Assessment for All initiative(a4a)
Stock assessment and management advice methods
DRAFT

Ernesto Jardim[1], Colin Millar[2], Finlay Scott[1], Iago Mosqueira[1], and Chato Osio[1]

[1]European Commission, Joint Research Centre, IPSC / Maritime Affairs Unit, 21027 Ispra (VA), Italy
[2]Marine Scotland Freshwater Laboratory, Faskally, Pitlochry, Perthshire PH16 5LB, UK
*Corresponding author ernesto.jardim@jrc.ec.europa.eu

May 14, 2014

Contents

1 Introduction

1.1 Background

(This section is based on Jardim, et.al, 2014)

The volume and availability of data useful for fisheries stock assessment is continually increasing. Time series of 'traditional' sources of information, such as surveys and landings data are not only getting longer, but also cover an increasing number of species.

For example, in Europe the 2009 revision of the Data Collection Regulation (EU, 2008a) has changed the focus of fisheries sampling programmes away from providing data for individual assessments of 'key' stocks (i.e. those that are economically important) to documenting fishing trips, thereby shifting the perspective to a large coastal monitoring programme. The result has been that data on growth and reproduction of fish stocks are being collected for more than 300 stocks in waters where the European fleets operate.

Recognizing that the context above required new methodological developments, the European Commission Joint Research Centre (JRC) started its 'Assessment for All' Initiative (`a4a`), with the aim to develop, test, and distribute methods to assess a large numbers of stocks in an operational time frame, and to build the necessary capacity/expertise on stock assessment and advice provision.

The long-term strategy of `a4a` is to increase the number of stock assessments by reducing the workload required to run each analysis and by bringing more scientists/analysts into fisheries management advice. The first is achieved by developing a working framework with the methods required to run all the analysis a stock assessment needs, as well as developing methods to deal with recognized bottlenecks, *e.g.* model averaging to deal with model selection (Millar, et.al, 2014). Such an approach should make the model exploration and selection processes easier, as well as decreasing the burden of moving between software platforms. The second can be achieved by making the analysis more intuitive, thereby attracting more experts to join stock assessment teams.

To achieve these objectives, the Initiative identified a series of tasks, which were or are being carried out, namely:

- define a moderate data stock;
- develop a stock assessment framework;
- develop a forecasting algorithm based on MSE;
- organize training courses for marine scientists.

1.1.1 The moderate data stock

The moderate data stock definition was an important step in the Initiative's development. It clearly focused the Initiative on stocks with some information, moving away from the data-poor stocks, but without moving into data rich methodologies. It was recognized that there is a lot of research at both extremes of the data availability spectrum, but comparatively little in the middle 'region'. From this came the idea of the 'moderate data stock'.

The 'moderate data stock' constitutes the entry level of our analysis. It has at least the following available data, which can be assembled in different ways, using distinct methods.

- in relation to exploitation:
  - volume of catches, which may be split into landings and discards if possible;
  - length frequencies of the catches, landings or discards;
  - nominal effort (optional, needed in case CPUE indices are to be derived);
- in relation to biology:
  - estimated maturity ogive (e.g. can be as simple as an estimate of L_{50});

  - estimated growth model and parameters;
  - estimated length-weight relationship;
- in relation to abundance:
  - index of abundance.

1.1.2 The stock assessment framework

The stock assessment model framework is a non-linear catch-at-age model implemented in `R`/`FLR`/`ADMB` that can be applied rapidly to a wide range of situations with low parametrization requirements. Later we'll come back to these characteristics and it's application (Section 5).

1.1.3 MSE

The MSE is a sophisticated forecasting algorithm that takes into account structural uncertainty about stock dynamics (growth, recruitment, maturity) and on exploitation by commercial fleets (selectivity), embedding the framework of decision making.

1.1.4 Training

During the last 2 years JRC organized 4 courses of introduction to R and `FLR`: Varese, January 2012; Varese, June, 2012; Barza, March 2013; FAO / GFCM, Rome, November 2013.

In 2013 a short course about `a4a` methods was organized in Lisbon. The first full course on `FLR` and `a4a` methods was organized in CEFAS, March 2014 and another one is planned for August 2014.

These courses are open to all participants and don't have an attendance fee.

1.2 The `a4a` approach to stock assessment and management advice

The approach presented here is split into 4 steps: (i) converting length data to age data using a growth model, (ii) modelling natural mortality, (iii) assessing the stock, and (iv) MSE[1].

These steps may be followed in sequence or independently, depending on the user's preferences. All that is needed is to use the objects provided by the previous step and provide the objects required by the next, so that data flows between steps smoothly. One can make the analogy with building with Lego, where for each layer the builder may use the pieces provided by a particular boxset, or make use of pieces from other boxsets. Figure 1 shows the process, including the class of the objects that carry the data (in black).

[1] Under development, to be released with version 2.0, scheduled for the fourth quarter of 2014

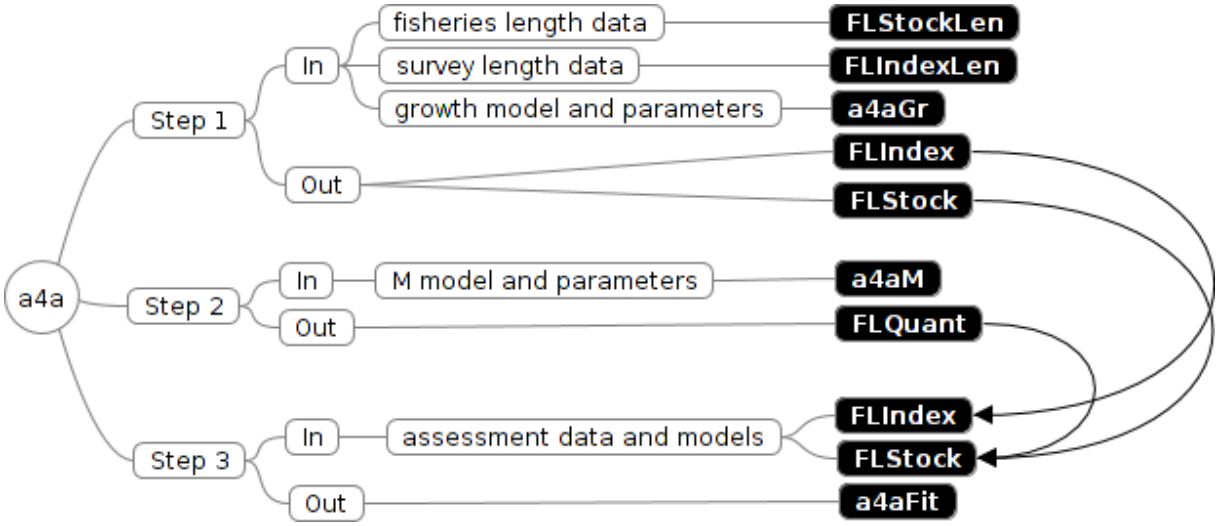

Figure 1: In/out process of the a4a approach. The boxes in black represent the classes of the objects that carry the information for each step and out of each step.

Analysis related to projections and biological reference points are dealt with by the FLR packages FLash and FLBRP. As such the Initiative does not provide specific methods for these analyses.

In Steps 1 and 2 there is no fitting of growth models or natural mortality models. The rationale is to provide tools that allow the uncertainty associated with these processes to be carried on into the stock assessment, e.g. through parameter uncertainty. This approach allows the users to pick up the required information from other sources of information such as papers, PhDs, Fishbase, other stocks, etc. If the stock under analysis does not have specific information on the growth or natural mortality processes, generic information about life history invariants may be used such as the generic priors suggested by Bentley, (2014).

Note that an environment like the one distributed by a4a promotes the exploration of different models for each process, giving the analyst a lot of flexibility. It also opens the possibility to efficiently include distinct models in the analysis. For example, a stock assessment using two growth, or several models for natural mortality could be performed. Our suggestion to streamline the assessment process is to combine the final outcomes using model averaging (Miller, et. al, 2014). Other solutions may be implemented, like scenario analysis, etc. What is important is to keep the data flowing smoothly and the models clear. R (R Core Team, 2014) and FLR (Kell, et.al, 200) provide powerful platforms for this approach.

