Splus-seminar –
Programmering i S

Geir Storvik

Innhold:

1. Muligheter innenfor Splus
2. Kontroll strukturer
3. Vektoriserte beregninger og løkker
4. Hvordan lage funksjoner
5. Editering av funksjoner (og objekter)
6. Debugging
7. Hvordan gjøre data og funksjoner tilgjengelig for andre
8. Dokumentasjon
9. Objekt orientert programmering

10. Interaksjon mot UNIX

11. Interaksjon mot C

Muligheter innenfor Splus

1. Stort antall ferdige funksjoner
2. Mange rutiner/biblioteker på nettet
3. Lokale biblioteker
4. Egne funksjoner
5. Objektorientert programmering
6. Interaksjon med Unix
7. Interaksjon med C

Kontroll strukturer

if-setninger:

```
if (condition) statement1 else statement2
```

Bruk av && og ||:

```
if (is.numeric(x) && min(x) > 0)
  sx < sqrt(x)
else
  stop("x must be numeric and
        all components positive")
```

ifelse:

Opererer på vektor

```
ifelse(x>0,log(x),0)
```

Kan gi unødige feilmeldinger.

switch:

```
switch(command,
 plot = do.plot(),
 print = do.print(),
 stop("command should be plot or print"))
```

Løkker:

Tre muligheter: `for, while, repeat`

```
for(i in 1:10)
  {
   kommandoer
  }


while(conv==0)
  {
   kommandoer
   conv <- ...
  }
```

For - løkker:

Kaller en separat S-prosessor:

```
For(i=1:250;res[i] <-
  myfun(sample(x,replace=T)) )
```

Vektoriserte beregninger og løkker

Beregninger vha løkker i Splus går tregt.

Eks: `x` er en 500×500 matrise der vi ønsker gj.snitt for hver kollonne. Tre mulige fremgangsmåter:

```
n <- 500

x <- matrix(rnorm(n*n),ncol=n)


y1 <- rep(0,n)

for(i in 1:n)

 y1[i] <- mean(x[,i])


y2 <- apply(x,2,mean)


y3 <- rep(1,n)%*%x/n
```

Beregningstider: 19 sek, 35 sek, 8 sek.

Apply-funksjonene (`apply, tapply, apply`, `lapply`) skal være raskere i versjon 3.2.

Hvordan lage funksjoner

Definisjon:

```
foo <- function(x,y,exp=2)
{
 z1 <- (x+y)^exp
 z2 <- x^exp+y^exp
 list(z1=z1,z2=z2)
}
```

Alternativ til `list(z1=z1,z2=z2)`:

```
 return(z1,z2)
```

- Bruk av **`return`** gir en liste av variablene
- Variable uten = etter seg må gis inn ved kall

- Variable med = etter seg er valgfrie (default-verdi etter likhetstegn)

```
> foo(2,3)
$z1:
[1] 25


$z2:
[1] 13


> unlist(foo(2,3))
 z1 z2
 25 13
```

`unlist` “pakker” ut lister.

Bruk av ... argument

Angir at et hvilket som helst *navngitt* argument kan spesifiseres ved kall.

Anta funksjon med opsjon om plotting.

Av interesse å kunne gi inn plotte-parametre:

```
simul.diff <- function(x1=50,x2=55,
  s1=sqrt(12),s2=sqrt(40),n1=20,n2=12,
  m=10000,dplot=T,...)
{
 t1 <- rt(m,n1-1)
 theta1 <- x1+t1*s1/sqrt(n1)
 t2 <- rt(m,n2-1)
 theta2 <- x2+t2*s2/sqrt(n2)
 gamma <- theta2-theta1
```

```
 if(dplot)
   plot(density(gamma),type="l",...)
 gamma
}

> gamma<-simul.diff(xlab="gamma",
                    ylab="density")
```

Editering av funksjoner (og objekter)

3 muligheter:

1. "Cut and paste"
2. `dump` og `source`
3. `fix`

   ```
   > options(editor="emacs")
   > fix(foo)
   ```

   vil lage et nytt emacs-vindu som inneholder funksjonen `foo`. Lukking av vinduet vil oppdatere `foo`.

Alternativ:

```
> foo <- ed(foo,editor="emacs")
```

NB:
I begge tilfelle vil en ny versjon av `foo` legges i *arbeidsområdet* uavhengig av hvor `foo` lå tidligere.

