1 Introduction

This document desribes some commands in Splus, both native commands and additions written at the MRI and the examples given are using data from the MRI. Many of the commands described work in R but some of the additions do not but the goal is to edit the additions so they will work in R which use is growing very rapidly at the moment.

2 Differences between R and Splus

R and $Splus$ are both based on the S-language that was developed at AT and T in early stone age according to computer development. For most parts scripts in R and Splus.

In R results are stored in a file called $.RData$ under the directory where R is started (in unix). If R is started under a directory other $.RData$ files can be attached similar to what is done in Splus

$>$ attach("SMB/.RData") $>$ search() [1] ".GlobalEnv" "file:SMB/.RData" "package:methods" "package:stats" [5] "package:graphics" "package:utils" "Autoloads" "package:base"

.GlobalEnv is by default saved to the .RData file under the directory that R is started from but can be saved to completely different files as may be seen by looking at the help file for *save*. .RData is platform independent so a user working in windows can attach on .RData generated by R in linux, solaris or machintosh. The user can also give the command

$>$ load("SMB/.RData")

But that adds the data in SMB/.RData to the data in position 1 (.GlobalEnv) and it can be saved there in the end if the user wants to.

In Splus each object is stored as a seperate file, either in a default directory or in the .Data directory under the directory where $Splus$ was started if the command $SplusCHAPTER$ was once given there. Each $.Data$ directory does therefore correspond to $.RData$ but the operating system can be used to see what is in a $.Data$ directory but not what is within $.RData$ file. The storage method in $Splus$ does allow to make use of features of the operating system for manipulating objects but the method used in R makes the system less depended on the operating system.

If large objects are stored in .RData loading it can take a long time and if loading of the object is specified in .Rprofile each start of R takes a long time. Splus does on the other hand not load objects until they are used, when a CHAPTER containing the object is attached Splus does only note that an object with the name is there but it does not load it. The way to bypass this in R is to set up package , starting with the package.skeleton function, changing the file DESCRIPTION so "LazyData: yes" and the package is built with "R CMD INSTALL packgagdir". Binary packages made this way very little disc space or less than 10% of comparable Splus directory. They are on the other hand platform dependent so a seperate copy has to be made for every package.

The process of making the binary packages is very memory demanding.

In windows things work somewhat differently and an Splus CHAPTER can be created by giving the command *createChapter* in Splus. Also *Splus* must be started by giving a command like

```
splus.exe S_WORKING_DIR="c:/MySplusDirectory"
```

Also due to limitations of the operating system names of objects and names of files do not alway correspond to each other in windows.

R has the advantage of the windows and linux version being nearly identical while Splus is quite different under windows and linux.

3 Data frames

Dataframes are probably the most used data structures in R and correspond to tables in relational databases like Oracle. It is therefore no coincidence that commands that fetch data into relational databases return dataframes. In other terms a dataframe can be thought of as corresponding to worksheets in Excel. As Splus structure dataframe can be looked at as some kind of combination of the objects matrix and list where the main difference between a dataframe and a matrix is that in a datafram some columns can be numeric and some characters where a matrix is either all numeric or all character.

On the directory $r : /reikn/SplusNamskeid$ or $/nethome/u2/reikn/SplusNamskeid$ there are few dataframes and list containing surveydata. The same dataframes are also stored in a .RData file under the directory $r : /reikn/SplusNamskeid/R2007''$

utbrteg Information on catch in all the surveys.

STODVAR.all Information on all stations in the surveys

STODVAR List with the station information in all the surveys. From the Oracle tables fiskar.stodvar, fiskar.togstodvar and fiskar.umhverfi

hadkv2004 Haddock otholiths from the survey 2004 from the Oracle table fiskar.kvarnir.

hadle2004 Length measurements of the haddock in the survey 2004. From the Oracle table fiskar.lengdir.

hadnu2004 Number of haddock counted and measured from the survey 2004. From the Oracle table fiskar.numer.

lodna.jan92Acoustic measurements of capelin from January 1992

haustrall.all.stStations in the autumn survey

AfliBySquareMonthBritish catch in Icelandic waters by month and square from 1932 to 1934