1.3 Loading libraries, data and defining some useful functions

```
library(FLa4a)
library(XML)
library(reshape2)
library(diagram)
data(ple4)
data(ple4.indices)
data(rfLen)
```

```
# functions for transforming the data

# quant 2 quant
qt2qt <- function(object, id = 5, split = "-") {
    qt <- object[, id]
    levels(qt) <- unlist(lapply(strsplit(levels(qt), split = split), "[[", 2))
    as.numeric(as.character(qt))
```

```
}

# check import and massage
cim <- function(object, n, wt, hrv = "missing") {
    v <- object[sample(1:nrow(object), 1), ]
    c1 <- c(n[as.character(v$V5), as.character(v$V1), 1, as.character(v$V2)] ==
        v$V6)
    c2 <- c(wt[as.character(v$V5), as.character(v$V1), 1, as.character(v$V2)] ==
        v$V7)
    if (missing(hrv)) {
        c1 + c2 == 2
    } else {
        c3 <- c(hrv[as.character(v$V5), as.character(v$V1), 1, as.character(v$V2)] ==
            v$V8)
        c1 + c2 + c3 == 3
    }
}

# and a plot for later
plotS4 <- function(object, linktext = "typeof", main = "S4 class", ...) {
    args <- list(...)
    obj <- getClass(as.character(object))
    df0 <- data.frame(names(obj@slots), unlist(lapply(obj@slots, "[[", 1)))
    nms <- c(t(df0))
    nslts <- length(nms)/2
    M <- matrix(nrow = length(nms), ncol = length(nms), byrow = TRUE, data = 0)
    for (i in 1:nslts) {
        M[i * 2, i * 2 - 1] <- linktext
    }
    args$A = M
    args$pos = rep(2, length(nms)/2)
    args$name = nms
    args$main = main
    do.call("plotmat", args)
}
```

2 Reading files and building FLR objects

For this document we'll use the plaice in ICES area IV dataset, provided by FLR, and a length-based simulated dataset based on red fish, using Gadget, provided by Daniel Howell (Institute of Marine Research, Norway).

In this section we read in the Gadget data files, and transform them into FLR objects.

First we read in the files as data frames and recode some variables.

```
# catch
cth.orig <- read.table("data/catch.len", skip = 5)

# stock
stk.orig <- read.table("data/red.len", skip = 4)

# surveys
idx.orig <- read.table("data/survey.len", skip = 5)
idxJmp.orig <- read.table("data/jump.survey.len", skip = 5)
idxTrd.orig <- read.table("data/tend.survey.len", skip = 5)
```

```
# Recode the length categories into something usable

# catch
cth.orig[, 5] <- qt2qt(cth.orig)

# stock
stk.orig[, 5] <- qt2qt(stk.orig)

# surveys
idx.orig[, 5] <- qt2qt(idx.orig)
idxJmp.orig[, 5] <- qt2qt(idxJmp.orig)
idxTrd.orig[, 5] <- qt2qt(idxTrd.orig)
```

Then we reshape the data frames into six dimensional arrays using `cast()` from package `reshape2`.

```
# catch
cth.n <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V6", data = cth.orig)
cth.wt <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V7", data = cth.orig)
hrv <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V8", data = cth.orig)

# stock
stk.n <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V6", data = stk.orig)
stk.wt <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V7", data = stk.orig)

# surveys
idx.n <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V6", data = idx.orig)
idx.wt <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V7", data = idx.orig)
idx.hrv <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V8", data = idx.orig)
idxJmp.n <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V6", data = idxJmp.orig)
idxJmp.wt <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V7", data = idxJmp.orig)
idxJmp.hrv <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V8", data = idxJmp.orig)
idxTrd.n <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V6", data = idxTrd.orig)
idxTrd.wt <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V7", data = idxTrd.orig)
idxTrd.hrv <- acast(V5 ~ V1 ~ 1 ~ V2 ~ 1 ~ 1, value.var = "V8", data = idxTrd.orig)
```

We take the arrays and make *FLQuant* objects from them.

```
# catch
dnms <- dimnames(cth.n)
names(dnms) <- names(dimnames(FLQuant()))
names(dnms)[1] <- "len"
cth.n <- FLQuant(cth.n, dimnames = dnms)
cth.wt <- FLQuant(cth.wt, dimnames = dnms)
hrv <- FLQuant(hrv, dimnames = dnms)
units(hrv) <- "f"

# stock
dnms <- dimnames(stk.n)
names(dnms) <- names(dimnames(FLQuant()))
names(dnms)[1] <- "len"
stk.n <- FLQuant(stk.n, dimnames = dnms)
stk.wt <- FLQuant(stk.wt, dimnames = dnms)

# surveys
dnms <- dimnames(idx.n)
names(dnms) <- names(dimnames(FLQuant()))
names(dnms)[1] <- "len"
```

```
idx.n <- FLQuant(idx.n, dimnames = dnms)
idx.wt <- FLQuant(idx.wt, dimnames = dnms)
idx.hrv <- FLQuant(idx.hrv, dimnames = dnms)

dnms <- dimnames(idxJmp.n)
names(dnms) <- names(dimnames(FLQuant()))
names(dnms)[1] <- "len"
idxJmp.n <- FLQuant(idxJmp.n, dimnames = dnms)
idxJmp.wt <- FLQuant(idxJmp.wt, dimnames = dnms)
idxJmp.hrv <- FLQuant(idxJmp.hrv, dimnames = dnms)

dnms <- dimnames(idxTrd.n)
names(dnms) <- names(dimnames(FLQuant()))
names(dnms)[1] <- "len"
idxTrd.n <- FLQuant(idxTrd.n, dimnames = dnms)
idxTrd.wt <- FLQuant(idxTrd.wt, dimnames = dnms)
idxTrd.hrv <- FLQuant(idxTrd.hrv, dimnames = dnms)
```

Some sanity checks to check that the resulting objects have matching dimensions.

```
# catch
cim(cth.orig, cth.n, cth.wt, hrv)

## [1] TRUE

# stock
cim(stk.orig, stk.n, stk.wt)

## [1] TRUE

# surveys
cim(idx.orig, idx.n, idx.wt, idx.hrv)

## [1] TRUE

cim(idxJmp.orig, idxJmp.n, idxJmp.wt, idxJmp.hrv)

## [1] TRUE

cim(idxTrd.orig, idxTrd.n, idxTrd.wt, idxTrd.hrv)

## [1] TRUE
```

Finally, we make FLR objects from the data.

```
# stock
rfLen.stk <- FLStockLen(stock.n = stk.n, stock.wt = stk.wt, stock = quantSums(stk.wt *
    stk.n), catch.n = cth.n, catch.wt = cth.wt/cth.n, catch = quantSums(cth.wt),
    harvest = hrv)
m(rfLen.stk)[] <- 0.05
mat(rfLen.stk)[] <- m.spwn(rfLen.stk)[] <- harvest.spwn(rfLen.stk)[] <- 0
mat(rfLen.stk)[38:59, , , 3:4] <- 1

# surveys
```

```
rfTrawl.idx <- FLIndex(index = idx.n, catch.n = idx.n, catch.wt = idx.wt, sel.pattern = idx.hrv)
effort(rfTrawl.idx)[] <- 100

rfTrawlJmp.idx <- FLIndex(index = idxJmp.n, catch.n = idxJmp.n, catch.wt = idxJmp.wt,
    sel.pattern = idxJmp.hrv)
effort(rfTrawlJmp.idx)[] <- 100

rfTrawlTrd.idx <- FLIndex(index = idxTrd.n, catch.n = idxTrd.n, catch.wt = idxTrd.wt,
    sel.pattern = idxTrd.hrv)
effort(rfTrawlTrd.idx)[] <- 100
```

3 Converting from length to age based data

The a4a stock assessment framework is based on age dynamics. Therefore, to use length information it must be processed before it can be used in an assessment. The rationale is that the processing should give the analyst the flexibility to use a range of sources of information, *e.g.* literature or online databases, to grab information about the species growth model and the uncertainty about the model parameters.

Within the a4a framework this is handled using the *a4aGr* class. In this section we introduce the *a4aGr* class and look at the variety of ways that parameter uncertainty can be included.

