engine=R

Working With Data

Laurence Kell mail to:laurie.kell@iccat.int
Population Dynamics Expert,
ICCAT Secretariat, C/Corazón de María, 8. 28002 Madrid, Spain.

1 R Data Objects

R has a rich variety of data types such as vectors, matrices, arrays, data frames and lists, as well as data in S4 classes and objects. However, before you can take advantage of R and contributed packages you have to be able to import your data that have been stored in text files, spreadsheets, databases or binary files. Once you have created data objects in R it will often be necessary to convert them from one type to another (i.e. coercion) to summarise them, and to subset and transform objects often by writing R functions. Therefore in this document we describe

1. the main R objects
2. how they map to FLR objects and
3. external data types and how they can be imported into R
4. how to manipulate R objects

¡¡results=hide, echo=FALSE¿¿= require(FLCore) @

2 R Data Objects

The main R data objects are vectors, matrices, arrays, data.frames and lists. See the R language definition for a complete list and documentation and the R reference card for a summary of how to use them.

Vectors are continuous cells containing data of a single type. Types can be of character (i.e. string), integers, doubles, logical, raw (or bytes) or complex. Matrices and arrays are similar to vectors but with attributes for the dimensions (dim) and, optionally, the dimension names (dimnames); a matrix having 2 dimensions (i.e. row and column) and arrays 2 or more dimensions.

A list is a collection of objects that may be of any type and length. Data frames are similar to data sets as used by statistical software such as SAS, in that they contain variables (i.e. columns) by observation (i.e. by row). The columns may contain vectors, factors, and/or matrices all having the same length (number of rows in the case of matrices). In addition, a data frame generally has a names attribute labelling the variables and a row.names attribute for labelling the observations. A data frame can also contain a list of the same length as the other components. It is worth mentioning the model.frame method (or function) which in certain packges returns a data.frame with the variable needed to use the formula in a model, such as a linear regression, stored as columns.

R is an object oriented language. It is not important that you know exactly what this means, however, an important concept is that everything in R is an object and that every object belongs to a class. What

class an object belongs to will determine what operations can and cannot be performed on it and what a generic method like plot() or or summary() will produce. For this reason it is important to know how to find the appropriate help and how to determine which class the object you wish to manipulate belongs to. Some basic help calls are shown below

```
?aov                  #  specific help - if you know the name of the function
args(aov)             #  gives the specific arguments to the function
help.search("plot")   #  if you only know the subject area
apropos('lm')         #  returns a character string of all potential matches
??plot                #  returns the package and all available methods
help.start()          #  launches the online html help.
example(array)        #  returns the example code provided in the help doc
find('par')           #  will tell you what package something is in
search()              #  returns a list of attached packages
```

Methods for finding out what class a particular object belongs to, what type of data it contains and what storage mode is used to store the information are also important. As an example we generate some random data and use it to create an array. You will see that the storage mode of the objects contents is 'numeric' whilst the specific data type is 'double'. The object oriented approach allows simple classes to be extended to form more complex classes. The 'is' method can be used to view the full class inheritance tree of the object.

¡¡¿¿= x ¡- array(rnorm(9),dim=c(3,3), dimnames=list(row=c("x","y","z"),col=1:3)) what class does the object x belong to class(x) typeof(x) mode(x) is(x) @

When working in an R Session you will want to occasionally save code as you are working. You can recall the code, i.e. the commands, that you have been running in your session by typing history(). By default only 25 lines are shown, you can also save your entire history to a file and even run previous commands. To find out more

¡¡results=hide, echo=TRUE¿¿= ?history @

¡¡results=hide, echo=TRUE¿¿= source(C:
mydir
mine.R ") @

2.1 FLR Objects

Without getting into a detailed explanation of Object Oriented Programming (OOP) FLR is based on the R S4 classes which are based on OOP. An S4 class contains structures composed of certain types of data and methods or functions that manipulate the classes. The fundamental data structure in FLR is the FLQuant which is an extension of a 6 dimensional array (where the dimensions correspond to age, year, season, unit, area and iteration). An FLQuant is used to represent data such as stock numbers, stock weights, fishing mortality, in FLR classes, for example in FLStock that represents a stock. Any FLR object can readily be converted to another through coercion, an R mechanism where one R data type can be converted to another type. This means that once you understand the basic methods required to manipulate objects in R you will have great flexibility about what you are able to do with data within

¡¡results=hide, echo=TRUE¿¿= @

3 R Data Types

3.1 Vectors

R is a vector orientated language and the following assignment creates a vector of length 5. The assignment symbol ¡- works in the same way as =. Typing only the name of the vector outputs its values. Calculations are performed on all elements of a vector thus the square of all elements in vector b can be computed simply by passing the vector b to the appropriate method as shown below.