Later it will be shown how those data can be obtained from the databases. To see what is in those dataframes the command *names* is used

```
>  names(utbrteg)
 [1] "synis.id"            "area"                "newstrata"
 [4] "ar"                  "lat"                 "lon"
 [7] "index"               "leidangur"           "ufsi.kg"
```

Here only the first 9 columns are shown but there are 95 columns. Most of the names are self explanatory, *lat* and *lon* are the average position in each tow i.e the mean of start and end of tow. *index* is an id of station that is comparable from year to year (reitur*100+tognumer), *area* is Bormicon area and *newstrata* is strata number in a stratification scheme.

synis.id is the column *synis_id* in the Oracle database *fiskar*, an unique index that connects the tables *fiskar.stodvar*, *fiskar.numer*, *fiskar.kvarnir* and *fiskar.lengdir*. The change of the name from *synis_id* to *synis.id* reflects difference between *Oracle* and *Splus* where a . in a name would not work in Oracle and and _ in a name would cause problem in *Splus* as in some version _ was allowed as and assignment operator so x_1 means the same as x <- 1. Therefore if a dataframe x has the column name synis_id x\$*synis_id* would not work but x\$"*synis_id*" would have to be used instead. It must be mentioned that *R* does not allow _ as an assignment operator and does therefore not have this stupid problem. To change names of column in dataframes in Splus a homemad routine skipta.texta has been written but it replaces _ with . in names.

```
> names(x) <- skipta.texta(names(x))
```

This change from _ to *dot* is not nessecary in R

4 Manipulation with the dataframes

As a first step few items regarding indexing of vectors will be mentioned, with an example as usually. In the examples following the expressions within the brackets are usually quitem complicated and the reader is encouraged to split them up to see what they are doing.

```
> x <- c(1,7,6,8,3,5,10,8,4,2)
> names(x) <- paste("a",1:10,sep="")
> x
 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
  1  7  6  8  3  5 10  8  4   2
```

We can select from the vector x in 3 different way. We show the example where we want to select items number 3 and 7.

```
> tmp <- x[c(3,7)]
> tmp <- x[ c("a3","a7")]
> ind <- c(F,F,T,F,F,F,T,F,F,F)
> tmp <- x[ind]
```

The last method deserves some further discussion and is the basis for a common method for selecting from dataframes. The vector *ind* in this command is a logical vector where each item is either T (TRUE) or F (FALSE). The length of the vector must be equal to the length of x and the command

```
tmp <- x[ind]
```

selects the items in x where the corresponding items of ind are T. If we want to select all items in x other than 3 and 7 we can write

```
> tmp <- x[-c(3,7)]
> tmp <- x[is.na(match(names(x),c("a3","a7")))]
> tmp <- x[!ind]
```

The second command could also be split up and a print command generated after each step. This kind of print commands should though not be used for large datasets except a care is to only print the first few values.

```
> ind <- match(names(x),c("a3","a7"))
> print(ind)
> ind <- is.na(ind)
> tmp <- x[!ind]
```

Examples are selection of data from the dataframe *utbrteg* but the $names(utbrteg)$ command can be used to list the column names of this dataframe. All those commands end on ...,] which means keeping all the columns, else we would write $..., c("col1","col2,"...,"lastcol")]$

```
# select data from 2004
> tmp <- utbrteg[utbrteg$ar==2004,]
#Select data from the year 2004 and area 2.
> tmp <- utbrteg[utbrteg$ar==2004 & utbrteg$area==2,]
#Select data from the years after 1999  and areas 2 and 3.
> tmp <- utbrteg[utbrteg$ar > 1999  & (utbrteg$area==2 | utbrteg$area==3),]
#Select stations where temperature is higher than 5 degrees
> tmp <- utbrteg[utbrteg$botnhiti > 5,]
```

When we try to list the data *tmp* from the last command we immediately note that something is wrong. The problem is that temperature is missing on a number of stations and there we have NA in the dataframe (Not available). The occurence of NA is tested with the function $is.na$ that returns $TRUE$ where there is a NA, else $FALSE$. To get rid of those stations where the temperature is not available the last command would have to be changed to

```
> tmp <- utbrteg[utbrteg$botnhiti > 5, & !is.na(tmp$botnhiti),]
```

Where ! is the not operator so while *is.na* returns $TRUE$ when there is a NA !*is.na* returns true when there is not a NA. But missing data are a common problem and need to be understood. Some functions have arguments like *na.rm* or *na.action* that are used to ignore the data with NA (usually the only thing that we can do).

