Manipulating FLR objects with apply and sweep

Laurence Kell
<Laurie.Kell@iccat.int>
&
Ernesto Jardim
<ernesto@ipimar.pt>

7. September 2010

Table of Contents

1 Introduction

R has many functions and methods for manipulating data objects such as arrays, data.frames and lists. Once **FLR** is based on object oriented (OO) programming where data (i.e. objects) and actions (i.e. methods) are grouped together in S4 classes, these methods can 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** there are a set of methods that replace "for loops" like `apply` and `sweep`. To the new user this can appear to be confusing but using them speeds up code, helps conceptually by moving towards a "whole object" view and makes the code more readable. 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.

```
> library(FLCore)
> data(ple4)
> data(ple4sex)
> data(nsher)
```

2 FLQuant

2.1 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.

```
apply(X,INDEX,FUN)
```

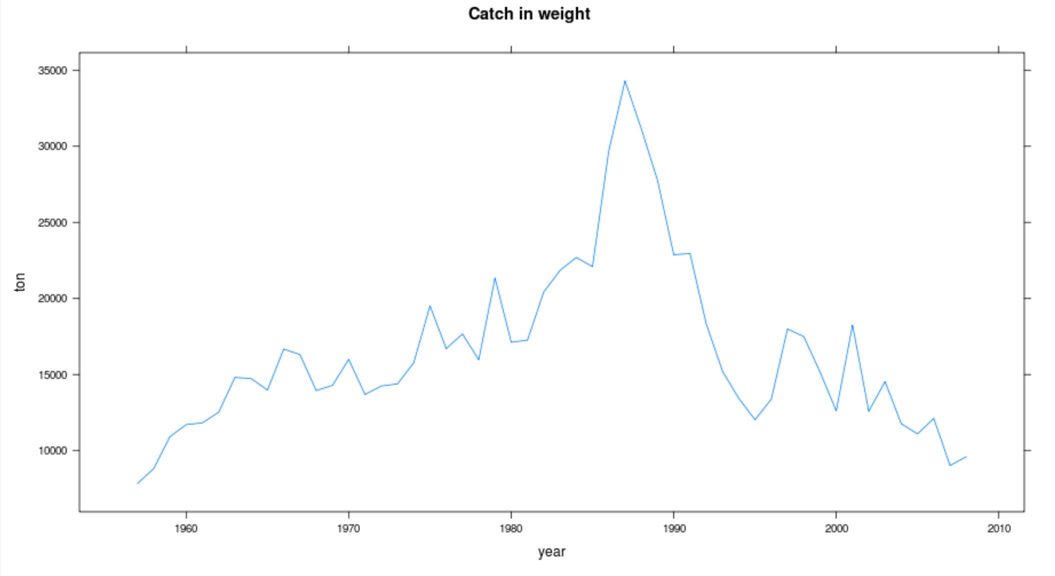
`X`	the array to be used.
`MARGIN`	a vector giving the subscripts which the function will be applied over. `1` indicates rows, `2` indicates columns, `c(1,2)` indicates rows and columns. For an `FLQuant` `1,2,3,4,5,6` indicate `age, year, unit, season, area and iter.`
`FUN`	the function to be applied. In the case of functions like `+`, `%*%`, etc., the function name must be quoted.
`...`	optional arguments to `FUN`.

A simple example is the computation of means and sums over one or more dimensions of the `FLQuant` matrix.

```
> apply(catch.n(ple4), 1, mean)
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  1
  1  172759.0
  2  286083.4
  3  167406.7
  4   89902.2
  5   46690.4
  6   22864.2
  7   12323.7
  8    6252.9
  9    3824.0
  10   8922.3

units:  thousands
> apply(catch.n(ple4), 2, sum)
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

     year
age   1957    1958    1959    1960    1961    1962    1963    1964    1965
  all  274463  348703  486073  500309  505585  554529  601358  578332  761373
     year
age   1966    1967    1968    1969    1970    1971    1972    1973    1974
  all  800262  649376  508329  466118  614984  442137  419151  458236  749838
     year
age   1975    1976    1977    1978    1979    1980    1981    1982    1983
  all  931012  679285  769163  704027  940585  700393  687764 1023162 1132551
     year
age   1984    1985    1986    1987    1988    1989    1990    1991    1992
  all 1166447 1142931 2143204 2354937 1922494 1468754 1075597 1097352  850222
     year
age   1993    1994    1995    1996    1997    1998    1999    2000    2001
  all  655108  515380  497708  697998 1061143 1187466  868346  648722  947864
     year
age   2002    2003    2004    2005    2006    2007    2008
  all  789193  873227  733308  662652  741494  516023  580831

units:  thousands
```

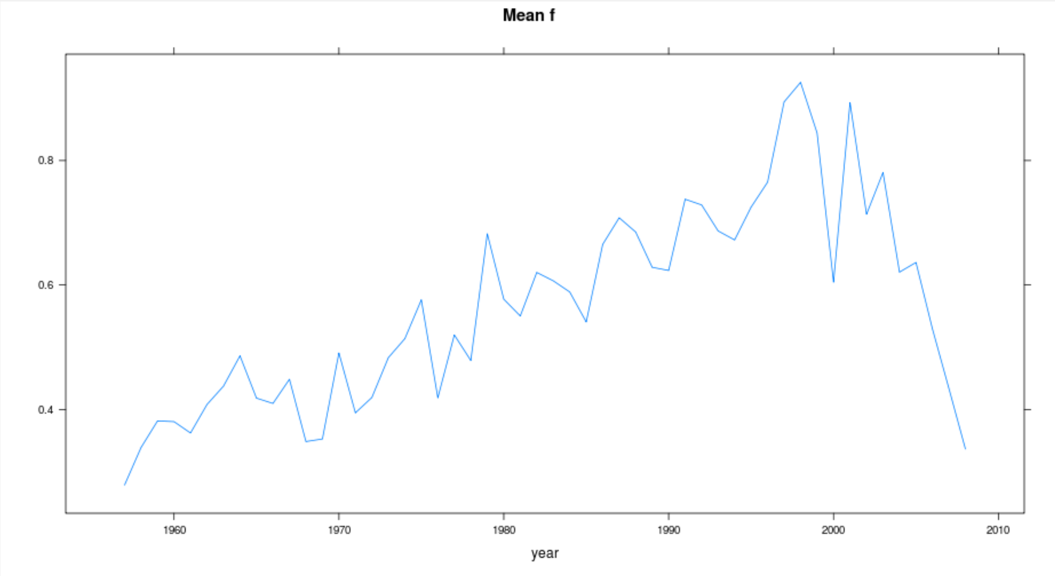
The function is particularly useful for operations that require combining the dims of an `FLQuant`, e.g. calculating trends over time or finding the average values at age. For example one can compute the catch in weight with

```
> flq <- apply(catch.wt(ple4) * catch.n(ple4), 2, mean)
> print(xyplot(data ~ year, data = flq, type = "l", main = "Catch in weight",
+      ylab = "ton"))
```



