Working with data

Laurence Kell
<laurie.kell@iccat.int>

Rob Scott
<robert.scott@jrc.ec.europa.eu>

17. January 2011

Table of Contents

1 Introduction

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 into it. Then you will soon want to convert data between the various internal R object formats and between R and a variety of external applications. This will require importing data that have been stored in text files, spreadsheets, databases or binary files, changing the type of R objects created and on occasions exporting R objects.

Therefore in this document we describe

i. the main R objects

ii. how they map to FLR objects,

iii. external data types and how they can be imported into R,

iv. how FLR objects can be exported.

Once you have data objects in R it also necessary to manipulate them in some way in order to summarise their contents, produce subsets or merge them with other objects.

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 (http://cran.r-project.org/doc/manuals/R-lang.html) and the R reference card http://cran.r-project.org/doc/contrib/Short-refcard.pdf for a summary of how to use them.

Vectors are continuous cells containing data of a single type. These types can be of character (or string), integers, doubles, logical, raw (or bytes) and complex. Matrices and arrays are similar to vectors but with attributes for the dimensions (dim) and, optionally, the dimension names (dimnames). A matrix can have 2 dimensions (i.e. row and column) and arrays can have 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 labeling the variables and a row.names attribute for labeling the cases. A data frame can also contain a list that is the same length as the other components. It is worth mentioning the model.frame method (or function) which within certain pacakges returns a data.frame with each variable needed to use the formula in a model, such as a linear regression, stored as columns.

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 in an FLR class designed to represent a stock. An 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 R.

2.1 Some basics before we start

R is an object oriented language. It is not important that you know exactly what this means, however, one 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. Some methods in R, such as plot() or summary() will perform different actions on different objects depending on which class they belong to. For this reason it is important to know how to find the appropriate help information within R 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
```

Some 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 shown below. In this 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 'integer'. 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.

```
<<chunk 1, echo=TRUE>>
x <- array(rnorm(9),dim=c(3,3),dimnames=list(a=letters[1:3],b=letters[24:26]))

class(x)              # will tell you what class object x belongs to
typeof(x)
mode(x)
is(x)
@
```

But as we said before, a comprehensive knowledge of object oriented programming is not essential to begin using R and FLR. Just remember that everything in R is an object.

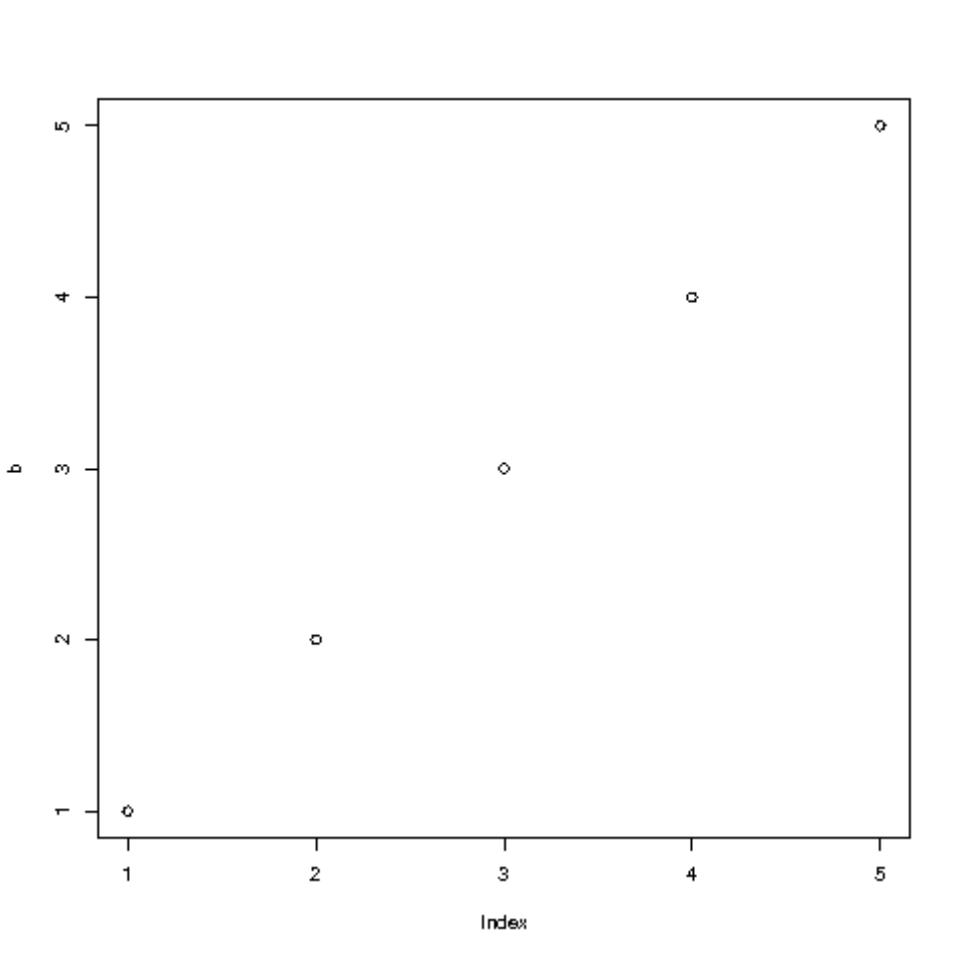
2.2 Vectors

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.