As may be seen the command selecting data from many areas and many years can soon become cumbersome to write. There is an other way of selecting based on a combination of the *match* and !*is.na* commands

The match command gives us the position of each item of a vector in another vector, returning NA if the item is not found. Example

```
> match(c(8,10,11,2,9,5,4,9,1,7,11,5),c(5,11,4,7,6))
  NA NA  2 NA NA  1  3 NA NA  4  2  1
```

The first two numbers are not found in the latter vector but the number 11 is number 2 in the latter vector. What we are looking for when selecting from a dataframe are most often the cases where match does not return NA.

Example

```
> tmp <- utbrteg[!is.na(match(utbrteg$ar,c(1985,1987,1993,1999)),]
or
> tmp <- utbrteg[utbrteg$ar %in% c(1985,1987,1993,1999),]
```

Selects all records from the years 1985, 1987, 1993 and 1999 but

```
> tmp <- utbrteg[is.na(match(utbrteg$ar,c(1985,1987,1993,1999))),]
```

More complicated example is

```
> tmp <- utbrteg[!is.na(match(utbrteg$ar,c(1985,1987,1993,1999)) &
  !is.na(match(utbrteg$area,c(1,9,10))),]
or
> tmp <- utbrteg[utbrteg$ar %in% c(1985,1987,1993,1999)) &
  utbrteg$area %in %c(1,9,10))),]
```

The syntax of these commands is rather inattractive (better if %inused) but after years of use one gets used to it. Writing a function that removes the need for this syntax is though not complicated.

5 Commands to sort data

There are two commands to sort data, *sort* and *order*, the former sorting the data in ascending order while the second one gives the position of each data point in the sorted vector. Example

```
> x  <- c(4.42,6.12,5.60,4.43,5.28,3.46,6.52,6.16,4.99,4.78)
> sort(x)
 [1] 3.46 4.42 4.43 4.78 4.99 5.28 5.60 6.12 6.16 6.52
> order(x)
 [1]  6  1  4 10  9  5  3  2  8  7
> x[order(x)]
 [1] 3.46 4.42 4.43 4.78 4.99 5.28 5.60 6.12 6.16 6.52
```

As may be seen $x[order(x)]$ is the same as $sort(x)$.

The data can be sorted in descending order by using $-x$ instead of x

```
> -sort(-x)
 [1] 6.52 6.16 6.12 5.60 5.28 4.99 4.78 4.43 4.42 3.46
> order(-x)
 [1]  7  8  2  3  5  9 10  4  1  6
```

The commands are often used to look at the N highest values in a table or sort according to the value in some column like

Getting the three highest values is done by

```
> -sort(-x)[1:3]
[1] 6.52 6.16 6.12
```

Saving the 10 tows where most cod was caught in a seperate dataframe

```
> tmp <- utbrteg[order(-utbrteg$torskur.kg)[1:10],]
> tmp[,c("ar","index","torskur.kg")]
       ar index torskur.kg
3022 1985 71812  17462.857
1698 1988 66815  15356.517
1327 1989 61904  14661.009
8365 1996 67425  14131.721
7544 1998 67114  13246.365
1564 1989 41201  12217.978
7321 1998 71701  12123.578
6133 1989 41213  11246.991
7625 1998 41204  10872.913
3576 1993 71812   9766.935
```

6 Few methods to tabulate or group data.

table Split a dataframe according to the value of listed column and count the number of occurences in each category.

tapply Split a dataframe according to the value of listed column and apply a specified function to each part. tapply does the same as table if the functions length (or count) are used except it returns NA where there are no values but table returns 0.

apply.shrink Same as tapply but different format of result

apply.shrink.dataframe Extension of apply.shrink

combine.rtSum up data in grid cells, padding with zeros

Of those combine.rt is the fastest but least useful. It calls a C program and is used in routines to manipulate logbookdata. Of the programs *tapply* and *table* are included with Splus and R but *apply.shrink* and *apply.shrink.dataframe* are home made additions.