Helvetica looks like this and Palatino looks like this

```
b = 1:5
length(b)
b
b^2
```

A vector must contain elements of the same type, e.g. a vector of doubles as before. The vector oriented approach of the the R language means that functions such as log will be applied to all elements in the vector passed as an argument e.g.

```
log(b)
```

Since R is an object orientated language it knows how to do things like plot any object, since it uses methods, i.e. generic functions that behave appropriately depending on the object passed as an argument.. In this case a trivial exercise but with more complicated objects a useful feature.

```
plot(b)
```

R is case sensitive and the correct case must be used when calling functions and methods and referencing variables. Variables names can contain letters, digits, ˙ and _, however the variable name must start with a letter. Recycling is an important feature of vector arithmetic. When two vectors of unequal length are combined, the elements of the shorter vector are reused. In the example below, after the third element in a, the first element in b is reused. This process is known as automatic recursion and can be a very powerful tool in R. But beware of it. R will use automatic recursion and will assume that the user is aware of it. If the length of the smaller object is a multiple of the larger object, R will use automatic recursion. If the length of the smaller object is not a multiple of the larger object, R will still perform the action but will give a warning that the sizes of the two objects are not compatible.

```
# recycling
a <- 1:6
b <- 1:3
a + b
c <- 1:4
a +c
```

A variety of missing values can be specified in R. Values can take Inf or -Inf for infinity, NA for a missing value and NaN for not a number. Dividing a positive number by 0 will produce Inf value and dividing a negative number by 0 will produce a -Inf value. NaN values can be created by trying to calculate the log of a negative number, for example.

```
a <- c(3, 4, NA, Inf)
a
tt <- c(1, NA, 3, 4, NA, 6, NaN)
is.na(tt)
is.nan(tt)
```

The sum of vector a will in the example above be defined as NA, since NA is propagated. The na.rm argument will remove the NAs and then sum the valid values in the vector.

```
sum(a)
sum(a, na.rm = T)
```

Similar rules apply for string manipulation, for which additional methods such as paste are available

```
b <- c("yes", "no", "maybe")
paste("is it", b)
paste("is it", b, sep = ",")
paste("is it", b, sep = " ", collapse = ", ")
```

A variety of methods exist for creating objects with different characteristics for example rnorm creates random numbers

```
a <- rnorm(100)
```

The vector can be summaried by calculating it s mean and standard deviation and plotted as a histogram. Many basic functions exist in R to create a variety of summary statistics. In addition a number of basic plotting functions are also available for quick and easy data visualisation e.g.

```
mean(a)
sd(a)
plot(a)
hist(a)
boxplot(a)
boxplot(a, col="red")
qqnorm(a)
qqline(a)
fivenum(a)
stem(a)
```

max() and min() return the maximum and minimum values in a vector, and pmax and pmin perform a pairwise comparison, i.e. comparing the vector with 0.0.

```
max(a)
pmax(a, 0)
```

Comparing elements of vectors can be done with ==, and ! is the logical inverse

```
c(T, F, T, T, F, F) == TRUE
!c(T, F, T, T, F, F)
```

A vector can be subset using logical values or using a comparison operator, in this instance to get all values greater than two or by sub-setting the vector by its index, i.e. to get the first three elements

```
a <- c(T, T, F, F, T)
b <- 1:5
b[a]
b[b > 2]
b[1:3]
```

Alternatively by giving a list of numbers, or by excluding an element

```
b[c(1, 2, 1, 3, 4, 1, 1, 5)]
b[-3]
```

Vectors of one type can also be converted to vectors of other types through coercion, i.e. from integer to logical

```
b <- c(0, b)
lb <- as.logical(b)
lb
```

3.2 Matrices and Arrays

Matrices and arrays are vectors with attributes that define the dimensions (dims) and optionally the dimension names (dimnames), recalling two-dimensional array (i.e. matrix) x that we created before use the following

```
# create an array
x <- array(rnorm(9),dim=c(3,3),
                    dimnames=list(row=c("x","y","z"),col=1:3))
```