3.1 a4aGr - The growth class

The conversion of length data to age is performed through the use of a growth model. The implementation is done through the *a4aGr* class.

```
showClass("a4aGr")

## Class "a4aGr" [package "FLa4a"]
##
## Slots:
##
## Name:      grMod  grInvMod    params      vcov     distr      name
## Class:   formula   formula     FLPar     array character character
##
## Name:       desc     range
## Class: character   numeric
##
## Extends: "FLComp"
```

To construct an *a4aGr* object, the growth model and parameters must be provided. Check the help file for more information.

Here we show an example using the von Bertalanffy growth model. To create the *a4aGr* object it's necessary to pass the model equation ($length \sim time$), the inverse model equation ($time \sim length$) and the parameters. Any growth model can be used as long as it's possible to write the model (and the inverse) as an R formula.

```
vbObj <- a4aGr(grMod = ~linf * (1 - exp(-k * (t - t0))), grInvMod = ~t0 - 1/k *
    log(1 - len/linf), params = FLPar(linf = 58.5, k = 0.086, t0 = 0.001, units = c("cm",
    "ano-1", "ano")))

# Check the model and its inverse
lc = 20
predict(vbObj, len = lc)

##    iter
##         1
##   1 4.866

predict(vbObj, t = predict(vbObj, len = lc)) == lc

##    iter
##        1
##   1 TRUE
```

The predict method allows the transformation between age and lengths using the growth model.

```
predict(vbObj, len = 5:10 + 0.5)
```

```
##    iter
##       1
## 1 1.149
## 2 1.371
## 3 1.596
## 4 1.827
## 5 2.062
## 6 2.301
```

```
predict(vbObj, t = 5:10 + 0.5)
```

```
##    iter
##       1
## 1 22.04
## 2 25.05
## 3 27.80
## 4 30.33
## 5 32.66
## 6 34.78
```

3.2 Adding parameter uncertainty with a multivariate normal distribution

Uncertainty in the growth model is introduced through the inclusion of parameter uncertainty. This is done by making use of the parameter variance-covariance matrix (the `vcov` slot of the *a4aGr* class) and assuming a distribution. The numbers in the variance-covariance matrix could come from the parameter uncertainty from fitting the growth model parameters.

Here we set the variance-covariance matrix by scaling a correlation matrix, using a cv of 0.2.

```
# Make an empty cor matrix
cm <- diag(c(1, 1, 1))
# k and linf are negatively correlated while t0 is independent
cm[1, 2] <- cm[2, 1] <- -0.5
# scale cor to var using CV=0.2
cv <- 0.2
p <- c(linf = 60, k = 0.09, t0 = -0.01)
vc <- matrix(1, ncol = 3, nrow = 3)
l <- vc
l[1, ] <- l[, 1] <- p[1] * cv
k <- vc
k[, 2] <- k[2, ] <- p[2] * cv
t <- vc
t[3, ] <- t[, 3] <- p[3] * cv
mm <- t * k * l
diag(mm) <- diag(mm)^2
mm <- mm * cm
# check that we have the intended correlation
all.equal(cm, cov2cor(mm))
```

```
## [1] TRUE
```

Create the a4aGr object as before but now we also include the vcov argument for the variance-covariance matrix.

```
vbObj <- a4aGr(grMod = ~linf * (1 - exp(-k * (t - t0))), grInvMod = ~t0 - 1/k *
    log(1 - len/linf), params = FLPar(linf = p["linf"], k = p["k"], t0 = p["t0"],
    units = c("cm", "ano-1", "ano")), vcov = mm)
```

First we show a simple example where we assume that the parameters are represented using a multivariate normal distribution.

```
# Note that the object we have just created has a single iteration of each
# parameter
vbObj@params

## An object of class "FLPar"
## params
##  linf     k    t0
## 60.00  0.09 -0.01
## units:  cm ano-1 ano

dim(vbObj@params)

## [1] 3 1

# We simulate 10000 iterations from the a4aGr object by calling mvrnorm()
# using the the variance-covariance matrix we created earlier.
vbNorm <- mvrnorm(10000, vbObj)
# Now we have 10000 iterations of each parameter, randomly sampled from the
# multivariate normal distribution
vbNorm@params

## An object of class "FLPar"
## iters:  10000
##
## params
##                linf                    k                   t0
## 59.808108(11.91994)  0.090093( 0.01824) -0.010032( 0.00204)
## units:  cm ano-1 ano

dim(vbNorm@params)

## [1]     3 10000
```

We can now convert from length to ages data based on the 10000 parameter iterations. This gives us 10000 sets of age data. For example, here we convert a single length vector using each of the 10000 parameter iterations:

```
ages <- predict(vbNorm, len = 5:10 + 0.5)
dim(ages)

## [1]     6 10000

# We show the first ten iterations only as an illustration
ages[, 1:10]

##    iter
##         1     2      3     4     5     6      7     8     9    10
```

```
##   1 1.297 1.193 0.9554 1.296 1.253 1.554 0.9908 1.268 1.241 1.344
##   2 1.556 1.432 1.1433 1.551 1.497 1.866 1.1817 1.517 1.486 1.606
##   3 1.822 1.678 1.3353 1.813 1.747 2.187 1.3756 1.770 1.738 1.873
##   4 2.096 1.932 1.5315 2.082 2.001 2.518 1.5726 2.029 1.995 2.147
##   5 2.378 2.194 1.7321 2.358 2.260 2.861 1.7729 2.295 2.259 2.425
##   6 2.669 2.465 1.9373 2.642 2.525 3.214 1.9766 2.566 2.530 2.710
```

The marginal distributions can be seen in Figure 2.

```
par(mfrow = c(3, 1))
hist(c(params(vbNorm)["linf", ]), main = "linf", xlab = "")
hist(c(params(vbNorm)["k", ]), main = "k", prob = TRUE, xlab = "")
hist(c(params(vbNorm)["t0", ]), main = "t0", xlab = "")
```