```
> b = 1:5
> length(b)
[1] 5
> b
[1] 1 2 3 4 5
> b^2
[1]  1  4  9 16 25
```

Since R is an object orientated language it also knows how to do things like plot with an object. In this case a trivial exercise but with more complicated objects a useful feature.

```
> plot(b)
```



A vector must contain elements of the same type, e.g. a vector of doubles (as before) or strings. 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 to it.

```
> log(b)
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

R is highly case sensitive. The correct case must be used when calling functions and methods. Variable names are also case sensitive. When naming variables one is allowed to use 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.

```
> a <- 1:6
> b <- 1:3
> a + b
[1] 2 4 6 5 7 9
> c <- 1:4
> a + c
[1] 2 4 6 8 6 8
```

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
[1]   3   4  NA Inf
> tt <- c(1, NA, 3, 4, NA, 6, NaN)
> is.na(tt)
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE
> is.nan(tt)
[1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

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)
[1] NA
> sum(a, na.rm = T)
[1] Inf
```

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

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

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

```
> c(T, F, T, T, F, F) == TRUE
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE
> !c(T, F, T, T, F, F)
[1] FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

rnorm creates random numbers, note that max returns the maximum value in a vector, but pmax does a pairwise comparison, comparing the vector with 0.0.

```
> d <- rnorm(10)
> d
[1] -0.71666425  0.83084007  1.05337048  0.41142583 -0.22335145 -0.14195007
[7] -0.39161067 -0.06317457  0.20563619  1.19762321
> max(d)
[1] 1.197623
> pmax(d, 0)
[1] 0.0000000 0.8308401 1.0533705 0.4114258 0.0000000 0.0000000 0.0000000
[8] 0.0000000 0.2056362 1.1976232
```

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]
[1] 1 2 5
> b[b > 2]
[1] 3 4 5
> b[1:3]
[1] 1 2 3
```

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

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

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)
```

2.3 Matrices and Arrays

Matrices and arrays are vectors with attributes that define the dimensions (dims) and optionally the dimensin names (dimnames), therefore to define a two-dimensional array (matrix) we use the following

```
> a <- matrix(rnorm(16), nrow = 4)
> a
           [,1]       [,2]       [,3]       [,4]
[1,] -2.1793525 -1.252523  1.0530357  0.6622141
[2,]  0.2133713  2.218222 -0.8196815 -1.8440235
[3,]  0.3401111  1.465529 -0.5241879  0.4524062
[4,]  0.5660242  2.056896  0.8799477  1.4071838
> dim(a)
[1] 4 4
```

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.

```
> a <- matrix(1:16, nrow = 4)
> a <- matrix(1:16, nrow = 4, byrow = TRUE)
> attributes(a)
$dim
[1] 4 4
```

It is possible to remove dimensions, for example to get a vector rather than a matrix, by changing the attributes of the object.

```
> aa <- a
> attr(aa, "dim") <- NULL
> attributes(aa)
NULL
> aa
[1]  1  5  9 13  2  6 10 14  3  7 11 15  4  8 12 16
```

A matrix can be subset to obtain specific values, i.e. a vector defined by row or column

```
> a[2, 3]
[1] 7
> a[2, ]
[1] 5 6 7 8
> a[2:3, ]
     [,1] [,2] [,3] [,4]
[1,]    5    6    7    8
[2,]    9   10   11   12
```

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

```
> attr(a, "dim") <- c(2, 4, 2)
> a
, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    9    2   10
[2,]    5   13    6   14

, , 2

     [,1] [,2] [,3] [,4]
[1,]    3   11    4   12
[2,]    7   15    8   16
```

2.4 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
$name
[1] "cod"

$age
[1] 3

$male
[1] FALSE
```

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

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

So, using double square brackets returns the original type, whilst single square brackets returns a list

```
> class(fish$name)
[1] "character"
> class(fish[["name"]])
[1] "character"
> class(fish[[1]])
[1] "character"
> class(fish[1])
[1] "list"
```

You can also create arrays of type list e.g.