Kan flyttes tilbake ved `assign`

Debugging

Mange helpefunkjoner for debugging

`print,cat`	Printe ut variable i rutine.
`options(warn=2)`	Gjør at "warning" blir endret til feil
`traceback()`	Printer kall i prosess frem til feil
`options(dump.frames)`	Gjør at "warning" blir endret til feil
`last.dump`	Inneholder liste over kall frem til feil
`debugger`	Funksjon for inspeksjon av `last.dump`
	Kan sjekke verdi av variable etc
`browser`	Kan legges inn i funksjon for inspeksjon av variable
`tprint`	Printer en linje-nummerert versjon av funksjon
`trace`	Sette merker i funksjon for inspeksjon,
	f.eks sette inn browser
`inspect`	Interaktiv debugger (Versjon 3.2)

`inspect` klart å foretrekke når versjon 3.2 kommer.

Frem til da, bruk av de andre funksjonene i kombinasjon.

Eksempel

```
> options(error=dump.frames)
> foo(-5,3,-2.5)
Error in (x + y)^exp:
  (warning converted to error)
  (-2)^(-2.5) DOMAIN error

Dumped
> traceback()
Message: (warning converted to error)
  (-2)^(-2.5) DOMAIN error

3: (x + y)^exp
2: foo(-5, 3, -2.5)
```

```
1:
> tprint(foo)
{
1: z1 <- (x + y)^exp
2: z2 <- x^exp + y^exp + 1
3: return(z1, z2)
}
```

```
> trace(foo,tracer=browser,at=1)
> foo(-5,3,-2.5)
At 1: Called from: foo(-5, 3, -2.5)
b(2)> ?
1: x
2: y
3: exp
b(2)> x+y
[1] -2
b(2)> 0
Error in (x + y)^exp:
 (warning converted to error)
  (-2)^(-2.5) DOMAIN error

Dumped
```

Hvordan gjøre data og funksjoner tilgjengelig for andre

Legge data i lokalt bibliotek:

```
> search()
[1] "/nr/pluto/u5/storvik/.Data"
[2] "/nr/rosmer/u2/splus3.1/library/local/.Data"
[3] "/nr/rosmer/u2/splus3.1/splus/.Functions"
[4] "/nr/rosmer/u2/splus3.1/stat/.Functions"
[5] "/nr/rosmer/u2/splus3.1/s/.Functions"
[6] "/nr/rosmer/u2/splus3.1/s/.Datasets"
[7] "/nr/rosmer/u2/splus3.1/stat/.Datasets"
[8] "/nr/rosmer/u2/splus3.1/splus/.Datasets"
```

```
> assign("local.x",x,where=2)
```

Legge funksjoner i lokalt bibliotek:

```
> assign("local.func",func,where=2)
```

Dokumentasjon

Anta du har en funksjon kalt `func`.

Ønsker å lage dokumentasjonsside. Si

```
> prompt("func")
```

Det blir da laget en fil kalt `func.d`. Fyll ut feltene.

Hvis kun egen funksjon:

```
> !mv func.d ~/.Data/.Help/func
> help.findsum("~/.Data")
```

Legg opp dokumentasjonssiden lokalt:

```
> makemanpage("func")
```

```
.BG
.FN func
.TL
~function to do ???
.DN
~brief description
.CS
func(filename, nx)
.RA
.OA
~move the above line to just above the first optional argument
.AG filename
~Describe filename here
.AG nx
~Describe nx here
.RT
~Describe the value returned
.SE
~describe any side effects if they exist
.DT
~explain details here.
.SH REFERENCES
```

```
~put references here, make other sections
 like NOTE and WARNING with .SH
.SA
~put functions to SEE ALSO here
.EX
# The function is currently defined as
function(filename, nx)
{
x <- scan(filename)
n <- floor(length(x)/nx)
y <- t(array(x[1:(n * nx)], c(nx, n)))
y
}
.KW ~keyword
.WR
```

Objekt orientert programmering

Objekter kan tillegges klasse:

```
> class(myobject) <- "lm"
> class(myobject)
[1] "lm"
```

Generiske funksjoner opererer på ulike klasser.