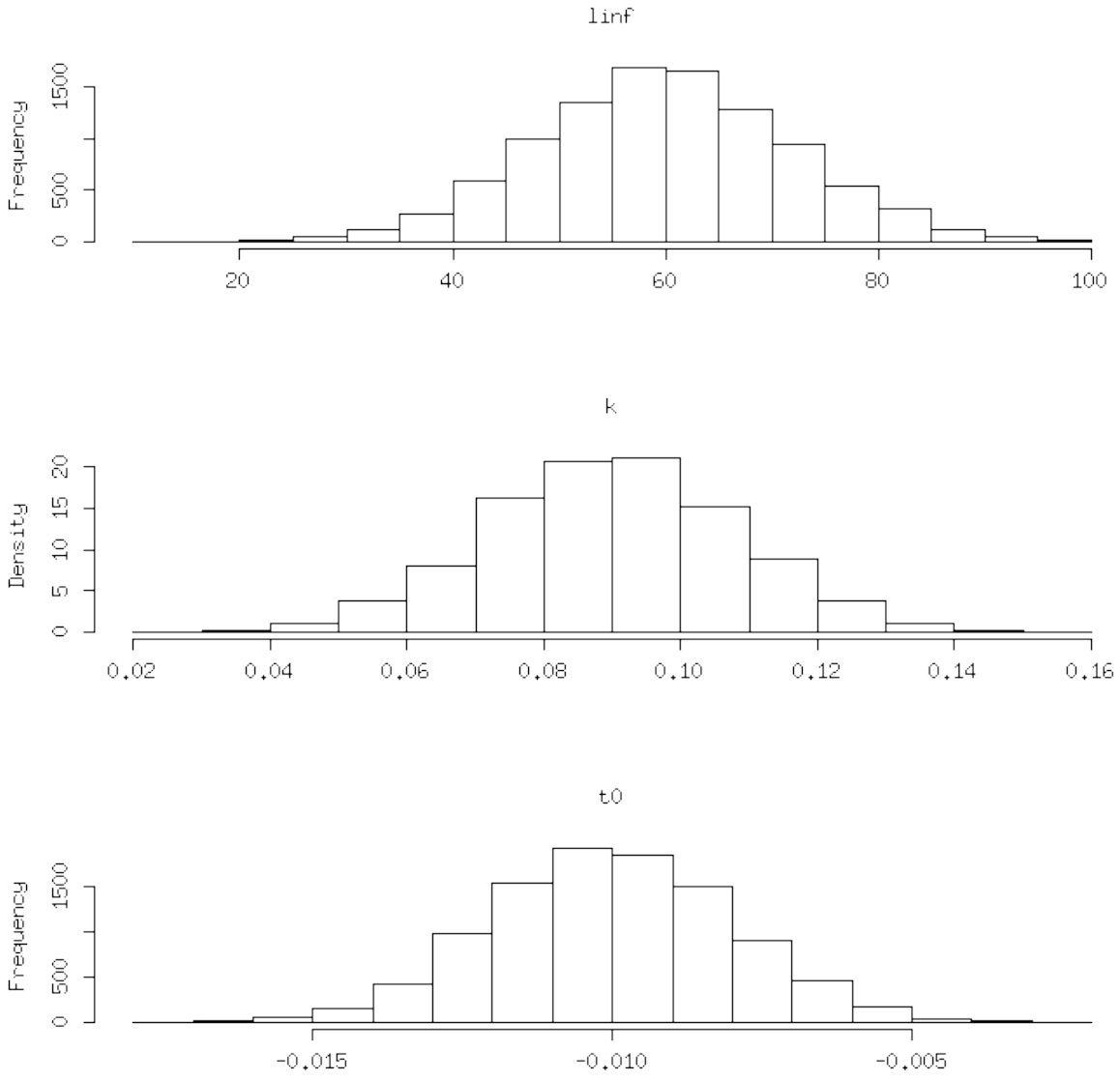

Figure 2: The marginal distributions of each of the parameters from using a multivariate normal distribution.

The shape of the correlation can be seen in Figure 3.

```
splom(data.frame(t(params(vbNorm)@.Data)), pch = ".")
```

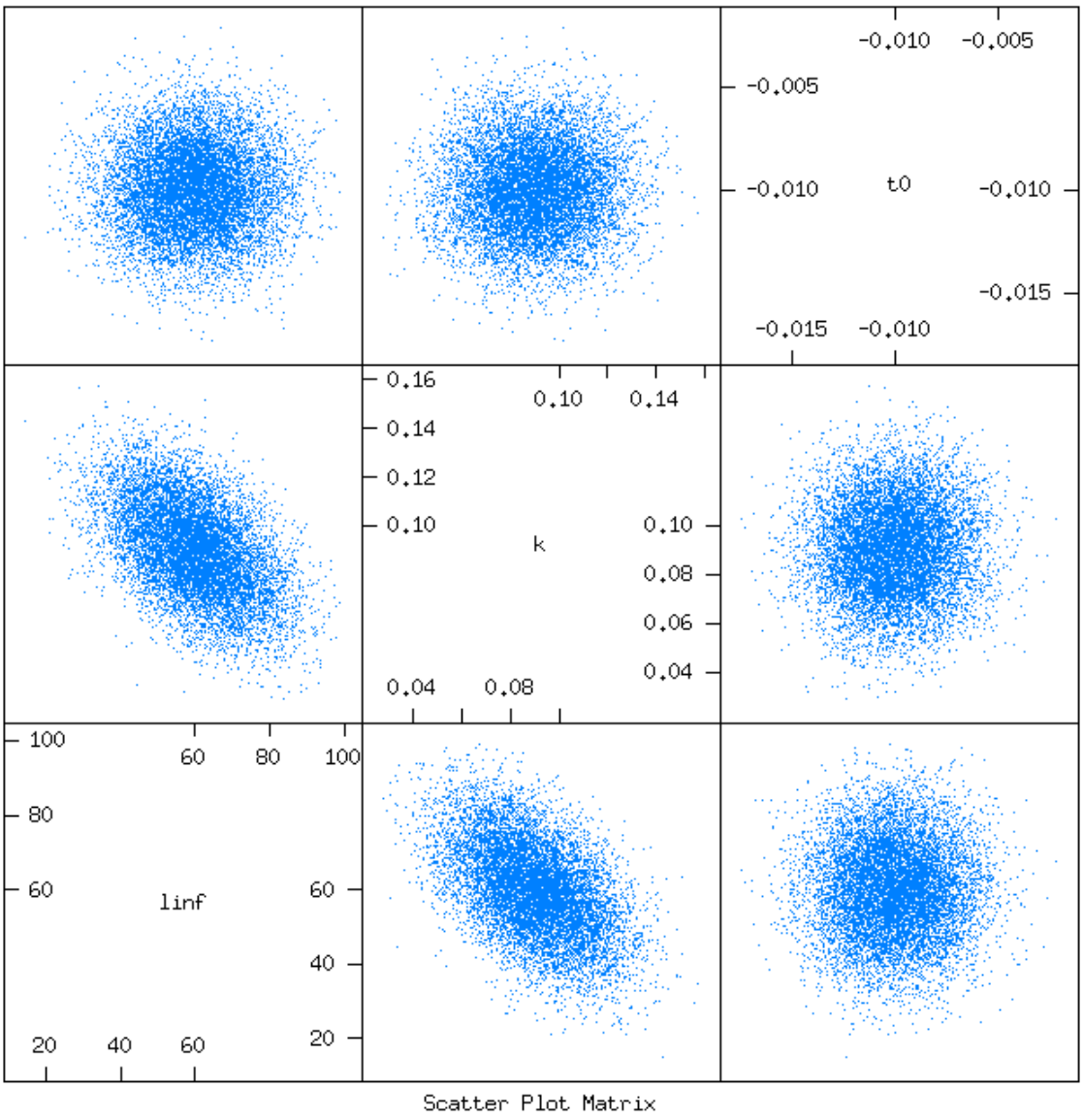

Figure 3: Scatter plot of the 10000 samples parameter from the multivariate normal distribution.

Growth curves for the 1000 iterations can be seen in Figure 4.

```
boxplot(t(predict(vbNorm, t = 0:50 + 0.5)))
```

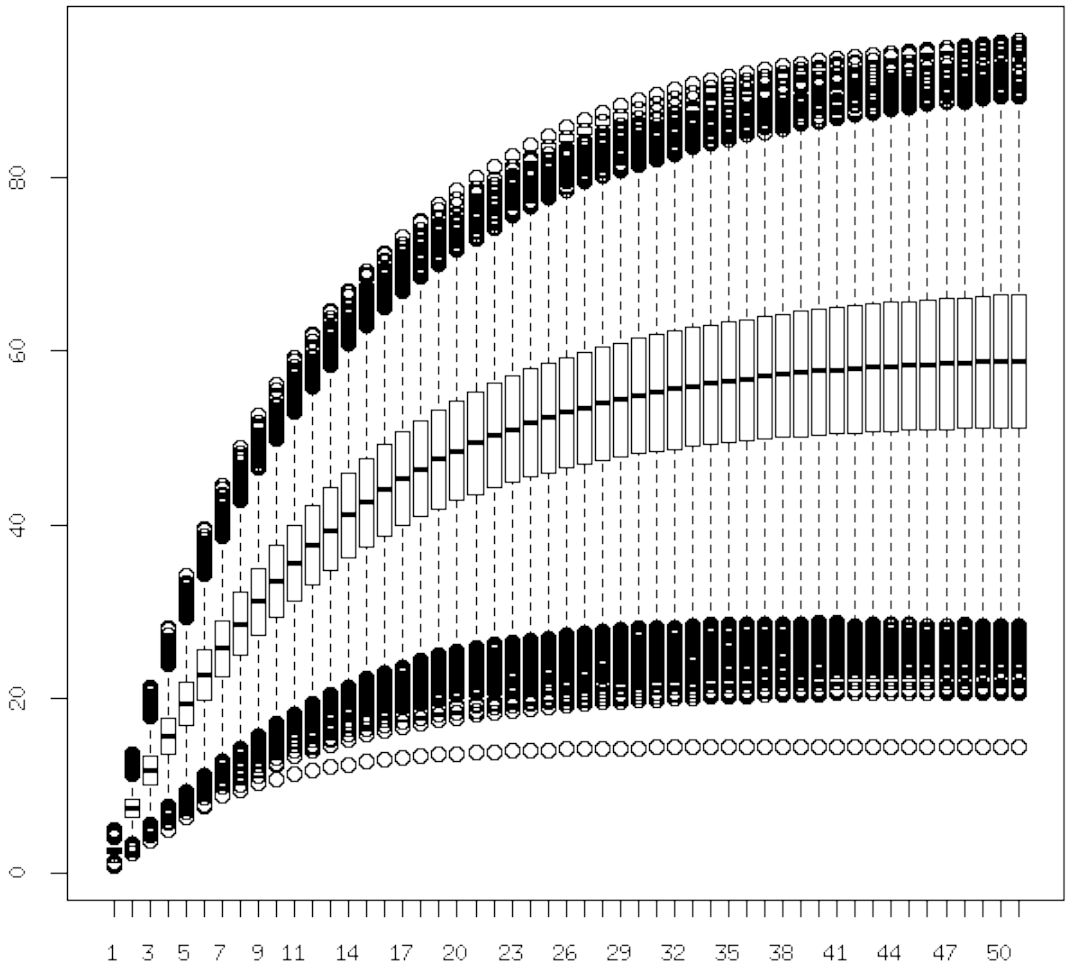

Figure 4: Growth curves using parameters simulated from a multivariate normal distribution.