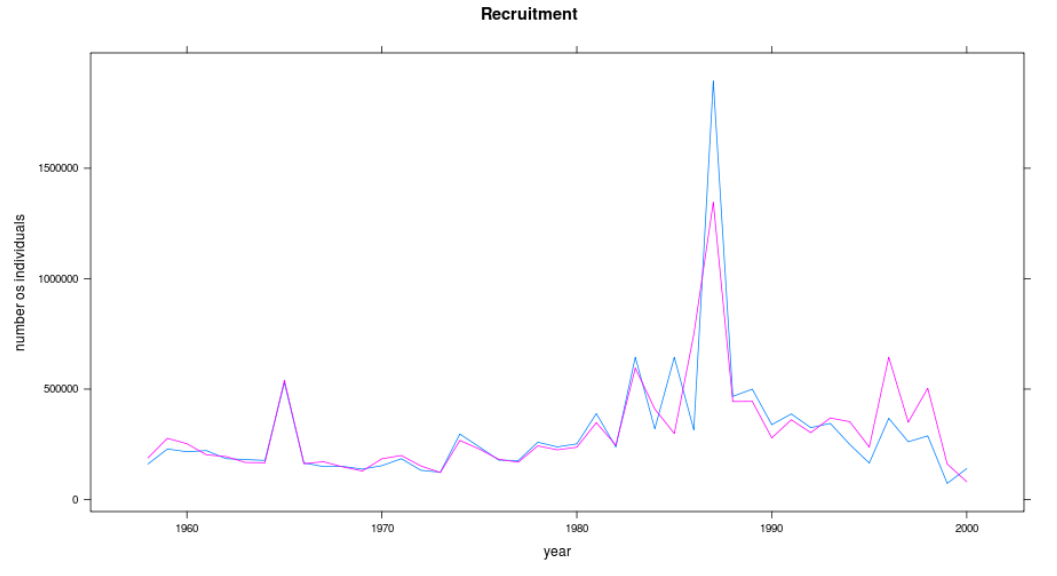
or mean fishing mortality with

```
> flq <- apply(harvest(ple4)[as.character(3:6)], 2, mean)
> print(xyplot(data ~ year, data = flq, type = "l", main = "Mean f",
+      ylab = ""))
```



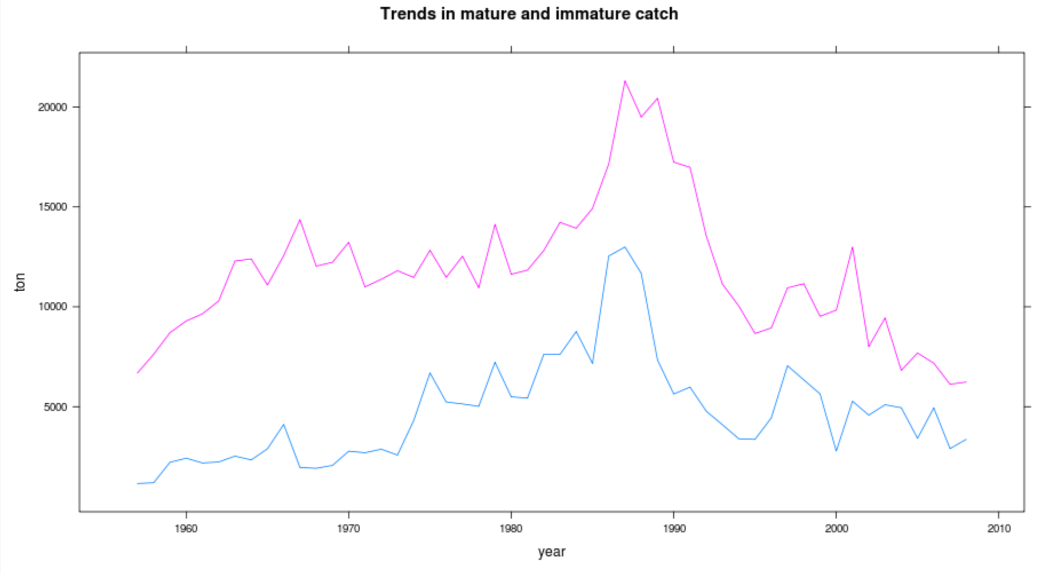
or recruitment by sex

```
> flq <- apply(rec(ple4sex), c(2, 3), sum)
> print(xyplot(data ~ year, group = unit, data = flq, type = "l",
+      main = "Recruitment", ylab = "number os individuals"))
```



or look at trends in mature and immature catch

```
> matCatch <- apply(catch.wt(ple4) * catch.n(ple4) * mat(ple4),
+     2, mean)
> immCatch <- apply(catch.wt(ple4) * catch.n(ple4) * (1 - mat(ple4)),
+     2, mean)
> flqs <- FLQuants(imm = immCatch, mat = matCatch)
> print(xyplot(data ~ year, groups = qname, data = flqs, type = "l",
+     main = "Trends in mature and immature catch", ylab = "ton"))
```



2.2 *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.

