddelanden.

```
test_funktion <- function(x){
  if(x>10) warning("x>10 juh!")
  return("Yay!")
}
```

3. Om ett program genererar flera varningar sparas dessa och det går att gå igenom samtliga varningar efter att programmet kört klart. För att koma åt dessa varningar använder vi funktionen `warnings()`. Pröva följande kod.

```
for(i in 1:20){
        test_funktion(i)
}
```

4. En fördel med varningar är att vi kan tysta dem om vi vill. Detta går inte med ex. `cat()` eller `print()` vilket gör att dessa funktioner inte ska användas för att generera varningar.

```
suppressWarnings(test_funktion(100))

[1] "Yay!"
```

1.4 Debugging

Debugging handlar om att hitta och rätta fel i ens programmeringskod. Det kan göras på en mängd olika sätt.

1. Syntaktiska fel: Testa att kör koden nedan. Vad händer? försök tolka felmeddelandet och rätta sedan koden.

```
f<-function(x,y){
x2<-sin(x)
y2<-log(y)
z<-x2^(y2^2-3*y2
return(z)
}
```

2. Semantiska fel:

   (a) Testa att kör koden nedan. Vad händer? Ger funktionen rätt respons?

```r
f<-function(x,y){
        x_mean<-mean(x)
        x_mean2<-x_mean^2
        print(x_mean2)
        y_mean<-mean(y)
        y_mean2<-y_mean^2
        print(y_mean2)
}
x<-4:193
y<-c("1","3","87","321","31")
f(x = x,y = y)
```

(b) Försök nu att lägga in test i funktionen som testar om `x` och `y` är numeriska innan medelvärdet beräknas. Om de inte är numeriska skriv ut `''not numeric''` till skärmen.

3. Logiska fel: Testa att kör koden nedan. Vad händer? Ger funktionen rätt respons? Försök att rätta funktionen.

```r
# my_col_stats: a function for calculating stats for a specified column in a dataset
# my_data - a data.frame
# index - the column of intrest

my_col_stats<-function(my_data,index){
        x<-my_data[,1]
        return(list(mean=mean(x),meadian=median(x),sd=sd(x)))
}

data("iris")
my_col_stats(my_data = iris,index = 1)
my_col_stats(my_data = iris,index = 2)
my_col_stats(my_data = iris,index = 3)
my_col_stats(my_data = iris,index = 4)
```

4. `browser()` och `debug()`: Kör koden nedan.

```r
h<-function(x){
        x_sum<-sum(x)
        x_mean<-mean(x)
        x2<-x^2
        z<-5
        y<-exp(x2)-z
        return(y)
}
debug(h)
a<-matrix(1:4,2,2)
h(a)
undebug(h)
```

(a) Använd `debug` för att stega igenom funktionen. Använd följande komandon för att navigera i debugg-mode: `n` ,`c` och `Q`. Kolla kontinuerligt i enviroment-filken i RStudio och se hur de lokala variablerna ändras. I debugg-mode kan vanliga R-funktioner anropas. Testa att köra `print(x)`, `sin(x)` och `x^5` i debugg-mode.

(b) Testa nu att sätta in `browser()` innan raden med `x2<-x^2` , läs den uppdaterade funktionen och anropa `h(a)`. Vad blir skilnaden jämfört med `debug()`?

(c) Testa att istället sätta in `if(!is.numeric(x)) browser()` innan raden med `x_sum<-sum(x)`. Testa nu med `h(a)` och `h(''hej'')`. Vad blir skilnaden från föregående uppgift?

(d) När det är bättre att använda `browser()` jämfört med `debug()`? Diskutera med någon!

5. Stina vill skriva en funktion som kollar om ett värde `a` (en skalär) finns som element i en vektor `b`. Hon skrev då funktionen `isIn` nedan.

```
isIn <- function(a, b) {
        j <- 1
        while (j <= length(b)) {
                if(a == b[j]) {
                        out <- TRUE}
                else {
                        out <- FALSE
                }
                j <- j + 1
        }
        return(out)
}
```

(a) Testa isIn med anropen `isIn(3,1:3)` och `isIn(3,1:5)`. Gör funktonen det den ska?

(b) Placera `browser()` på lämpligt ställe i koden. Undersök vad som händer i loopen. Du kan även testa andra debuggingfunktioner. Tips `?debug`

(c) Ta bort buggen med minimal förändring av koden.

6. Stina vill nu utöka sin funktion så att `a` kan vara en vekor, för att kunna kolla vilka element i `a` som finns i `b`. Hon ändrade då `isIn` till:

```
isIn <- function(a, b) {
        out <- rep(NA, length(a))
        j <- 0
        while(j <= length(b)) {
        j <- j + 1
                for ( i in 1:length(a)) {
                        out[i] <- (a[i] == b[j])
                }
        }
        return(out)
}
```

(a) Testa `isIn` med `a<-1:5` och `b<-3:9`. Funkar funktionen som den ska?