3.3 Adding parameter uncertainty with a multivariate triangle distribution

One alternative to using a normal distribution is to use a triangle distribution. We use the package `triangle`, where this distribution is parametrized using the minimum, maximum and median values. This can be very attractive if the analyst needs to scrape information from the web or literature and perform some kind of meta-analysis.

Here we show an example of setting a triangle distribution with values taken from Fishbase.

```
# The web address for the growth parameters for redfish (Sebastes
# norvegicus)
addr <- "http://www.fishbase.org/PopDyn/PopGrowthList.php?ID=501"
# Scrape the data
tab <- try(readHTMLTable(addr))
# Interrogate the data table and get vectors of the values
linf <- as.numeric(as.character(tab$dataTable[, 2]))
```

```
k <- as.numeric(as.character(tab$dataTable[, 4]))
t0 <- as.numeric(as.character(tab$dataTable[, 5]))
# Set the min (a), max (b) and median (c) values for the parameter as a list
# of lists Note that t0 has no 'c' (median) value. This makes the
# distribution symmetrical
triPars <- list(list(a = min(linf), b = max(linf), c = median(linf)), list(a = min(k),
    b = max(k), c = median(k)), list(a = median(t0, na.rm = T) - IQR(t0, na.rm = T)/2,
    b = median(t0, na.rm = T) + IQR(t0, na.rm = T)/2))
# Simulate 10000 times using mvrtriangle
vbTri <- mvrtriangle(10000, vbObj, paramMargins = triPars)
```

The marginals will reflect the uncertainty on the parameter values that were scraped from Fishbase but, as we don't really believe the parameters are multivariate normal, here we adopted a distribution based on a t copula with triangle marginals. The marginal distributions can be seen in Figure 5 and the shape of the correlation can be seen in Figure 6.

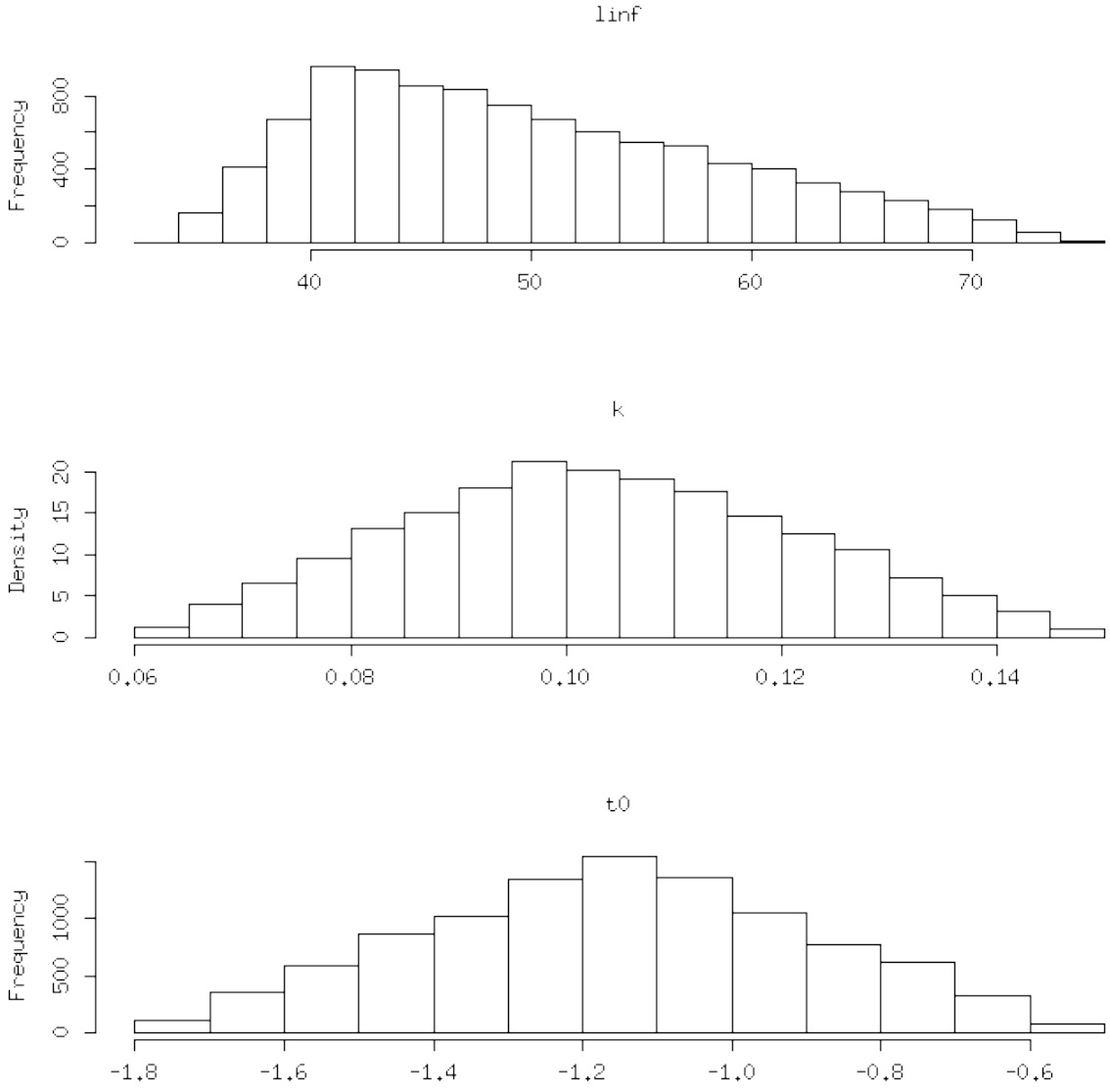

Figure 5: The marginal distributions of each of the parameters from using a multivariate triangle distribution.