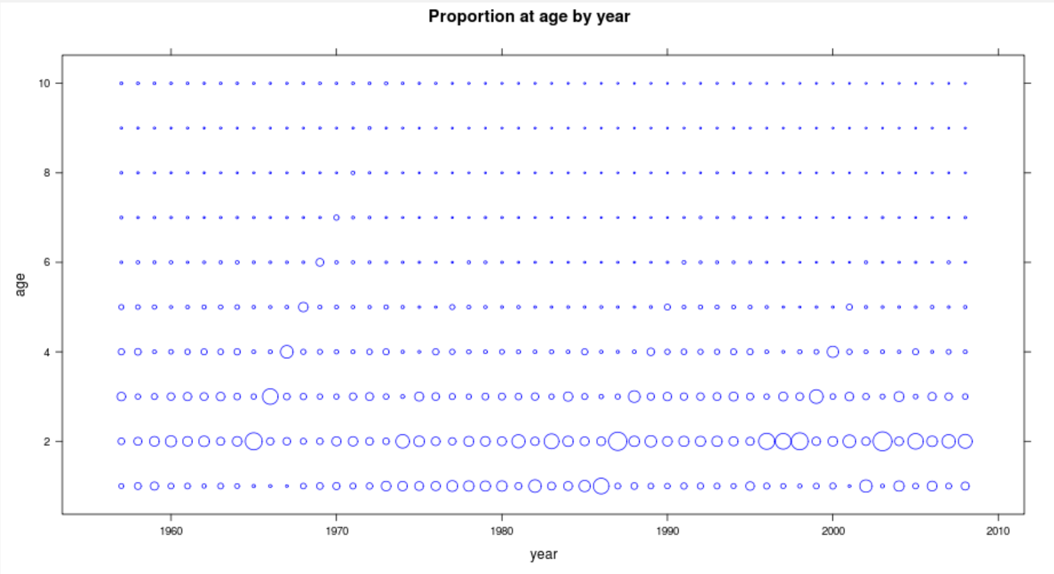
```
> flq1 <- apply(catch.n(ple4), 1, sum)
> flq2 <- yearSums(catch.n(ple4))
> all.equal(flq1, flq2)
[1] TRUE
> flq1 <- apply(catch.n(ple4), 2, sum)
> flq2 <- quantSums(catch.n(ple4))
> all.equal(flq1, flq2)
[1] TRUE
```

Where there are other dimensions using apply becomes more difficult

```
> flq1 <- apply(catch.n(ple4sex), c(1:2, 4:6), sum)
> flq2 <- unitSums(catch.n(ple4sex))
> all.equal(flq1, flq2)
[1] TRUE
```

xxxTotals is similar but rather than “condensing” a dimension it replicates the sum across it, making it easy to calculate proportions for example. These methods are only implemented for quant and year.

```
> yearTotals(catch.n(ple4))[, as.character(2004:2008)]
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  2004   2005   2006   2007   2008
  1  733308 662652 741494 516023 580831
  2  733308 662652 741494 516023 580831
  3  733308 662652 741494 516023 580831
  4  733308 662652 741494 516023 580831
  5  733308 662652 741494 516023 580831
  6  733308 662652 741494 516023 580831
  7  733308 662652 741494 516023 580831
  8  733308 662652 741494 516023 580831
  9  733308 662652 741494 516023 580831
  10 733308 662652 741494 516023 580831

units:  thousands
> flq <- catch.n(ple4)/yearTotals(catch.n(ple4))
> print(bubbles(age ~ year, data = flq, main = "Proportion at age by year"))
```



`xxxMeans` and `xxxVars` are useful summary methods, although the names may be confusing since to calculate the mean by age you have to use yearMeans.

```
> yearVars(catch.wt(ple4sex))
An object of class "FLQuant"
, , unit = male, season = all, area = unique

    year
age  1
  1  0.01401351
  2  0.00098992
  3  0.00108040
  4  0.00172071
  5  0.00262924
  6  0.00304352
  7  0.00325781
  8  0.00526777
  9  0.00407896
  10 0.00648250
  11 0.00673112
  12 0.00754950
  13 0.02284452
  14 0.03602802
  15 0.02287087

, , unit = female, season = all, area = unique

    year
age  1
  1  0.01378140
  2  0.00201949
  3  0.00239446
  4  0.00321595
  5  0.00434408
  6  0.00470233
  7  0.00458162
  8  0.00310044
  9  0.00848206
  10 0.00648272
  11 0.00924828
  12 0.00920520
  13 0.01532536
  14 0.02941595
  15 0.06100897

units:  NA
```

Combining these methods make it simple to compute other statistics like the coefficient of variation

```
> sqrt(yearVars(catch.wt(ple4sex)))/yearMeans(catch.wt(ple4sex))
An object of class "FLQuant"
, , unit = male, season = all, area = unique

    year
age  1
  1  1.524037
  2  0.129218
  3  0.120186
  4  0.131541
  5  0.140081
  6  0.132601
  7  0.125734
  8  0.149398
  9  0.125572
  10 0.148333
  11 0.147301
  12 0.147094
  13 0.245902
  14 0.298204
  15 0.224549

, , unit = female, season = all, area = unique

    year
age  1
  1  1.515901
  2  0.159042
  3  0.151813
  4  0.147431
  5  0.136782
  6  0.116272
  7  0.100123
  8  0.073513
  9  0.110405
  10 0.088979
  11 0.096753
  12 0.091153
  13 0.110257
  14 0.143566
  15 0.191992

units:  NA NA
```