```
> array(list(), c(2, 3))
     [,1] [,2] [,3]
[1,] NULL NULL NULL
[2,] NULL NULL NULL
```

2.5 Data frames

In a data.frame all vectors need to be of 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
$age
[1]  1  2  3  4  5  6  7  8  9 10

$weight
[1] 0.05 0.10 0.20 0.30 0.40 0.50 0.60 0.60 0.70 0.90

> b <- as.data.frame(a)
> b
   age weight
1    1   0.05
2    2   0.10
3    3   0.20
4    4   0.30
5    5   0.40
6    6   0.50
7    7   0.60
8    8   0.60
9    9   0.70
10  10   0.90
```

It is also possible to make dataframe in one go, because of recycling you don´t need to replicate all elements

```
> f <- data.frame(species = "plaice", 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))
```

You can subset a dataframe i.e. to get a vector

```
> b$age
[1]  1  2  3  4  5  6  7  8  9 10
> b[, 2]
[1] 0.05 0.10 0.20 0.30 0.40 0.50 0.60 0.60 0.70 0.90
```

It is sometimes useful to determine what class an object belongs to and to add comments

```
> class(b)
[1] "data.frame"
> class(b$age)
[1] "integer"
> attr(b, "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

```
> write.table(b, )
"age" "weight"
"1" 1 0.05
"2" 2 0.1
"3" 3 0.2
"4" 4 0.3
"5" 5 0.4
"6" 6 0.5
"7" 7 0.6
"8" 8 0.6
"9" 9 0.7
"10" 10 0.9
> args(write.table)
function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
    eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
    qmethod = c("escape", "double"))
NULL
> args(read.table)
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
    encoding = "unknown")
NULL
> args(fix)
function (x, ...)
NULL
```

3 Creating Summaries of Objects

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.

```
a <- rnorm(100)
mean(a)
sd(a)
plot(a)
hist(a)
boxplot(a)
boxplot(a, col="red")
qqnorm(a)
qqline(a)
```

median and first and quarter of data

```
> a <- rnorm(100)
> fivenum(a)
[1] -1.98911852 -0.60397999  0.08674817  0.72753232  2.60073614
> stem(a)

  The decimal point is at the |

  -2 | 00
  -1 | 8887665
  -1 | 443321
  -0 | 99888877766665
  -0 | 44443322222221111110
   0 | 11111222233333444
   0 | 555556777889
   1 | 000001133334
   1 | 556689
   2 | 124
   2 | 6
```

The faithful data set is loaded with the base distribution of R. The following example shows a simple histogram plot of the waiting times between the eruptions of the geyser.

```
> data(faithful)
> hist(faithful$waiting)
> rug(faithful$eruptions)
```

Using source you can "batch" a script running it completely, NOTE this is fake reference to script file

```
source("C:\\mydir\\...")
```

4 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.

```
> x <- c(2, 3, 4, 5)
> 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
[1] 4 5 6 7
```

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. We can see here that 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.

```
> cube <- function(x, na.rm = TRUE) {
+     if (na.rm == TRUE)
+         x <- x[!is.na(x)]
+     return(x^3)
+ }
> a <- 1:5
> cube(a)
[1]   1   8  27  64 125
```

If we want to see the formal arguments to any function we can use the args() method, as below.

```
> args(cube)
function (x, na.rm = TRUE)
NULL
```

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. 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,..)

```
> as.double("2")
[1] 2
> as.character(2)
[1] "2"
> df <- data.frame(string = c("eeny", "meeny", "miny", "mo"), integer = 1:4)
> as.matrix(df)
     string  integer
[1,] "eeny"  "1"
[2,] "meeny" "2"
[3,] "miny"  "3"
[4,] "mo"    "4"
```

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.