The matrix function takes an argument that determines how the matrix is filled, i.e. by row or by column. Note that by default a matrix is filled by column, you can change the way that the matrix is filled by setting the argument 'byrow' to TRUE. The kind of data structure (i.e. number of rows and columns) is determined by the attributes of the object.

```
# dimensions
dim(x)
dimnames(x)
attributes(x)

# alternatives for creating an array
a <- matrix(1:16, nrow = 4)
aa<- matrix(1:16, nrow = 4, byrow = TRUE)
a
aa
```

It is possible to remove dimensions, for example to get a vector rather than a matrix, by changing the attributes of the object. A matrix can be subset to obtain specific values, i.e. a vector defined by row or column A vector can also be reshaped to get a higher dimensional array, note that you cannot go out of a vector s bounds, since this is protected

```
# changing attributes
dim(a) <- NULL
attributes(a)
a

attr(aa, "dim") <- c(2, 4, 2)
attributes(aa)
aa

aa[2, 3,1]
aa[2,,]
aa[1:2,, ]
```

2.3.Lists

Lists are simply collections of objects. The objects contained in any given list can be of different types. In this example we create a list of length 3 containing string, integer and logical types.

```
fish <- list(name = "cod", age = 3, male = FALSE)
fish
```

You can retrieve an element of a list in different ways, either by name or by specifying its index using square brackets.

```
fish$name
fish[["name"]]
fish[[1]]
```

Note how we use double square brackets to index the list. Indexing a list with single square brackets will return an object of class list that has length 1 and contains the element of the list that we have requested. When using double square brackets we are returned an object in the class of the contents of the element of the list that we are accessing.

```
class(fish$name)
class(fish[["name"]])
class(fish[[1]])
class(fish[1])
```

So, using double square brackets returns the original type, whilst single square brackets returns a list. You can also create arrays of type list e.g.

```
array(list(), c(2, 3))
```

2.4.Data frames

In a data.frame all vectors need to be of the same length, this is equivalent to a SAS data set or simple Excel sheet. First we create a list with equal vector length, then make the data.frame.

```
a <- list(age = 1:10, weight = c(0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.6, 0.7, 0.9))
a
b <- as.data.frame(a)
b
```

It is also possible to make data.frame in one go and because of recycling you don t need to replicate all elements.

```
fish <- list(name = "cod", age = 3, male = FALSE)
fish
```

You can subset a data.frame like you would a list i.e. to get a vector

```
fish$name
fish[["name"]]
fish[[1]]
```

Sometimes it will be useful to determine what class an object belongs to and to add comments.

```
class(fish$name)
class(fish[["name"]])
class(fish[[1]])
class(fish[1])
attr(fish, "rem") <- "This is my dataframe"
```

You can edit a data.frame as you would spreadsheet using fix and import and export dataframes, by default if no path, then file is put in setwd() dir

```
args(fix)
write.table(b, )
args(write.table)
args(read.table)
```

2.5. Coercion Coercion is the act of creating one type of R object from another, this can be changing the basic type, i.e. a string into a number, or a matrix into a data.frame.

```
as.double("2")
as.character(2)
df <- data.frame(string = c("eeny", "meeny", "miny", "mo"), integer = 1:4)
as.matrix(df)
```

Coercion just requires sticking an as. in front of the type you want to create e.g. for both the atomic types (i.e. vectors of type integer, double,..) This also works with FLR objects, even though these are quite complex classes. This means that you can think about what you want rather than how to produce it. For example load FLR and the example FLStock object ple4

```
library(FLCore)
data(ple4)
# what is it?
is(ple4)
summary(ple4)
plot(ple4)
# coerce into a data.frame
head(as.data.frame(ple4))
# coerce into a model.frame
head(model.frame(ple4))
```

You can also do this for the individual slots in the FLStock head(as.data.frame(stock.wt(ple4))) The reverse also works, so if you already have data in R you can create an FLR slot or object.

```
as.FLQuant(data.frame(age = 1:10, data = 0.2))
as.FLQuant(data.frame(expand.grid(age = 1:10, year = 1990:2010),
data = 0.2))
```