2.3 Sweep

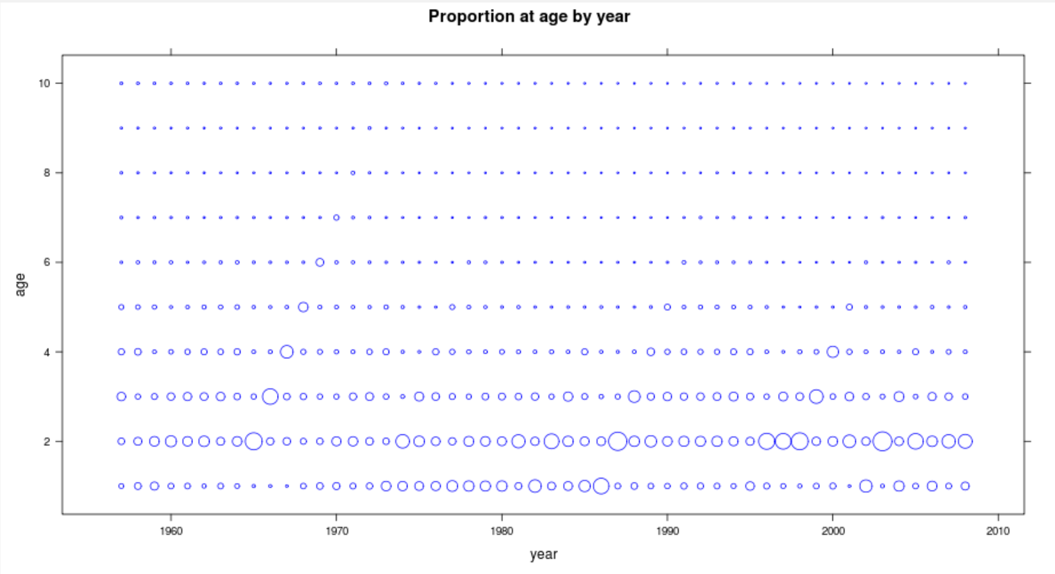
`sweep` return an `array/FLQuant` derived from an input `array/FLQuant` by sweeping out a summary statistic

```
sweep(x, MARGIN, STATS, FUN="-", check.margin=TRUE, ...)
```

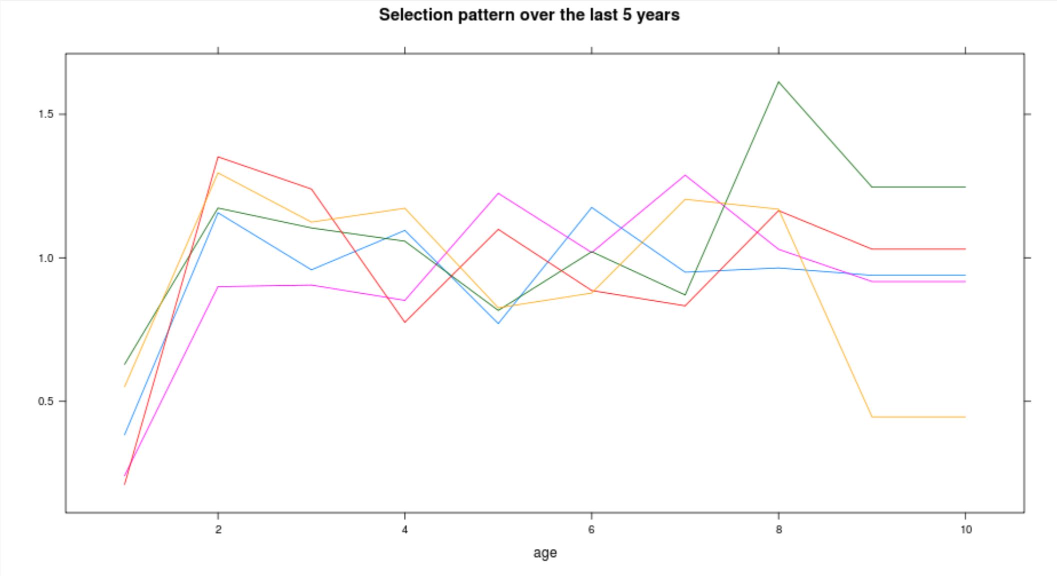
`x`	an array of FLQuant
`MARGIN`	a vector of indices giving the extents of `x` which correspond to `STATS`.
`STATS`	the summary statistic which is to be swept out.
`FUN`	the function to be used to carry out the sweep. In the case of binary operators such as `"/"` etc., the function name must backquoted or quoted. (`FUN` is found by a call to `match.fun`.)
`check.margin`	logical. If `TRUE` (the default), warn if the length or dimensions of `STATS` do not match the specified dimensions of `x`. Set to `FALSE` for a small speed gain when you *know* that dimensions match.
`...`	optional arguments to `FUN`.

The function is useful when performing an operation on two arrays which don´t 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

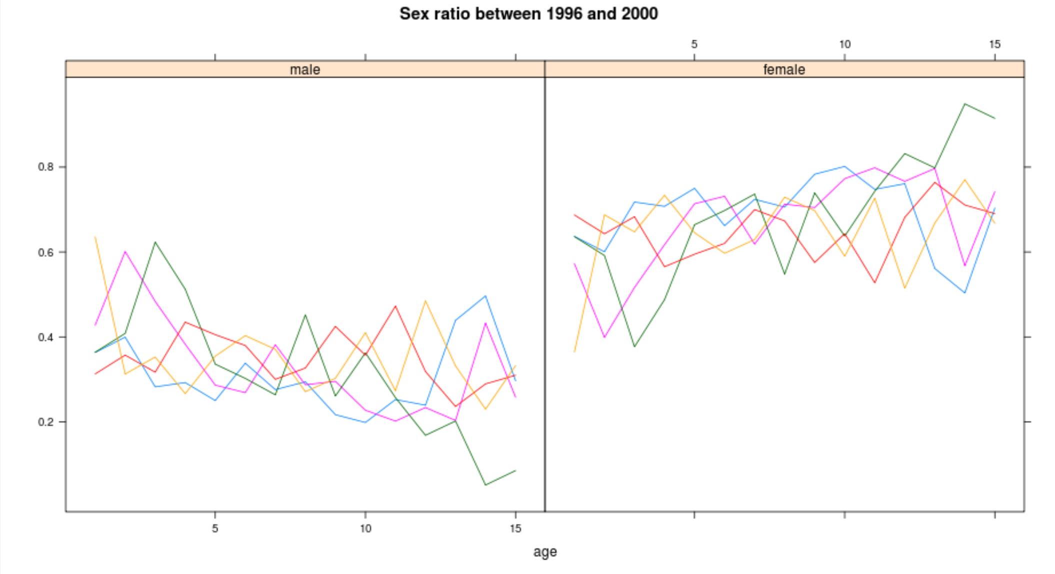
```
> flq <- sweep(catch.n(ple4), 2, apply(catch.n(ple4), 2, sum),
+      "/")
> print(bubbles(age ~ year, data = flq, main = "Proportion at age by year"))
```



```
> fbr <- apply(harvest(ple4)[ac(3:6)], 2, mean)
> selPattern <- sweep(harvest(ple4), 2, fbr, "/")
> print(xyplot(data ~ age, groups = year, data = trim(selPattern,
+     year = c(2004:2008)), type = "l", main = "Selection pattern over the last 5
years",
+     ylab = ""))
```



```
> num <- sweep(stock.n(ple4sex), c(1, 2), apply(stock.n(ple4sex),
+     c(1, 2), sum), "/")
> print(xyplot(data ~ age | unit, groups = year, data = trim(num,
+     year = c(1996:2000)), type = "l", main = "Sex ratio between 1996 and 2000",
+     ylab = ""))
```