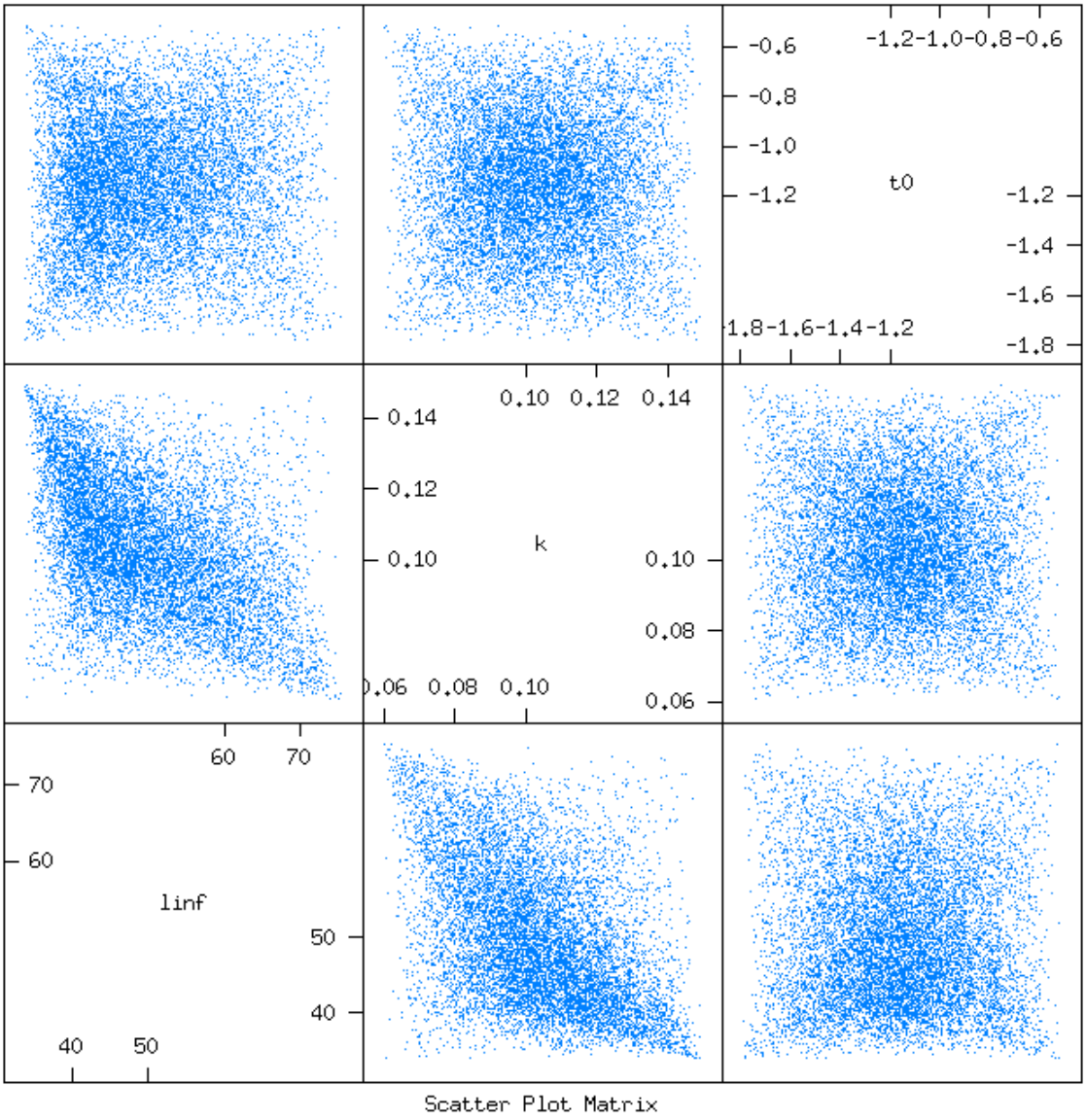

Figure 6: Scatter plot of the 10000 samples parameter from the multivariate triangle distribution.

We can still use `predict()` to see the growth model uncertainty (Figure 7).

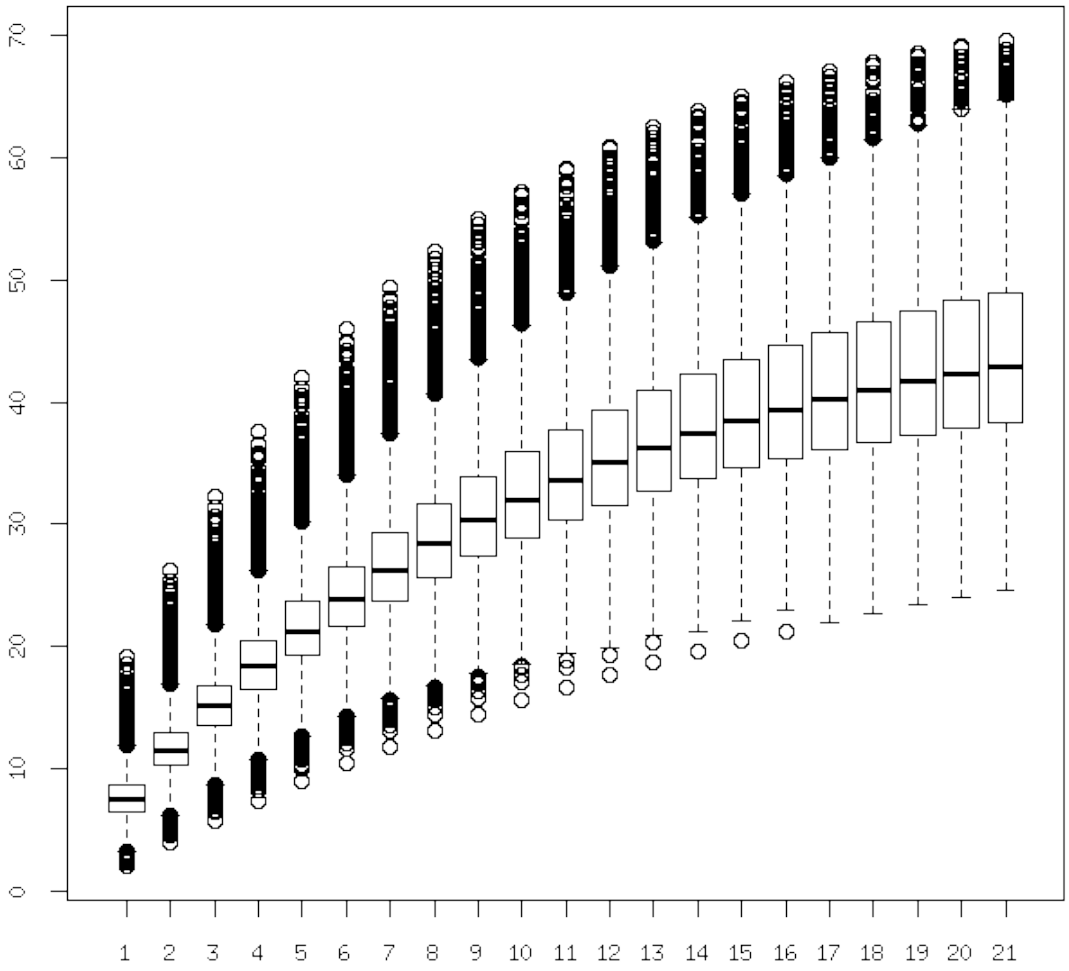

Figure 7: Growth curves using parameters simulated from a multivariate triangle distribution.

Remember that the above examples use a variance-covariance matrix that we essentially made up. An alternative would be to scrape the entire growth parameters dataset from Fishbase and compute the shape of the variance-covariance matrix yourself.

3.4 Adding parameter uncertainty with copulas

A more general approach to adding parameter uncertainty is to make use of statistical copulas[2] and marginal distributions of choice. This is possible with the `mvrcop()` function borrowed from the package `copula`. The example below keeps the same parameters and changes only the copula type and family but a lot more can be done. Check the package `copula` for more information.

```
vbCop <- mvrcop(10000, vbObj, copula = "archmCopula", family = "clayton", param = 2,
    margins = "triangle", paramMargins = triPars)
```

[2] http://en.wikipedia.org/wiki/Copula_%28probability_theory%29

The shape of the correlation changes (Figure 8) as well as the resulting growth curves (Figure 9).