Examples of use. First the total total number of stations each year is found, then the median of the catch of cod each year and then the mean catch of cod each yera for each type of tow but there are 3 types, selected randomly, selected by captains and added in later surveys.

```
> x <- table(utbrteg$ar)
# the command rep(1,nrow(utbrteg)) makes a vector full of 1 of length
# same as number of row in utbrteg
> x1 <- tapply(rep(1,nrow(utbrteg)),utbrteg$ar,length)
> x2 <-apply.shrink(rep(1,nrow(utbrteg)),utbrteg$ar,length,
  names=c("ar","fj.stodva"))
> x3 <- apply.shrink.dataframe(utbrteg,"synis.id","ar","length")
```

In the last command any column can be given as argument 2 as the function length is applied and all columns have the same number of record each year. But it must emphasized that all the 4 commands shown before do exactly the same thing.

```
> x <- tapply(utbrteg$torskur.kg,utbrteg$ar,median)
> x1 <- apply.shrink(utbrteg$torskur.kg,utbrteg$ar,
  median,names=c("ar","torskur.med"))
> x2 <- apply.shrink.dataframe(utbrteg,"torskur.kg","ar",median)
```

Then the more complicated example splitting according to two columns i.e *ar* and *togtegund*.

```
> x <- tapply(utbrteg$torskur.kg,list(utbrteg$ar,utbrteg$togtegund),mean)

> x1 <- apply.shrink(utbrteg$torskur.kg,list(utbrteg$ar,
       utbrteg$togtegund),mean,names=c("ar","togtegund","torskur"))

> x2 <-  apply.shrink.dataframe(utbrteg,"torskur.kg",c("ar","togtegund"),mean)

# Hear cod, haddock and saithe are all run at the same time.
>  x3 <-
apply.shrink.utbrtegframe(utbrteg,c("torskur.kg","ysa.kg","ufsi.kg")
```

```
,c("ar","togtegund"),c(mean,mean,mean))
```

The output from tapply is OK when splitting according to two columns but splitting according to more columns will generate multidimensional arrays that are difficult to manage while in apply.shrink and apply.shrink.dataframe additional columns will be added to the result that will still be easy to look at.

The function *tapply* can be used with functions that return more than one number but *apply.shrink* and *apply.shrink.dataframe* do not extend so easily. An example could be if we want to find the 10 largest tows each year. Then we write a function that find the 10 largest values of a vector.

```
> nlargest <- function(x,n)return(-sort(-x)[1:n])
> x <- tapply(utbrteg$torskur.kg,utbrteg$ar,nlargest,n=10)
```

In this case tapply returns a list but *apply.shrink* and *apply.shrink.dataframe* do not work.

7 Graphics in Splus and R

This section describes graphics in Splus and R and is more thought of like an introduction to the section on the geo library. Graphics in Splus and R are usually reasonably similar with the exception of the *image* function and definition of color schemes is very different in *R* and *Splus* and in *Splus* color schemes in *windows* are very much different from what they are on *unix* systems.

In Splus in unix systems a color scheme on the screen does not need to be the same as what applies on postscript printer while in Splus for windows the correspondence is reasonable.

Color schemes in R are much better than in Splus and there is a perfect agreement between linux and windows and between postscript files and graphics terminal.

Color schemes that have been used in the geo library are now implemented in R (see the functions colps, bwps, geoplotpalette, and geoplotpalette). The

8 Geo library

A set of routines written 11 years ago for Splus. The programs had to be changed somewhat when rewritten for R as many of the graphical parameters that can be set in R but not in S. Some cleaning of the code was done when it was transferred to R. Other and probably better mapping software is the PBSmapping library from Nanimo which is more in line with the GMT program. Both the PBSmapping library and the GMT program might be very useful at the MRI.