2.4 *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.

```
trim(x, ...)
```

`x`	an FLQuant.
`...`	The dimensions of the FLQuant to be trimmed, which can any of `year`, `unit`, `season`, `area` or `iter`. In the case of the quant dimension it will depend on its name, usualy `age` or `all`, but it can be something else that the user must know. In case of doubts run `quant` on your object. In any case the trim information is given in a numeric or character vector that must be in agreement with the dimensions of the quant. No point on selecting year 1 if it does not exist ...

Some examples below, note that the selection does not need to be continuous

```
> trim(catch.n(ple4), year = c(2000:2008), age = c(1:4))
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

   year
age 2000   2001   2002   2003   2004   2005   2006   2007   2008  
  1 106056  34804 311003  68309 233397  96478 220828  78598 135786
  2 187647 388808 193978 560788 191141 342682 244064 224345 263599
  3  91904 240444 136274 101562 221343  72076 177680 106621  80258
  4 230201 123671  61739  68836  43238 103120  35910  57788  44207

units:  thousands
> trim(catch.n(ple4sex), year = c(1995:2000), unit = "female",
+     age = 10:14)
An object of class "FLQuant"
, , unit = female, season = all, area = unique

    year
age  1995    1996    1997    1998    1999    2000   
  10 2318.98 1848.49 1716.03 1317.92  657.66 1054.35
  11  727.62  977.43 1310.99  730.05  411.88  633.39
  12  599.75  448.37  634.74 1163.33  414.76  349.49
  13  560.61  210.95  298.14  592.06  611.70  224.43
  14  296.50  199.10  218.98  315.91  218.08  367.98

units:  NA
```

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

```
> trim(catch.n(ple4), year = c(1990:1995, 2000, 2008), age = c(4:1))
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

   year
age 1990   1991   1992   1993   1994   1995   2000   2008
  4 139099 156894 106842  84252  81274  79230 230201  44207
  3 266440 225170 179701 146463 137320  97151  91904  80258
  2 311831 344841 263573 208032 140851 119251 187647 263599
  1 146864 184587 142165  99832  63516 126614 106056 135786

units:  thousands
```

As mentioned above, to select and extend continuous years on the object use

```
window(x, start, end)
```

`x`	an FLQuant.
`start`	First year in new FLQuant
`end`	Last year in new FLQuant

A simple example

```
> window(catch.n(ple4), 2005, 2010)
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  2005   2006   2007   2008   2009   2010  
  1   96478 220828  78598 135786     NA     NA
  2  342682 244064 224345 263599     NA     NA
  3   72076 177680 106621  80258     NA     NA
  4  103120  35910  57788  44207     NA     NA
  5   21746  45525  16408  31758     NA     NA
  6   15610   6965  24293   6217     NA     NA
  7    5489   5721   2663  15634     NA     NA
  8    2615   2247   3505   1803     NA     NA
  9    2223   1235    543    673     NA     NA
  10    613   1319   1259    896     NA     NA

units:  thousands
```

While more technical manipulations like setting a new plus group can be done with

```
setPlusGroup(x, plusgroup, ...)
```

`x`	an FLQuant.
`plusgroups`	the age for the new plus group
`...`	not implemented for the moment.

A simple example

```
> setPlusGroup(catch.n(ple4), 5)[, ac(2005:2008)]
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

   year
age 2005     2006     2007     2008
  1  96478.0 220828.0  78598.0 135786.0
  2 342682.0 244064.0 224345.0 263599.0
  3  72076.0 177680.0 106621.0  80258.0
  4 103120.0  35910.0  57788.0  44207.0
  5   8049.3  10502.0   8111.8   9496.8

units:  thousands
> quantMeans(catch.n(ple4)[ac(5:10), ac(2005:2008)])
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

     year
age   2005    2006    2007    2008
  all   8049.3 10502.0  8111.8  9496.8

units:  thousands
```

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

```
expand(x, ...)
```

`x`	an FLQuant.
`...`	The dimensions of the FLQuant to be expanded, which can any of `year`, `unit`, `season`, `area` or `iter`. In the case of the quant dimension it will depend on its name, usualy `age` or `all`, but it can be something else that the user must know. In case of doubts run `quant` on your object. In any case the expand information is given in a numeric or character vector that must be in agreement with the dimensions of the quant. In particular it must start on the initial value of the dimension. This is redundant and likely to be changed in the future.

Note that the usage of `expand` for the non-numeric dimensions like `unit` and `area` can be tricky. For example if you add extra seasons or units what should be the new stock weights-at-age?

```
> expand(catch.n(ple4), age = 1:12)[, ac(2005:2008)]
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  2005   2006   2007   2008  
  1   96478 220828  78598 135786
  2  342682 244064 224345 263599
  3   72076 177680 106621  80258
  4  103120  35910  57788  44207
  5   21746  45525  16408  31758
  6   15610   6965  24293   6217
  7    5489   5721   2663  15634
  8    2615   2247   3505   1803
  9    2223   1235    543    673
  10    613   1319   1259    896
  11     NA     NA     NA     NA
  12     NA     NA     NA     NA

units:  thousands 
> expand(catch.n(ple4), unit = c("unique", "female"))[, ac(2005:2008)]
An object of class "FLQuant"
, , unit = unique, season = all, area = unique

    year
age  2005   2006   2007   2008  
  1   96478 220828  78598 135786
  2  342682 244064 224345 263599
  3   72076 177680 106621  80258
  4  103120  35910  57788  44207
  5   21746  45525  16408  31758
  6   15610   6965  24293   6217
  7    5489   5721   2663  15634
  8    2615   2247   3505   1803
  9    2223   1235    543    673
  10    613   1319   1259    896

, , unit = female, season = all, area = unique

    year
age  2005   2006   2007   2008  
  1      NA     NA     NA     NA
  2      NA     NA     NA     NA
  3      NA     NA     NA     NA
  4      NA     NA     NA     NA
  5      NA     NA     NA     NA
  6      NA     NA     NA     NA
  7      NA     NA     NA     NA
  8      NA     NA     NA     NA
  9      NA     NA     NA     NA
  10     NA     NA     NA     NA

units:  thousands
```