(b) Placera `browser()` på lämpligt ställe i koden. Undersök vad som händer i looparna. Du kan även testa andra debuggingfunktioner. Tips `?debug`

(c) Ta bort buggen med minimal förändring av koden.

(d) Jämför med `%in%` om `isIn` fungerar som den ska.

Del II

Inlämningsuppgifter

Tips!

Inlämningsuppgifterna innebär att konstruera funktioner. Ofta är det bra att bryta ned programmeringsuppgifter i färre små steg och testa att det fungerar i varje steg.

1. Lös uppgiften med vanlig kod direkt i R-Studio (precis som i datorlaborationen ovan) utan att skapa en funktion.
2. Testa att du får samma resultat som testexemplen.
3. Implementera koden du skrivit i 1. ovan som en funktion.
4. Testa att du får samma resultat som i testexemplen, nu med funktionen.

Automatisk återkoppling med `markmyassignment`

Som ett komplement för att snabbt kunna få återkoppling på de olika arbetsuppgifterna finns paketet `markmyassignment`. Med detta är det möjligt att direkt få återkoppling på uppgifterna i laborationen, oavsett dator. Dock krävs internetanalsutning.

Information om hur du installerar och använder `markmyassignment` för att få direkt återkoppling på dina laborationer finns att tillgå **här**.

Samma information finns också i R och går att läsa genom att först installera `markmyassignment`.

```
install.packages("markmyassignment")
```

Om du ska installera ett paket i PC-pularana så behöver du ange följande:

```
install.packages("markmyassignment",lib="sökväg till en mapp i din hemkatalog")
```

Tänk på att i sökvägar till mappar/filer i R i Windowssystem så används ‘‘\\’’, tex ‘‘C:\\Users\\Josef’’.

Därefter går det att läsa information om hur du använder `markmyassignment` med följande kommando i R:

```
vignette("markmyassignment")
```

Det går även att komma åt vignetten **här**. Till sist går det att komma åt hjälpfilerna och dokumentationen i `markmyassignment` på följande sätt:

```
help(package="markmyassignment")
```

Lycka till!

Kapitel 2

Inlämningsuppgifter

För att använda `markmyassignment` i denna laboration ange:

```
library(markmyassignment)
```

```
Loading required package: methods
Loading required package: yaml
Loading required package: testthat
Loading required package: httr
```

```
lab_path <-
"https://raw.githubusercontent.com/STIMALiU/KursRprgm/master/Labs/Tests/d3.yml"
set_assignment(lab_path)
```

```
Assignment set:
D3 : Statistisk programmering med R: Lab 3
```

Kom ihåg: Om era funktioner inte fungerar som de ska testa debuggging! Använd `print` eller `cat` för att skriva ut relevanta värden. Använd `browser`/`debug` för att stega igenom funktionerna. Glöm inte att ta bort debuggingkoden innan du lämnar in labben.

2.1 `bmi()`

Skriv en funktion som du kallar `bmi()` med argumenten `bodyHeight` och `bodyWeight`. Funktionen ska beräkna BMI på följande sätt:

$$\texttt{bmi(body\_weight,body\_height)} = \frac{\texttt{body\_weight}}{\texttt{body\_height}^2}$$

och returnera värdet. Om `body_height` och/eller `body_weight` är mindre eller lika med 0 ska funktionen **varna** att den aktuella variabeln är mindre eller lika med 0 och att resultatet inte är meningsfullt:
`body_weight is not positive, calculation is not meaningful` eller
`body_height is not positive, calculation is not meaningful`
Testa med olika värden för `body_length` och `body_weight`.

Här är ett textexempel på hur funktionen ska fungera:

```
bmi(body_weight = 95, body_height = 1.98)
```

```
[1] 24.232
```

```
myBMI <- bmi(body_weight = 95, body_height = 1.98)
myBMI
```

```
[1] 24.232
```

```
bmi(body_weight = 74, body_height = -1.83)
```

```
Warning in bmi(body_weight = 74, body_height = -1.83): body_height is not positive, calculation
is not meaningful
```

```
[1] 22.097
```

```
bmi(body_weight = 0, body_height = 1.63)
```

```
Warning in bmi(body_weight = 0, body_height = 1.63): body_weight is not positive, calculation
is not meaningful
```

```
[1] 0
```

```
bmi(body_weight = -73, body_height = 0)
```

```
Warning in bmi(body_weight = -73, body_height = 0): body_weight is not positive, calculation
is not meaningful
Warning in bmi(body_weight = -73, body_height = 0): body_height is not positive, calculation
is not meaningful
```

```
[1] -Inf
```

```
suppressWarnings(bmi(body_weight = -73, body_height = 0))
```

```
[1] -Inf
```

2.2 my_matrix_prod()

En central del inom den linjära algebran är matrismultiplikation , d.v.s. att multiplicera två matriser med varandra. Du ska nu skapa en funktion kallad myMatrixProd() med argumenten A och B som multiplicerar två matriser med varandra på följande sätt:

$$\texttt{my\_matrix\_prod(A, B)} = \mathbf{A} \cdot \mathbf{B}$$

Om dimensionerna inte gör det möjligt att multiplicera matriserna ska funktionen stoppas och returnera felmeddelandet `Matrix dimensions mismatch`. Observera att det inte är tillåtet att använda R:s funktion för matrismultiplikation **%*%**. Du får dock använda den för att generera fler testfall för att testa att din funktion räknar rätt.