Hvordan generiske funksjoner fungerer

Anta `fit` har klasse `lm`:

```
> fit <- lm(y~x)
> fit <- lm(y~x)
> class(fit)
[1] "lm"
```

Ved kall på `print`, vil det letes etter en funksjon `print.lm`

Hvis `print.lm` ikke finnes, vil `print.default` brukes.

Arving:

Et objekt kan ha flere tilhørende klasser, f.eks

```
> x <- ordered(1:10)
> class(x)
[1] "ordered" "factor"
```

I dette tilfelle vil `x` “arve” metodene tilhørende klasse `ordered`.

Ved

```
> plot(x)
```

vil det først letes etter `plot.ordered`. Denne finnes ikke

Deretter letes det etter `plot.factor` som vil brukes.

Arving kan anvendes for å slippe mye programmering.

Ved definering av en ny klasse, kan en bruke visse funksjoner til eksisterende klasser.

Eksempel:

Ønsker å lage ny en robust regresjonsmetode, `rlm`.

La objektet ha klasse `rlm`

Kan være naturlig å "arve" mange av funksjonene til klassen `lm`

Interaksjon mot UNIX

Bruk av programvare som finnes på Unix nivå kan lette mye programmeringsarbeid.

Kommando:

```
> unix(command, input=NULL, output.to.S=T)
```

`command` er en kommando (i Bourne shell) som blir utført.

Eksempel::

Konvertering fra store til små bokstaver.

```
> to.upper <- function(str)
   unix('tr "[a-z]" "[A-Z]"',str)
> to.upper("abc")
```

```
[1] "ABC"
```

Interaksjon med C

Muligheter:

1. Kall på C (og Fortran) rutiner fra S
2. Kall på S rutiner fra C (og Fortran)

Vil kunne snakke om kall fra S.

Hvorfor:

1. Mange ferdig-rutiner
2. Programmering i S direkte er tregt

Hvordan:

1. Skriv C-rutine

2. Kompiler rutinen
3. Last inn rutinen
4. Kall på rutinen

Skriving av C-rutiner

1. Rutinen som skal kalles fra S, må deklareres som void.
2. All overføring av parametre gjøres ved pekere.
3. Korresponderende data-typer:

S	C
"logical"	long *
"integer"	long *
"single"	float *
"double"	double *
"character"	char **

NB:

Ved overføring av matriser og arrays fra S, vil disse oppfattes som vektorer i C.

I S løper første indeks fortest, mens i C løper siste indeks fortest: La `m` være en $p \times q \times r$ array i S. La `mat` være tilsvarende vektor i C. Da er

```
m[i,j,k] = mat[k * q * p + j * p + i]
```

Kompilering av C-rutiner

```
> !Splus COMPILE test.c
```

Ved flere `.c`-filer:

```
ld -r -d -o all.o test1.o test2.o ...
```

Innlasting av C-rutiner til S:

```
> dyn.load("test.o")
```

Sjekk om rutine er lastet inn:

```
> is.loaded("test")
```

som vil svare `T` eller `F` avhengig av om rutinen er lastet inn eller ikke

Kall på C-rutiner fra S

Bruker funksjonen `.C`. Første parameter er navn på rutinen, neste parametre er parametre til rutinen.

Eksempel:

C-rutinen:

```
void classify(long *i_n, long *i_p,
        long *i_ncl, double *i_data,
        double *i_mu, double *i_sigmainv,
        long *o_clas)
```

```
classify_function(data, mu, sigma)
{
 if(!is.loaded("classify"))
  dyn.load("classify.o")
 n <- dim(data)[1]
 p <- dim(data)[2]
 cl <- dim(mu)[2]
 sinv_array(0,dim(sigma))
 for(i in 1:cl)
  sinv[,,i] <- solve(sigma[,,i])
 data <- t(data)
 clas <- rep(0,n)
 storage.mode(data) <- "double"
 storage.mode(mu) <- "double"
 storage.mode(sinv) <- "double"
 storage.mode(clas) <- "integer"
```

```
  .C("classify",as.integer(n),
     as.integer(p),as.integer(cl),
     data,mu,sinv,clas = clas)$clas
}
```

Referanser

"Introductory Guide to S-Plus," B.D. Ripley. (statlib)

"Modern Applied Statistics with S-Plus", W.N. Venables & B.D. Ripley

"Statistical Models in S", J, Chambers & T. Hastie

"The New S Language", R. Becker, J. Chambers & A. Wilks