And to expand and fill the empty information at once use

```
propagate(object, ...)
```

`object`	an FLQuant.
`...`	These can be `iter` , the number of iters to be added, and `fill.iter` a logical defining it the new iters should have the same information as the first (the default) or not.

Note the effect of the `fill.iter` argument.

```
> flq <- propagate(catch.n(ple4), 10)
> all.equal(iter(flq, 1), iter(flq, 10), check.attributes = FALSE)
[1] TRUE
> flq <- propagate(catch.n(ple4), 10, fill.iter = FALSE)
> all.equal(iter(flq, 1), iter(flq, 10), check.attributes = FALSE)
[1] "'is.NA' value mismatch: 520 in current 0 in target"
```

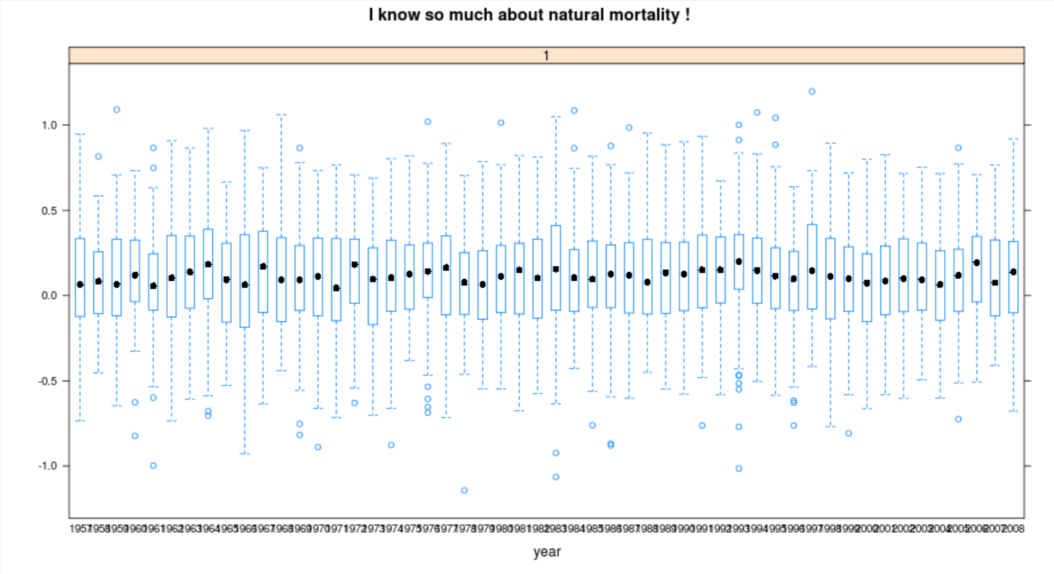
2.5 Random Variables

In **FLR** random variables are modelled in the 6th dimension of the `FLQuant` i.e. `iter`. Some distributions are already implemented in **FLR** making coding easier and more readable, which interact directly with `FLQuants`.

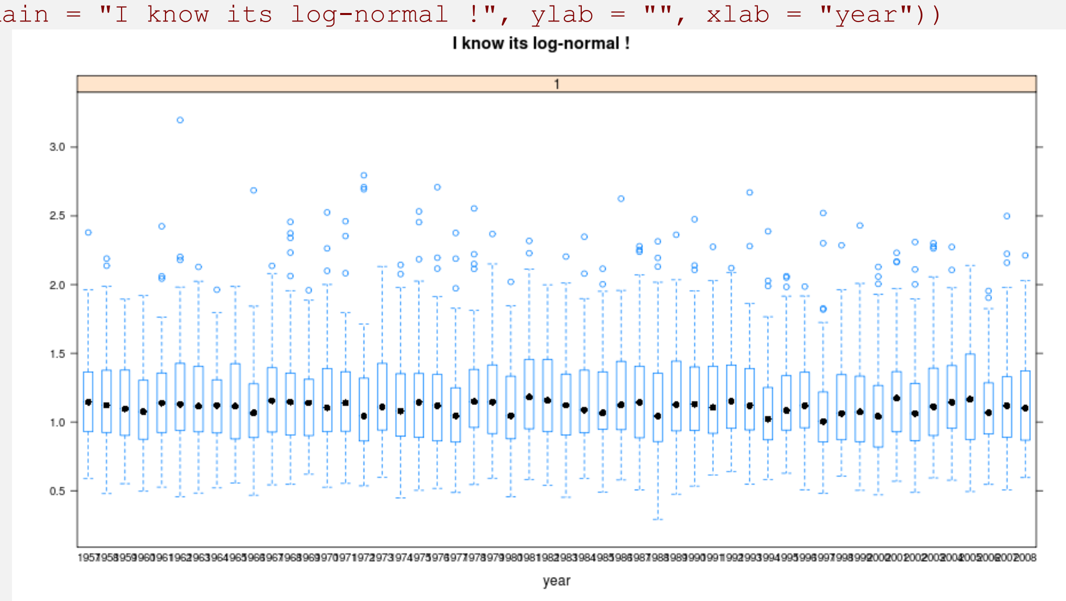
- `runif(n, min, max)`
- `rnorm(n, mean, sd)`
- `rlnorm(n, meanlog,  sdlog)`