De steg funktionen kan gå igenom är följande:

1. Pröva om dimensionerna av matris **A** och **B** innebär att de kan multipliceras med varandra, stoppa annars funktionen och returnera felmeddelandet.
2. Skapa en ny matris (ex. kallad **C**) med de dimensioner som produkten av A och B har.
3. Loopa över elementen i **C** och räkna ut varje element för sig. [**Tips!** Här kan du använda din kod från `orth_scalar_prod()`]

Här är textexempel på hur funktionen ska fungera:

```
X <- matrix(1:6, nrow=2, ncol=3)
Y <- matrix(6:1, nrow=3, ncol=2)
my_matrix_prod(A = X, B = Y)
```

```
     [,1] [,2]
[1,]   41   14
[2,]   56   20
```

```
my_matrix_prod(A = Y, B = X)
```

```
     [,1] [,2] [,3]
[1,]   12   30   48
[2,]    9   23   37
[3,]    6   16   26
```

```
my_matrix_prod(A = X, B = X)
```

```
Error : Matrix dimensions mismatch
```

2.3 babylon()

En algoritm för att approximera kvadratroten ur ett tal är den så kallade babylonska metoden, en metod som flera säkert känner igen från gymnasiet. Det är ett sätt att räkna ut kvadratroten för ett godtyckligt tal.

Metoden, som är ett specialfall av Newton-Raphsons metod, kan beskrivas på följande sätt:

1. Starta med ett godtyckligt förslag på kvadratroten, kallat r_0. Vi behöver starta vår algoritm i någon punkt. Ju närmare den sanna kvadratroten vi startar desto färre iterationer kommer behövas.
2. Beräkna ett nytt förslag på roten på följande sätt:

$$r_{n+1} = \frac{r_n + \frac{x}{r_n}}{2}$$

3. Om $|r_{n+1} - r_n|$ inte har uppnått godtycklig nogrannhet: gå till steg 2 igen.
   Obs! || indikerar absolutbeloppet av skillnaden mellan iterationerna. Mer information om absolutbeloppet finns **[här]**.

Implementera denna algoritm som en funktion i R. Funktionen ska heta `babylon()` och argumenten `x`, `init` och `tol`. `x` är talet för vilket kvadratroten ska approximeras, `init` är det första förslaget på kvadratroten och `tol` är hur stor noggrannhet som ska krävas för att avsluta algoritmen.

Funktionen kan implementeras antingen som en `for` - loop med `break` eller en `while` loop. Funktionen ska returnera en lista med två element, `rot` och `iter` (båda numeriska värden). I elementet `rot` ska approximationen av kvadratroten returneras och i elementet `iter` ska antalet iterationer returneras.

Obs! Det är inte tillåtet att använda funktionen `sqrt()` i denna uppgift.

Här är textexempel på hur funktionen ska fungera:

```
babylon(x = 2, init = 1.5, tol = 0.01)
```

```
$rot
[1] 1.4142

$iter
[1] 2
```

```
sqrt(2)
```

```
[1] 1.4142
```

```
babylon(x = 3, init = 2, tol = 0.000001)
```

```
$rot
[1] 1.7321

$iter
[1] 4
```

```
sqrt(3)
```

```
[1] 1.7321
```

```
babylon(x = 15, init = 1.5, tol = 0.01)

$rot
[1] 3.873

$iter
[1] 5

sqrt(15)

[1] 3.873
```

2.4 `my_moving_average()`

I denna uppgift ska vi skapa en funktion som beräknar glidande medelvärden som du kallar `my_moving_average()`.

Funktionen ska kunna ta en godtycklig vektor `x`, och ett argument `n`. Först ska den kontrollera att vektorn är numerisk. Är vektorn inte numerisk ska funktionen avbrytas och skriva ut felmeddelandet `''Not a numeric vector!''`.

Annars ska funktionen beräkna det glidande medelvärdet på följande sätt

$$y_t = \frac{x_t + \cdots + x_{t+n-1}}{n}$$

där y_t är det t:e elementet i vektorn som ska returneras. Detta innebär att vektorn y är $n-1$ element kortare än x.

Här är testexempel på hur funktionen ska fungera:

```
my_moving_average(x = 1:10, n=2)

[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5

my_moving_average(x = "Pelle", n=2)

Error : Not a numeric vector!

my_moving_average(x = 5:15, n=4)

[1]  6.5  7.5  8.5  9.5 10.5 11.5 12.5 13.5
```

Grattis! Nu är du klar!