3.3 Lists

3.4 Data frames

3.5 Coercion

4 Reading Data into R

4.1 Text

Very often the data that you want to analyse will be available in a file format that is supported by another piece of software such as a spreadsheet, a database or another statistical package. There are a variety of methods for importing data of many different formats into your R session. In this section we will consider the most commonly used methods which include comma or space separated data saved as a flat text file and accessing data from excel spreadsheets. A number of packages have been developed to enable import of data from other formats. The 'foreign' package allows a number of alterrnative data formats to be read into R including SAS formatted data and SAS transfer files, and the package 'RODBC' provides database functionality. The most commonly used methods are read.table and read.csv, although your data do not necessarily have to be comma separated. Spaces, tabs and any other form of separator may be used, but they must be used consistently. Files containing a combination of different separators will be very difficult to read using the methods below. There are a number of functions provided in the base distribution of R for importing data into your work session. These include scan, readLines and read.table. Perhaps the simplest method is to use scan which simply reads all of the data contained in the file into a single vector. The data can then be coerced into the required type as shown above. The readLines method will do something very similar but will read in only the specified number of lines. The read.table method is particularly useful. It returns a data.frame object containing the data. If the header is specified as TRUE then the first line of data to be read in will be used for the column names in the returned data.frame object. read.csv() is similar to read.table and is basically a wrapper that defines the seperators ad , and sets header=TRUE by default e.g. to read in the catch distribution data. You will need to change the file path given in the example below to the location of the file on your machine.

```
# dirData<-"..."
## read.csv
fileCdis <-paste(dirData,"\\csv\\cdis.csv",sep="")
cdis
<-read.csv(fileCdis)
head(cdis)
```

read.table() can also read from http and fttp, e.g. the NAO data on the NOAA website Read table accepts web addresses too, This can be used to read in data from the internet

```
nao <-read.table("http://www.cdc.noaa.gov/data/correlation/nao.data", skip=1,
nrow=62,
na.strings="-99.90")
```

scan() is more flexible but requires a bit more effort, here we read in data from a file that holds the catch-at-age data for mediterannean swordfish and create an array with appropriate dims and dimnames

```
fileSwo<-paste(dirData,"\\VPASuite\\swoCN.dat"
yrs <-scan(fileSwo,skip=2,nlines=1)
ages<-scan(fileSwo,skip=3,nlines=1)
dat <-scan(fileSwo,skip=5)
caa <-array(dat,dim
=c(diff(ages)+1, diff(yrs)+1),
dimnames=list(age =ages[1]:ages[2],year=yrs[1]:yrs[2]))
```

4.2 Spreadsheets

As is almost always the case with R, there are several different ways to perform the same task. Data can be read in from an Excel spreadsheet using one of a number of methods. The packages RODBC, xlsReadWrite, xlsx and gdata all contain routines for accessing data in spreadsheet form. In the example below we use the read.xls method available in the gdata package to read data from the excel file swordfish.xls. This is because it uses the same defaults as for read.csv() making it easier to use. You will need to change the file path given in the example below to the location of the file on your machine. If you haven't already installed the gdata package you will need to do this first, and also install perl

```
install.packages('gdata')
library(gdata)
yrs <-read.xls("swordfish.xls",sheet="CatchN",skip=1,nrow=1)[1:2]
ages <-read.xls("swordfish.xls",sheet="CatchN",skip=2,nrow=1)[1:2]
dat <-read.xls("swordfish.xls",sheet="CatchN",skip=5,header=F)
caaXl<-array(dat,dim
=c(diff(ages)+1, diff(yrs)+1),
dimnames=list(age =ages[1]:ages[2],year=yrs[1]:yrs[2]))
is(ages)
is(yrs)
is(dat)
ages<-unlist(ages)
yrs <-unlist(yrs)
dat <-t(as.matrix(dat))
caaXl<-array(dat,dim
=c(diff(ages)+1, diff(yrs)+1),
dimnames=list(age =ages[1]:ages[2],year=yrs[1]:yrs[2]))
caaXl
```

As you can see, the example given above counts the number of sheets in the excel file, obtains the names of each of those sheets and finally accesses the data in the second sheet, ommitting the first four rows of data and extracting the following 5 rows of data. The results are returned as a data.frame to the object we have called xlsdata.