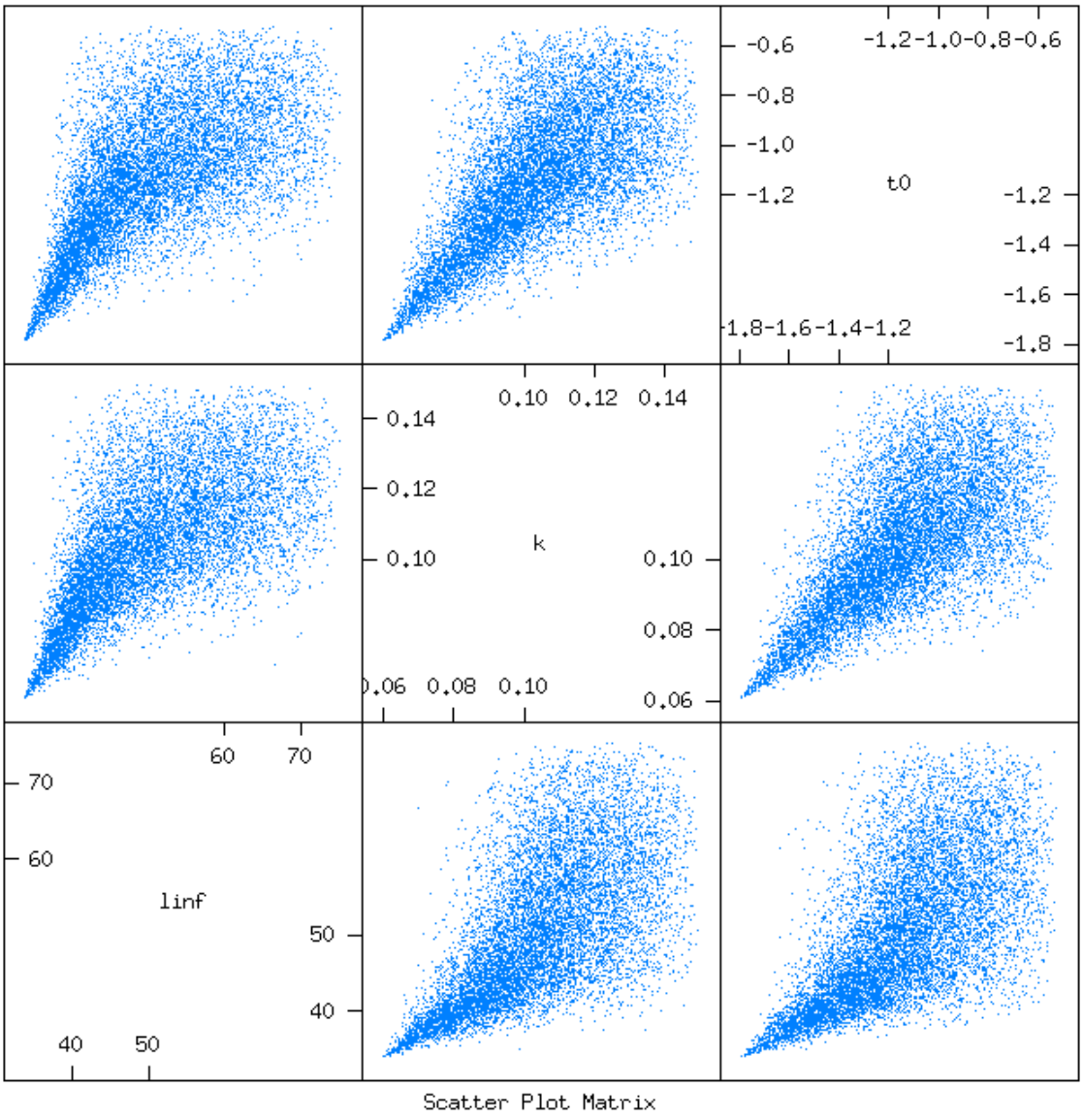

Figure 8: Scatter plot of the 10000 samples parameter from the using an archmCopula copula with triangle margins.

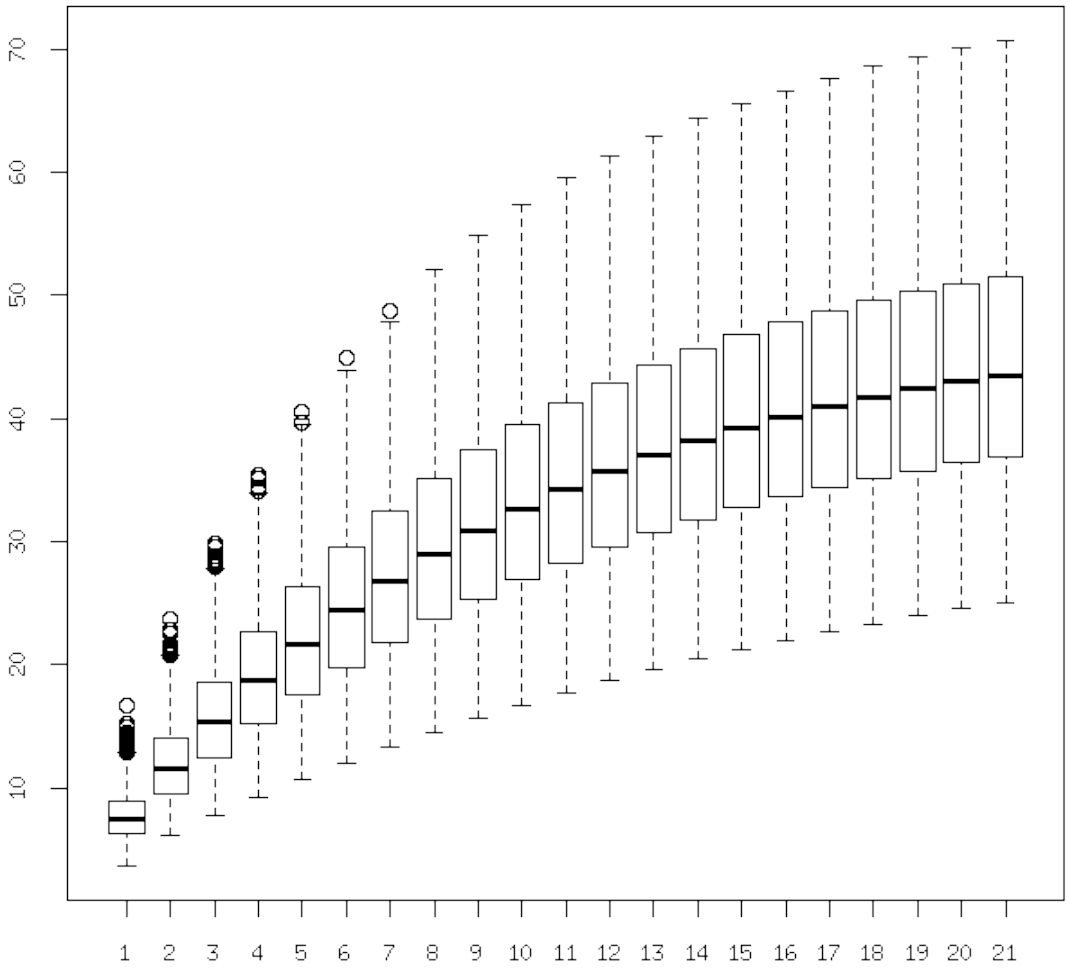

Figure 9: Growth curves from the using an archmCopula copula with triangle margins.

3.5 The `l2a()` method

After introducing uncertainty in the growth model through the parameters it's time to transform the length-based dataset into an age-based dataset. The method that deals with this process is `l2a()`. The implementation of this method for the *FLQuant* class is the main workhorse. There are two other implementations, for the *FLStock* and *FLIndex* classes, which are mainly wrappers that call the *FLQuant* method several times.

When converting from length-based data to age-based data you need to be aware of how the aggregation of length classes is performed. For example, individuals in length classes 1-2, 2-3, and 3-4 cm may all be considered as being of age 1 (obviously depending on the growth model). How should the values in those length classes be combined?

If the values are abundances then the values should be summed. Summing other types of values, such as mean weight, does not make sense. Instead these values are averaged over the length classes (possibly weighted by the abundance). This is controlled using the `stat` argument which can be either `mean` or `sum`

(the default). Fishing mortality is not computed to avoid making wrong assumptions about the meaning of F at length.

We demonstrate the method by converting a catch-at-length *FLQuant* to a catch-at-age *FLQuant*. First we make an *a4aGr* object with a multivariate triangle distribution (using the parameters we set above). We use 10 iterations as an example. And call `l2a()` by passing in the length-based *FLQuant* and the *a4aGr* object.

```
vbTriSmall <- mvrtriangle(10, vbObj, paramMargins = triPars)
cth.n <- l2a(catch.n(rfLen.stk), vbTriSmall)
```

```
dim(cth.n)

## [1] 62 26  1  4  1 10
```

In the previous example, the *FLQuant* object that was sliced (`catch.n(rfLen.stk)`) had only one iteration. This iteration was sliced by each of the iterations in the growth model. It is possible for the *FLQuant* object to have the same number of iterations as the growth model, in which case each iteration of the *FLQuant* and the growth model are used together. It is also possible for the growth model to have only one iteration while the *FLQuant* object has many iterations. The same growth model is then used for each of the *FLQuant* iterations. As with all `FLR` objects, the general rule is *one or n* iterations.

As well as converting one *FLQuant* at a time, we can convert entire *FLStock* and *FLIndex* objects. In these cases the individual *FLQuant* slots of those classes are converted from length-based to age-based. As mentioned above, the aggregation method depends on the type of values the slots contain. The abundance slots (`*.n`, such as `stock.n`) are summed. The `*.wt`, `m`, `mat`, `harvest.spwn` and `m.spwn` slots of an *FLStock* object are averaged. The `index`, `catch.wt`, `index.var`, `sel.pattern` and `index.q` slots of an *FLIndex* object are averaged[3].