For all of these check the relevant help page for a description of each argument. **FLR**'s implementation simply allows the usage of `FLQuant` objects for the arguments.

```
> flq <- rnorm(100, m(ple4), sd = 0.3)
> print(bwplot(data ~ year | factor(age), data = trim(flq, age = 1),
+     main = "I know so much about natural mortality !", ylab = "",
+     xlab = "year"))
```



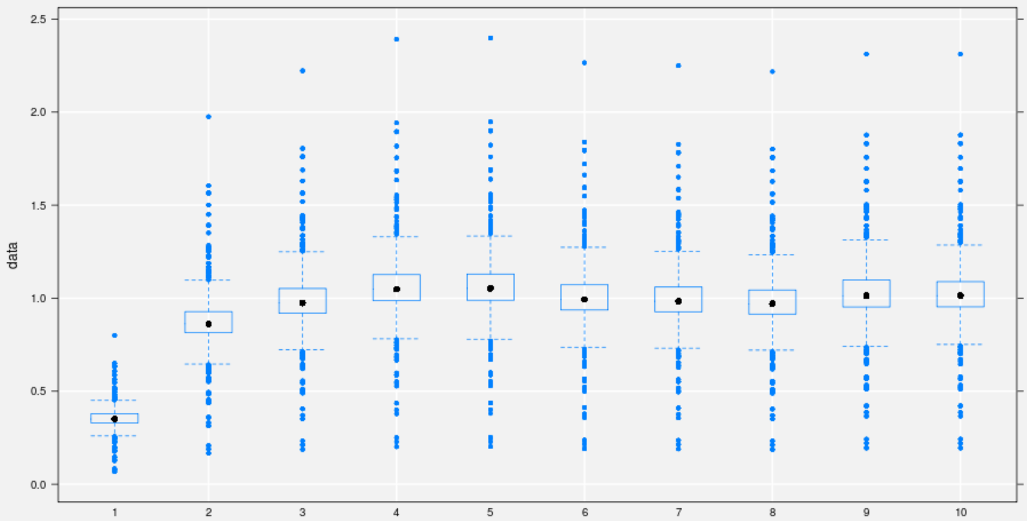

```
> flq <- rlnorm(100, m(ple4), sd = 0.3)
> print(bwplot(data ~ year | factor(age), data = trim(flq, age = 1),
+     main = "I know its log-normal !", ylab = "", xlab = "year"))
```



`runif` can be used to generate other distributions, since if you set min=0 and max=1, then it returns the probability of a value and if you know the quantile function either empirical (e.g. qnorm) or non-parametric from a model fit (e.g. the residuals) then an appropriate pdf can be generated.

Creating an non-parametric random variable can be done using the basic **R** methods, sweep, apply and propagate.

```
> selPat <- sweep(harvest(ple4), 2, fbar(ple4), "/")
> mnSel <- apply(selPat, 1, mean)
> selRsd <- log(sweep(selPat, 1, mnSel, "/"))
> rvSel <- propagate(selRsd, 100)
> rvSel[] <- c(selRsd)[sample(1:length(selRsd), prod(dim(rvSel)),
+     replace = T)]
> rvSel <- sweep(exp(rvSel), 1, mnSel, "*")
> pfun <- function(x, y, ...) {
+     panel.grid(h = -1, v = -1, col = "white", lwd = 2)
+     panel.bwplot(x, y, ...)
+ }
> pset <- list(background = list(col = "gray95"), plot.symbol = list(pch = 19,
+     cex = 0.5))
> print(bwplot(data ~ factor(age), data = rvSel, par.settings = pset,
+     panel = pfun))
```



3 FLlst and FLQuants

The methods applied to `FLQuant` can also be used for the generic `FLlst` objects and more directly to `FLQuants`, by using `lapply`. Which will apply a function over all elements in a list. For more information on `lapply` look for the help pages.

```
lapply(X, FUN, ...)
```

X	an FLlst object
FUN	the function to be applied to each element of 'X'. In the case of functions like '+', '%*%', etc., the function name must be backquoted or quoted.
...	optional arguments to 'FUN'

The examples below illustrate the functionality of lapply. Check that if the function applied does not return a number then the output of lapply will be a list instead of an `FLlst` object. The `as` method is used to coerce objects between classes, here it is used to coerce a `FLStock` object into a `FLQuants` list.

```
> flqs <- as(ple4, "FLQuants")
> flqs2 <- lapply(flqs, yearMeans)
> lst <- list()
> length(lst) <- length(flqs)
> names(lst) <- names(flqs)
> for (i in 1:length(flqs)) lst[[i]] <- yearMeans(flqs[[i]])
> all.equal(flqs2, lst, check.attributes = FALSE)
[1] TRUE
> is(flqs2)
[1] "FLQuants" "FLlst"    "list"     "vector"
> flqs3 <- lapply(flqs, units)
> is(flqs3)
[1] "list"   "vector"
```

Remember that `units` return a character with the units of the data, which is not possible to coerce into a `FLQuant`.

The next example is based on a function that calculates the average of the last n years.

```
> wts <- FLQuants(stock.wt = stock.wt(ple4), catch.wt = catch.wt(ple4),
+     landings.wt = landings.wt(ple4), discards.wt = discards.wt(ple4))
> mnWts <- lapply(wts, function(x, nyrs) apply(x[, dim(x)[2] -
+     0:nyrs], 1, mean), nyrs = 3)
```

More complicated procedures could be written in the same way, e.g. creating random variables from weights-at-age by turning the non parametric random variable example into a function and then using `lapply` with it.