4.3 Databases

Databases prove more flexibility Accessing a database, since you can access different tables and views. Where a view is a virtual table, i.e. it is constructed from other tables.

```
chlT3sz   <-odbcConnectAccess(fileT3sz)
sqlTables(chlT3sz)
t3sz     <- sqlQuery(chlT3sz, "select * from [t2szFreqs]")
head( t3sz)
names(t3sz)

head(sqlQuery(chlT3sz, "select * from [Species]"))
head(sqlQuery(chlT3sz, "select * from [t2szFreqs]"))
head(sqlQuery(chlT3sz, "select * from [t2szProcs]"))
head(sqlQuery(chlT3sz, "select * from [t2szStrata]"))
head(sqlQuery(chlT3sz, "select * from [cat_szDetail]"))
head(sqlQuery(chlT3sz, "select * from [cat_szSummary]"))
head(sqlQuery(chlT3sz, "select * from [subSets]"))
head(sqlQuery(chlT3sz, "select * from [t2szProcs Query]"))
```

An example of creating a database and making a query

```
library(RSQLite)
## Initialise the SQLite engine
SQLite()
## connect DB
dbMC <-"c:/temp/t.dbf"
conMC <-dbConnect(dbDriver("SQLite"), dbname=dbMC)
df<-data.frame(n=rep(1:4,25),a=rep(c("a","b","c","d"),each=25))
dbWriteTable(conMC,"EastMC",df,append=TRUE)
dbListTables(conMC)
file.info(dbMC)
query<-"SELECT * FROM 'EastMC' WHERE n IN (2) AND a IN ('a','b') LIMIT 10"
x
<- dbGetQuery(conMC, query)
```

A utility function for database queries

```
setGeneric("sqlVar", function(object, ...)
standardGeneric("sqlVar"))
setMethod("sqlVar", signature("character"),
function(object, ...) paste("('",paste(object,collapse="','"),"')",sep=""))
setMethod("sqlVar", signature("numeric"),
function(object, ...) paste("(",paste(object,collapse=","),")",sep=""))
sqlVar(c("b"))
sqlVar(2)
query<- paste("SELECT * FROM 'EastMC' WHERE n IN", sqlVar(2), "AND a IN", sqlVar(c("a","b")), "LIMIT
10")
x
<- dbGetQuery(conMC, query)
dbDisconnect(conMC)
```

3.4.Importing Data into FLR objects Once you have grasped the concepts above, importing data into FLR objects becomes easier. It involves reading data in from an external source and coercing it to a form compatible with the creation of an FLR object. The basic FLR object is the FLQuant; essentially a 6 dimensional array with named dimensions. You can create FLQuant objects from a variety of data types, vectors, arrays, matrices, data.frames and so on. In the next example below we access the monthly NAO (North Atlantic Oscillation) data set from the NOAA website, store the data in an FLQuant and produce a plot of the data. Read table accepts web addresses too. This can be used to read in data from the internet