```
> library(FLCore)
> data(ple4)
> head(as.data.frame(ple4))
   slot age year   unit season   area iter      data
1 catch all 1957 unique    all unique    1  78422.95
2 catch all 1958 unique    all unique    1  88240.06
3 catch all 1959 unique    all unique    1 109237.62
4 catch all 1960 unique    all unique    1 117137.86
5 catch all 1961 unique    all unique    1 118330.99
6 catch all 1962 unique    all unique    1 125272.44
> M <- array(0.2, dim = c(10, 20), dimnames = list(age = 1:10,
+     year = 1991:2010))
> as.FLQuant(M)
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
  1   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  2   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  3   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  4   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  5   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  6   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  7   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  8   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  9   0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  10  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
    year
age  2006 2007 2008 2009 2010
  1   0.2  0.2  0.2  0.2  0.2
  2   0.2  0.2  0.2  0.2  0.2
  3   0.2  0.2  0.2  0.2  0.2
  4   0.2  0.2  0.2  0.2  0.2
  5   0.2  0.2  0.2  0.2  0.2
  6   0.2  0.2  0.2  0.2  0.2
  7   0.2  0.2  0.2  0.2  0.2
  8   0.2  0.2  0.2  0.2  0.2
  9   0.2  0.2  0.2  0.2  0.2
  10  0.2  0.2  0.2  0.2  0.2

units:  NA
```

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))
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  1
  1  0.2
  2  0.2
  3  0.2
  4  0.2
  5  0.2
  6  0.2
  7  0.2
  8  0.2
  9  0.2
  10 0.2

units:  NA
> as.FLQuant(data.frame(expand.grid(age = 1:10, year = 1990:2010),
+     data = 0.2))
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004
  1  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  3  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  4  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  5  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  6  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  7  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  8  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  9  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
  10 0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
    year
age  2005 2006 2007 2008 2009 2010
  1  0.2  0.2  0.2  0.2  0.2  0.2
  2  0.2  0.2  0.2  0.2  0.2  0.2
  3  0.2  0.2  0.2  0.2  0.2  0.2
  4  0.2  0.2  0.2  0.2  0.2  0.2
  5  0.2  0.2  0.2  0.2  0.2  0.2
  6  0.2  0.2  0.2  0.2  0.2  0.2
  7  0.2  0.2  0.2  0.2  0.2  0.2
  8  0.2  0.2  0.2  0.2  0.2  0.2
  9  0.2  0.2  0.2  0.2  0.2  0.2
  10 0.2  0.2  0.2  0.2  0.2  0.2

units:  NA
```

```
> qt <- as.FLQuant(data.frame(expand.grid(age = 1:10, year = 1990:2010),
+     data = 0.2))
> stk <- FLStock(as.FLQuant(as.data.frame(expand.grid(age = 1:10,
+     year = 1990:2010, data = 0.2))))
```

6 Creation

To Do

7 Reading Data into R

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.

7.1 Comma separated data

First of all please note that 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.

```
> cat("Some title information", "1,2,3,4,5", "3,2,9,1,8", file = "test.data",
+     sep = "\n")
> test1 <- scan("test.data", skip = 1, sep = ",")
> test2 <- scan("test.data", skip = 0, sep = ",", what = "character")
> test3 <- readLines("test.data", n = 2)
> test4 <- read.table("test.data", header = F, sep = ",", skip = 1)
```

7.2 Importing Data from a Spreadsheet

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 and xlsx all contain routines for accessing data in spreadsheet form, although at the time of writing the package xlsReadWrite was not available for download from cran. In actual fact, we won't use any of these packages. In the example below we use the `read.xls` method available in the gdata package to read data

from the excel file swordfish.xls which can be downloaded from the FLR website. First download the excel file and save it to a folder somewhere. 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 also (install.packages('gdata')).

```
library(gdata)
file <- '/home/scottro/FLR/Data/swordfish.xls'
sheetcount <- sheetCount(file)
sheetnames <- sheetNames(file)
xlsdata <- read.xls(file, sheet=sheetnames[2], skip=4, nrow=5)
```

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.

7.3 Importing Data into FLR objects

Once you have grasped the concepts above, importing data into FLR objects becomes very easy. It simply 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; integers, vectors, arrays, matrices and so on. In the 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.

```
#There seems to be a problem with this code - needs fixing

#nao <- read.table('http://www.cdc.noaa.gov/data/correlation/noa.data', skip=1,
#nrow=62, na.strings='-99.90')
#dmns <- list(quant="all", year=1948:2009, unit="unique", season=1:12,
#area="unique")
#nao.flq <- FLQuant(unlist(nao[,-1]), dimnames=dmns, units="nao")
#pfun <- function(x,y,...){
#  panel.xyplot(x,y,...)
#  panel.loess(x,y,...)
#}
#xyplot(data~year|season, data=nao.flq, type="l", panel=pfun)
```

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.