Now suppose one is dealing with a large simulation study and using lists of `FLStock` objects, the so called `FLStocks`, which is an extended class of `FLlst`. From this object one is interesting in extracting all the catch matrices,

```
> stks <- FLStocks(stk1 = ple4, stk2 = ple4, stk3 = ple4)
> is(stks)
[1] "FLStocks" "FLlst"    "list"     "vector"
> flqs <- lapply(stks, catch.n)
> is(flqs)
[1] "FLQuants" "FLlst"    "list"     "vector"
```

because `catch.n` is a method one can apply it with `lapply` and get those slots, with the benefit of generating a `FLQuants` object, instead of a simple list. One way of introducing variability in catch-at-age in all `FLStocks` is the following:

```
> stkS <- lapply(stks, function(x) {
+     catch.n(x) <- rlnorm(1000, catch.n(x), sd = 0.3)
+     x
+ })
```

Note that one can embed `lapply` within other `lapply` calls but be aware that the sequence of methods can have a large impact on the speed and memory used.

```
> system.time(lst1 <- lapply(lapply(stkS, catch.n), trim, age = 1:3))
   user  system elapsed
  0.900   0.040   0.947
> system.time(lst2 <- lapply(lapply(stkS, trim, age = 1:3), catch.n))
   user  system elapsed
  0.350   0.000   0.361
> all.equal(lst1, lst2)
[1] TRUE
```

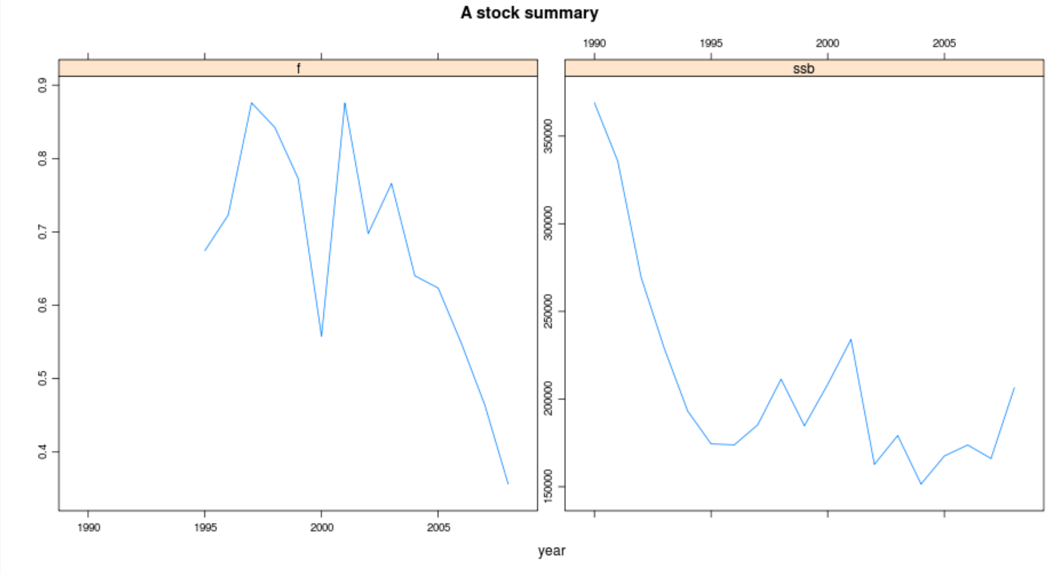
An interesting method, with lots of applications for ploting is `mcf`, wich makes `FLQuants` compatible with respect to their dimensionality. Hence, the `FLQuants` in the returned object all have the same dimensions, padded with NAs if necessary.

`mcf(object, ...)`

`object`	A list of `FLQuant`, not necessarily a `FLQuants`.
`...`	Not imlpemented for the moment.

A simple example is

```
> flqs <- mcf(list(f = fbar(ple4)[, ac(1995:2008)], ssb = ssb(ple4)[,
+     ac(1990:2008)]))
> is(flqs)
[1] "FLQuants" "FLlst"    "list"     "vector"
> print(xyplot(data ~ year | qname, data = flqs, scales = list(y = "free"),
+     type = c("a"), main = "A stock summary", ylab = ""))
```



4 FLR Complex objects

Examples of **FLR** complex objects are `FLBiol`, `FLStock` and `FLIndex`, often you may wish to process all of them using the same function, this can be done using `qapply`.

`qapply(X, FUN, ...)`

X	an FLR object, FLStock, FLBiol, FLFleet, FLIndex, etc
FUN	the function to be applied to each element of 'X'. In the case of functions like '+', '%*%', etc., the function name must be backquoted or quoted.
...	optional arguments to 'FUN'

The examples below illustrate the funcionality of qapply.

```
> dm <- qapply(ple4, dim)
> do.call("rbind", dm)
                 [,1] [,2] [,3] [,4] [,5] [,6]
catch               1   52    1    1    1    1
catch.n            10   52    1    1    1    1
catch.wt           10   52    1    1    1    1
discards            1   52    1    1    1    1
discards.n         10   52    1    1    1    1
discards.wt        10   52    1    1    1    1
landings            1   52    1    1    1    1
landings.n         10   52    1    1    1    1
landings.wt        10   52    1    1    1    1
stock               1   52    1    1    1    1
stock.n            10   52    1    1    1    1
stock.wt           10   52    1    1    1    1
m                  10   52    1    1    1    1
mat                10   52    1    1    1    1
harvest            10   52    1    1    1    1
harvest.spwn       10   52    1    1    1    1
m.spwn             10   52    1    1    1    1
> ple4new <- qapply(ple4, apply, 1, mean, na.rm = TRUE)
> all.equal(ple4new, qapply(ple4, yearMeans))
[1] TRUE
> ple4new <- qapply(ple4, function(x) {
+     names(dimnames(x))[1] <- "what"
+     return(x)
+ })
> unlist(qapply(ple4new, quant))
       catch      catch.n     catch.wt     discards   discards.n  discards.wt 
      "what"       "what"       "what"       "what"       "what"       "what" 
    landings   landings.n  landings.wt        stock      stock.n     stock.wt 
      "what"       "what"       "what"       "what"       "what"       "what" 
           m          mat      harvest harvest.spwn       m.spwn 
      "what"       "what"       "what"       "what"       "what" 
```

5 Final thoughs

As with all computing languages there's more than one way of doing the same thing and the “best” way will depend on the objectives of the analysis. Writing one off functions can often help in analyses, however if one is interested in speed it's a good idea to test your code and try to compute everything on the lowest level of the language. Using functions can also obscure what is being done and clarity is important for transparency.

FLR provides a few methods to deal with its objects which in our view cover all the important manipulations one is required to do. The biggest value is on the mixing and combination of these methods, but off course if something is missing **R** is a great language for quick development. Another major gain is in using other packages which may have implemented better methods than ours, that's the case of plyr (http://had.co.nz/plyr/).