Note in the example above how we manipulate the nao data.frame object before using it to create the FLQuant object. We first subset the data using [,-1] to remove the first column of data and then use unlist to coerce it to a numeric vector before passing it to the FLQuant constructor method.

```
# wide format
dirMy<-

args(read.table)
wid <- read.table(file=paste(dirMy,'data_wide.dat',sep="/"), sep="\t", header=TRUE)
wid

tabq <- FLQuant(as.matrix(wid)[,-1], dimnames=list(age=wid[,1], year=substr(names(wid)[-1],2,5)), units='t')

# data.frame in FLQuant-friendly format
flq <- FLQuant(rnorm(200), dimnames=list(age=1:10, year=1990:1999), units='t')

# as exported by as.data.frame
# - column named 'data' for numbers in array
head(as.data.frame(flq))

# but file can be reduced to only columns with information
lon <- read.table(file=paste(dirMy,'data_long.dat',sep="/"), sep=",", header=TRUE)
args(read.csv)
lon <- read.csv(file=paste(dirMy,'data_long.dat',sep="/"), sep=",", header=TRUE)

flq <- as.FLQuant(lon, units='t')

# data in VPA Suite format
dat <- read.table(file=paste(dirMy,"VPASuite\\swoCN.dat",sep=""),skip=5)
yrs <- read.table(file=paste(dirMy,"VPASuite\\swoCN.dat",sep=""),skip=2,nrows=1,what=numeric())
ages<- read.table(file=paste(dirMy,"VPASuite\\swoCN.dat",sep=""),skip=3,nrows=1)
flq <- FLQuant(t(as.matrix(dat)), dimnames=list(age=ages[1,1]:ages[1,2], year=yrs[1,1]:yrs[1,2]))

dat <- scan(file=paste(dirMy,"VPASuite\\swoCN.dat",sep=""),skip=1)
flq <- FLQuant(dat[-(1:7)], dimnames=list(age=dat[5]:dat[6], year=dat[3]:dat[4]))

readVPAFile(paste(dirMy,"VPASuite\\swoCN.dat",sep=""))

# data in wide format for FLStock
stk <- read.table(file=paste(dirMy,'stock_data_wide.dat',sep="/"), header=TRUE, sep='\t')

# create a list
lst <- FLQuants()
# and loop through sections of stk to fill it
for (i in 3:4){
  df <- stk[,-i]
  names(df)[3] <- 'data'

  # with FLQuant objects
  lst[[names(stk)[i]]] <- as.FLQuant(df)
  }

# call FLStock creator with list elements as arguments
sto <- FLStock(catch.n=lst[['catch.n']], catch.wt=lst[['catch.wt']])

# trick: use list as argument for do.call
sto <- do.call('FLStock', lst)

# functions exist for loading data in VPASuite, Adapt, ICA, MFCL and VPA2Box formats

# VPA Suite or LWT format

cod4 <- readFLStock("cod_2007/Cod347.idx", tyep="VPA")

summary(cod4)

# Landings and discards added together in this object, so need to compute manually

# Swap the landings and catch over
catch.n(cod4)     <- landings.n(cod4)

# Read individual files for landings and discards
landings.n(cod4)  <- readVPAFile(paste(input.dir,"Cod347cn_L.dat",sep="/"))
discards.n(cod4)  <- readVPAFile(paste(input.dir,"Cod347cn_D.dat",sep="/"))

# Do the same for the weights
catch.wt(cod4)    <- readVPAFile(paste(input.dir,"Cod347cw_T.dat",sep="/"))
landings.wt(cod4) <- readVPAFile(paste(input.dir,"Cod347cw_L.dat",sep="/"))
discards.wt(cod4) <- readVPAFile(paste(input.dir,"Cod347cw_D.dat",sep="/"))

# And the same for the totals
catch(cod4)    <- readVPAFile(paste(input.dir,"Cod347la_T.dat",sep="/"))
```

5 Importing Data into FLR objects

5.1 Stock Assessment Files

A particular stock assessment method is generally based upon a single software package with specific input and output files. These files are usually in text format, which can result in rounding errors and difficulty in replicating results. Although in some case (VPA2Box) binary files are also used. During a stock assessment meeting there are many occassion when these files need to be read into R to produce additional summaries or plots, to conduct further analyses or compare results from different assessment methods. To make these tasks easier there are a variety of methods in FLR to read in files from stock assessment software and FLR objects and data.frames.

There are generally six types of quantities found in assessment files

- Input data;
- Assessment assumptions including any priors used
- Parameter estimates
- Bench marks or reference points
- Time series of historic stock status and exploitation levels
- Projected outcomes for the stock and fishery under different management regimes

For the more complex models such as Stock Synthesis and Multifan-CL this results in many files; and so the authors to make life easier have developed R code to read the results. See ? & ? for more details. Even so it can be difficult to extract the required quantities and so in FLR there are various methods to create objects for use within FLR.

The FLR methods return either FLR objects (e.g. FLQuant, FLStock or list of these e.g. FLQuants, FLStocks or FLlst) or data.frames (again either as seperate objects or as lists). The FLR object is prefered as then the many FLR methods can then be used to summarise and conduct further analysis on the objects produced.

Syntax of the methods is of the form

```
readXXXX(x,scen=NULL,type=NULL,data.frame=FALSE,msy=NULL)
```

Where x is either a file or directory, type refers to a particular type of output dependent on the software package that the files are designed to be used with. In many cases the extension or an internal flag indicates what type of file the target file is and this is used by default to determine what will be read by the method, although the type argument can overide this and data.frame determines whether the method returns an FLR object or a data.frame. If scen is specified then the first arg represents a directory rather than a file and scen represents the different files. For example in the case of projections where different assessment runs maybe projected for different hypotheses (e.g. about future recruitment) and different management options (e.g. TACs).

MSY specifies whether results such as biomass, harvest rate and catch are scaled by MSY based reference points or proxies for them, by default the results are ?