Many of the functions in the geo library are similar to functions in Splus with the name geo put in front. The parameters can though sometimes be quite

different and some graphical parameters are not allowed as the ... syntax has not been implemented (and will not)

Most of the geo routines take a vector of lat and a vector of lon as arguments but a dataframe or list with two of the colnames lat and lon can also be used. In some cases other colnames than lat and lon can be used but they do then have to be specified. For comparison the plot command takes a vector of x and a vector of y or a dataframe with the component names x and y.

geoplot	plot	For setting up the map
geopoints	points	adding points on a map
geolines	lines	adding lines on a map
geosymbols	symbols	adding symbols on a map
geotext	text	adding text on a map
geopolygon	polygon	adding a polygon on a map
geolocator	locator	getting positions on the map by clicking the mouse
geoidentify	identify	identifying a point in a dataset be the mouse
geoarea		calculating size of an area
geoaxis	axis	plot axis if geoplot is called with axlabels=F
geocontour	contour	plotting contour lines
geocontour.fill		plotting filled countours
geodefine		defining a closed area
geointersect		calculating the intersection or union of two polygons
gbplot		plotting depth contours
geoworld		plotting coastlines
geoexpand	expand.grid	expanding a grid
geoimage	image	plotting an image (rarely used).
geoinside		find the points inside a polygon
pointkriging		surface fitting using kriging
variogram		calculations of variograms
variofit		fitting of variogram
SMB.std.background		setting up plot that fits the SMB area
arc.dist		geographic distance
lodist		geograpich distance using the mouse
litir		plot a chessboard that shows the colors

The use of these programs is best described by an example

```
# Cod Catch 2004
> SMB.std.background(grid=F)
> i <- utbrteg$ar== 2004
> gbplot(c(200,500),col=25,lwd=2)
> labloc <- list(lat=c(63.95,65.4),lon=c(-19.8,-17.3))
> geosymbols(utbrteg[i,],z=utbrteg[i,"torskur.kg"],circles=0.2,sqrt=T,
> levels=c(10,100,500),label.location=labloc)
> geopoints(utbrteg[i,],pch=17,csi=0.08)
>
> # Plot filled circles with the same
```

```
> SMB.std.background(grid=F)
> i <- utbrteg$ar== 2004
> gbplot(c(200,500),col=25,lwd=2)
> labloc <- list(lat=c(63.95,65.4),lon=c(-19.8,-17.3))
> geosymbols(utbrteg[i,],z=utbrteg[i,"torskur.kg"],sqrt=T,
> levels=c(10,100,500),fill.circles=T,colors=c(0.02,0.04,0.06,0.1),
> label.location=labloc)


# As contours but not circles geoaxis used

> grd.smb <- list(lat=seq(62.8,67.5,length=50),lon=seq(-28,-10,length=80))
> SMB.std.background(grid=F)
> geogrid(grd.smb)
> geopoints(STODVAR$y2004)


> tmp <- utbrteg[utbrteg$ar==2004,]
> vg <- list(nugget=0.1,sill=1,range=0.4) # variogram
> zgr  <- pointkriging(tmp$lat,tmp$lon,tmp[,"torskur.kg"],
  grd.smb,vg,maxnumber=12,set=-1,maxdist=30)
> hist(zgr) # look at distribution of zgr
> levels <- c(10,50,100,200,500)
> col <- c(0,3,27,52,100,150)
> SMB.std.background(grid=F,axlabels=F,dlat=1,dlon=2)
> geoaxis(side=2,dlat=1,inside=F) # y axis
> geoaxis(side=1,dlon=2,inside=F) # x axis
> geocontour.fill(zgr,white=T,levels=levels,col=col)
> geopolygon(island,col=0);geolines(island)
> geopolygon(gbdypif.500,col=0,exterior=T)
```

9 selection of data from the databases at MRI

A number of routines are available to select data from the databases at the MRI. These routines have been written around a Oraperl program called sql++ which is only available on unix systems so the routines do not work in windows. The most recent version of *Splus* for *windows* which has just been set up at the MRI does though have more possibilities for generating sql commands so the routines should also work in windows withing not too long time and the same applies to *R* for *windows*

```
hadnu2004 <- lesa.numer(STODVAR$y2004$synis.id, 2)
hadle2004 <- lesa.lengdir(STODVAR$y2004$synis.id, 2)
```

```
hadkv2004 <- lesa.kvarnir(STODVAR$y2004$synis.id, 2,
      names = c("kyn","kynthroski", "oslaegt", "slaegt"))
```

10 Programs to work with logbook data