5.1.1 ASPIC

ASPIC is an biomass dynamic model, which uses age aggregated data, it can also perform projections for different TACs [and Fs?]. There are six types of files

.bio, bootstrap estimates of historic biomass and harvest rate

.prj bootstrapped projections with predicted biomass and harvest rates

.det

.inp

.prb

.bot

```
library(FLAdvice)

### Assessments
## 1 file
aspic<-readASPIC(paste(dirAspic,"/",scen=scen[1],".bio",sep=""))
class(aspic)
names(aspic)

aspic<-readASPIC(paste(dirAspic,"/",scen=scen[1],".bio",sep=""),data.frame=T)
class(aspic)
names(aspic)

## many files
aspics<-readASPIC(dirAspic,scen=scen,type="b",data.frame=T))
```

5.1.2 ASPIC

```
#### Projections
## 1 file
prj<-readASPIC(paste(dirAspic,"/","bumcont1bproj500",".prj",sep=""))
class(prj)
names(prj)

prj<-readASPIC(paste(dirAspic,"/","bumcont1bproj500",".prj",sep="",data.frame))
class(prj)
names(prj)

## many
prjs<-readASPIC(dirAspic,scen=expand.grid(scen=c("bumcont1bproj","bumhighpproj"),TAC=seq(0,6000,500)))
class(prjs)
names(prjs)
```

5.1.3 VPA Suite

```
```

5.1.4 VPA2Box

```
```

5.1.5 Multifan-CL

6 Manipulating Objects

R has many functions and methods for manipulating data objects such as arrays, data.frames and lists. Although FLR is based on object oriented (OO) programming where data (i.e. slots) and actions (i.e. methods) are grouped together in S4 classes, these methods can still be used with FLR. For example an FLQuant is derived from an array and so can be manipulated using functions written for arrays, while an FLR class has similarities to a list (in that it can contain a variety of data types) and functions that work for lists have been overloaded so that they work for FLR classes. In R it is recommended that rather than using for loops that you use apply and sweep. To the new user this can appear to be confusing but using them speeds up code and helps conceptually by moving towards a "whole object" view. The apply family of functions allow functions to be applied on subsets of different types of R classes (i.e. lapply, tapply, sapply, rapply mapply etc). Sweep is useful when trying to perform an operation on two arrays that might have different dimensions.

We show how these and related functions can be used within FLR. Firstly showing how R functions used for arrays can be used with the FLQuant class, we then describe additional features and functions that have been added to FLR. Following this we describe functions that have been added for the FLR classes themselves.Let's start by loading FLCore and some data sets distributed with it.

6.1 apply

The apply function in base R applies a function to the margins of an array and returns a vector or array or list of the values obtained, i.e. A simple example is the computation of means and sums over one or more dimensions of the FLQuant matrix. The function is particularly useful for operations that require combining the dims of an FLQuant, e.g. calculating trends over time or finding the average values at age. For example one can compute the catch in weight with or mean fishing mortality with or recruitment by sex or look at trends in mature and immature catch The apply family To simplify some common operations with apply and make the code more readable, a set of wrappers are implemented in FLR to compute sums, totals, means and vars for each FLQuant dimension. These are named by the combination of the dimension and the computation, e.g., to compute the sum over an FLQuant dimension use xxxSums (i.e. quantSums, yearSums,). xxxSums condenses the xxx dimension by summing up across it, i.e. for FLQuant objects with just age and year. Where there are other dimensions using apply becomes more difficult xxxTotals is similar but rather than condensing a dimension it replicates the sum across it, making it easy to calculate proportions for example. xxxMeans and xxxVars are useful summary methods, although are confusingly named since to calculate the mean by age you have to use yearMeans

6.2 sweep

sweep return an array/FLQuant derived from an input array/FLQuant by sweeping out a summary statistic The function is useful when performing an operation on two arrays which dont have the same dimensions, for example when calculating catch proportions (see yearTotals example above), selection pattern by first scaling fishing mortality-at-age by the average value or looking at sex ratios where numbers by sex are divided by total numbers

6.3 Re-shaping

Often the dimensions of an FLQuant have to be changed, e.g. to add or remove extra years or ages, this can be done by trim and window. The first allows the selection of different dimensions at once in a very flexible way, while the second deals only with year but allows it to be extended. When conducting Monte Carlo Simulations iterations can be added using propagate, or using the random variable methods (e.g. rnorm). Changing other dimensions can be a bit more tricky as this depends upon what the FLQuant represents, for example if you increase the number of seasons, sexes or areas how does weight-at-age change or how should mortality be partitioned? expand provides a method for increasing the size of an FLQuant without actually filling up the new object with values. In cases where there is an accepted algorithm, for example to create a plusgroup then a specific method such as setPlusGroup can be implemented. Some examples below, note that the selection does not need to be continuous Bare in mind that when changing the information about a stock by trimming some information, the other information must also be computed, e.g., if 'age' is trimmed in ple4, catch, landings and discards need to be recalculated. However, the effect of getting rid of younger ages is not necessarily the same as getting rid of older ages since often the oldest age is a plus group and represents the sum of all ages greater or equal to that age. Another example to illustrate trim's flexibility As mentioned above, to select and extend continuous years on the object use A simple example While more technical manipulations like setting a new plus group can be done with A simple example Note that when applied to a FLQuant the method simply computes the average of the age groups to be merged. Expanding in general can be done with Note that the usage of expand for the non-numeric dimensions like unit and area can be more than a bit tricky. For example if you add extra seasons or units what should be the new stock weights-at-age? And to expand and fill the empty information at once use Note the effect of the fill.iter argument.

6.4 Random Variables

6.5 FLQuants

6.6 lapply

6.7 FLR Complex objects

6.8 Qapply

6.9 FLPar

6.10 Sweep

6.11 Conversions

7 Functions

Very often you will want to write a function in R to perform a given task or routine that may be required several times. If you have previously written code for other languages you may be familiar with the conventional approach of writing 'for loops' whereby the code loops successively through each element of the object and performs the same action on each. The example below is a trivial case, simply adding 2 to a numeric vector using the conventional programming approach. In R, this is generally the slow and inefficient method.

```
for (i in 1:length(x))
x[i] <- x[i] + 2
```

A far more effective approach is to use the vectorised arithmetic of R. But as mentioned above, beware of automatic recursion.

```
x <- c(2, 3, 4, 5)
x + 2
```

In the example below we create a very simple function that calculates the cube of any value, or vector of values passed to it. A function comprises 2 main components: the function arguments and a function body. The function body contains error trapping, the main processes of the function and the return statement.

```
cube <- function(x, na.rm = TRUE) {
if (na.rm == TRUE)
x <- x[!is.na(x)]
return(x^3)
}
a <- 1:5
cube(a)
```

Our function takes two arguments, x and na.rm and that by default na.rm is set to TRUE. The function performs some simple error trapping conditional on the value of na.rm and returns an object of the same class as x, cubed. If we want to see the formal arguments to any function we can use the args() method, as below.

```
args(cube)
```

7.1 Debugging

As soon as you write a function you will find that it doesn t work as expected due to a bug.

```
x <- c(2, 3, 4, 5)
err <- function(a,b){
v1 <- mean(a)
v2 <- median(a)
loga <- log(a[,1])
res
<- mean(log(a))
return(res)}
err(rnorm(5,0,1))
```

You can debug functions written in R using the environment browser. This will interrupt the execution of an expression and allow the inspection of the environment where browser was called from. A call to browser can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the R interpreter, enabling you to interrogate the objects that have been declared locally within the function. For example if we have the following function which takes a numeric vector and performs a number of simple tasks eventually returning the object res ... You will see that the function returns an error: Error in a[, 1] : incorrect number of dimensions To debug

the function we simply insert a call to browser() within the function as below. This will halt execution of the function at the point of the call to browser(). Entering 'n' at the command prompt will execute the remainder of the function line by line. Entering a 'c' will execute the remaining code and exit the function.

```
err2 <- function(a,b){
browser()
v1 <- mean(a)
v2 <- median(a)
loga <- log(a[,1])
res
<- mean(log(a))
return(res)}
err2(rnorm(5,0,1))
```

By stepping through the code and interrogating the objects that are created you should be able to detect the line at which the function fails. One disadvantage of browser() is that on encountering the point of failure the environment immediately returns to the top level. A more sophisticated debugger is available in the 'debug' package

```
install.packages('debug')
library(debug)
mtrace(err)
err(rnorm(5,0,1))
mtrace.off(err)
```

With debug there is no need to insert break points or commands within the function. Instead you declare the function for debugging using 'mtrace' as shown below