The method for *FLStock* classes takes an additional argument for the plusgroup.

```
aStk <- l2a(rfLen.stk, vbTriSmall, plusgroup = 14)

## [1] "Some ages are less than 0, indicating a mismatch between input data lengths and growth paramete
## [1] "Trimming age range to a minimum of 0"
## [1] "maxfbar has been changed to accomodate new plusgroup"

aIdx <- l2a(rfTrawl.idx, vbTriSmall)

## [1] "Some ages are less than 0, indicating a mismatch between input data lengths and growth paramete
## [1] "Trimming age range to a minimum of 0"
```

When converting with `l2a()` all lengths above Linf are converted to the maximum age, as there is no information in the growth model about how to deal with individuals larger than Linf.

[3] Still working on l2a for index. Not all of these slots can be averaged

4 Natural mortality

In the a4a natural mortality is dealt with as an external parameter to the stock assessment model. The rationale to modelling natural mortality is similar to that of growth: one should be able to grab information from a range of sources and feed it into the assessment.

The mechanism used by a4a is to build an interface that makes it transparent, flexible and hopefully easy to explore different options. In relation to natural mortality it means that the analyst should be able to use distinct models like Gislasson's, Charnov's, Pauly's, etc in a coherent framework making it possible to compare the outcomes of the assessment.

Within the a4a framework, the general method for inserting natural mortality in the stock assessment is to:

- Create an object of class *a4aM* which holds the natural mortality model and parameters.
- Add uncertainty to the parameters in the *a4aM* object.
- Apply the m() method to the *a4aM* object to create an age or length based *FLQuant* object of the required dimensions.

The resulting *FLQuant* object can then be directly inserted into an *FLStock* object to be used for the assessment.

In this section we go through each of the steps in detail using a variety of different models.

4.1 a4aM - The M class

Natural mortality is implemented in a class named *a4aM*. This class is made up of three objects of the class *FLModelSim*. Each object is a model that represents one effect: an age or length effect, a scaling (level) effect and a time trend, named shape, level and trend, respectively. The impact of the models is multiplicative, i.e. the overal natural mortality is given by *shape* x *level* x *trend*. Check the help files for more information.

```
showClass("a4aM")
```

```
## Class "a4aM" [package "FLa4a"]
##
## Slots:
##
## Name:       shape      level      trend       name       desc      range
## Class: FLModelSim FLModelSim FLModelSim  character  character    numeric
##
## Extends: "FLComp"
```

The *a4aM* constructor requires that the models and parameters are provided. The default method will build each of these models as a constant value of 1.

As a simple example, the usual "0.2" guessestimate could be set up by setting the level model to have a single parameter with a fixed value, while the other two models, *shape* and *trend*, have a default value of 1 (meaning that they have no effect).

```
mod02 <- FLModelSim(model = ~a, params = FLPar(a = 0.2))
m1 <- a4aM(level = mod02)
m1
```

```
## a4aM object:
##   shape: ~1
##   level: ~a
##   trend: ~1
```

More interesting natural mortality shapes can be set up using biological knowledge. The following example uses an exponential decay over ages (implying that the resulting *FLQuant* generated by the `m()` method will be age based). We also use Jensen's second estimator (Kenshington, 2013) as a scaling `level` model, which is based on the von Bertalanffy K parameter, $M = 1.5K$.

```
shape2 <- FLModelSim(model = ~exp(-age - 0.5))
level2 <- FLModelSim(model = ~1.5 * k, params = FLPar(k = 0.4))
m2 <- a4aM(shape = shape2, level = level2)
m2


## a4aM object:
##   shape: ~exp(-age - 0.5)
##   level: ~1.5 * k
##   trend: ~1
```

Note that the `shape` model has `age` as a parameter of the model but is not set using the `params` argument.

The `shape` model does not have to be age-based. For example, here we set up a `shape` model using Gislason's second estimator (Kenshington, 2013): $M_l = K(\frac{L_{\inf}}{l})^{1.5}$. We use the default `level` and *trend* models.

```
shape_len <- FLModelSim(model = ~K * (linf/len)^1.5, params = FLPar(linf = 60,
    K = 0.4))
m_len <- a4aM(shape = shape_len)
```

Another option is to model how an external factor may impact the natural mortality. This can be added through the `trend` model. Suppose natural mortality can be modelled with a dependency on the NAO index, due to some mechanism that results in having lower mortality when NAO is negative and higher when it's positive. In this example, the impact is represented by the NAO value on the quarter before spawning, which occurs in the second quarter.

We use this to make a complicated natural mortality model with an age based shape model, a level model based on K and a trend model driven by NAO, where mortality increases by 50% if NAO is positive on the first quarter.

```
# Get NAO
nao.orig <- read.table("http://www.cdc.noaa.gov/data/correlation/nao.data",
    skip = 1, nrow = 62, na.strings = "-99.90")
dnms <- list(quant = "nao", year = 1948:2009, unit = "unique", season = 1:12,
    area = "unique")
# Build an FLQuant from the NAO data
nao.flq <- FLQuant(unlist(nao.orig[, -1]), dimnames = dnms, units = "nao")
# Build covar by calculating mean over the first 3 months
nao <- seasonMeans(nao.flq[, , , 1:3])
# Turn into Boolean
nao <- (nao > 0)
# Constructor
trend3 <- FLModelSim(model = ~1 + b * nao, params = FLPar(b = 0.5))
shape3 <- FLModelSim(model = ~exp(-age - 0.5))
level3 <- FLModelSim(model = ~1.5 * k, params = FLPar(k = 0.4))
m3 <- a4aM(shape = shape3, level = level3, trend = trend3)
m3


## a4aM object:
##   shape: ~exp(-age - 0.5)
##   level: ~1.5 * k
##   trend: ~1 + b * nao
```

4.2 Adding multivariate normal parameter uncertainty

Uncertainty on natural mortality is added through uncertainty on the parameters.

In this section we show how to add multivariate normal uncertainty. We make use of the class *FLModelSim* method `mvrnorm()`, which is a wrapper for the method `mvrnorm()` distributed by the package `MASS`.

We create an *a4aM* object with an exponential shape, a `level` model based on k and temperature (Jensen's third estimator), and a `trend` model driven by the NAO (as above). We include a variance-covariance matrix for the `level` and `trend` models. We create a 100 iterations using the `mvrnorm()` method.

```
shape4 <- FLModelSim(model = ~exp(-age - 0.5))
level4 <- FLModelSim(model = ~k^0.66 * t^0.57, params = FLPar(k = 0.4, t = 10),
    vcov = array(c(0.002, 0.01, 0.01, 1), dim = c(2, 2)))
trend4 <- FLModelSim(model = ~1 + b * nao, params = FLPar(b = 0.5), vcov = matrix(0.02))
m4 <- a4aM(shape = shape4, level = level4, trend = trend4)
# Call mvrnorm()
m4 <- mvrnorm(100, m4)
m4

## a4aM object:
##   shape: ~exp(-age - 0.5)
##   level: ~k^0.66 * t^0.57
##   trend: ~1 + b * nao

# Look at the level model (for example)
m4@level

## An object of class "FLModelSim"
## Slot "model":
## ~k^0.66 * t^0.57
##
## Slot "params":
## An object of class "FLPar"
## iters:  100
##
## params
##                k                t
##  0.4012(0.0368) 10.1241(1.1244)
## units:  NA
##
## Slot "vcov":
##       [,1] [,2]
## [1,] 0.002 0.01
## [2,] 0.010 1.00
##
## Slot "distr":
## [1] "norm"

# Note the variance in the parameters. The trend model also has uncertainty
params(trend(m4))

## An object of class "FLPar"
## iters:  100
##
## params
##               b
## 0.53099(0.141)
## units:  NA
```

```
# However, the shape model has no parameters and no uncertainty
params(shape(m4))


## An object of class "FLPar"
## param
##
## NA
## units:  NA
```

In this particular case, the `shape` model will not be randomized because it doesn't have a variance-covariance matrix. Also note that because there is only one parameter in the `trend` model, the randomization will use a univariate normal distribution.

The same model could be achieved by using `mnrnorm()` on each model component:

```
m4 <- a4aM(shape = shape4, level = mvrnorm(100, level4), trend = mvrnorm(100,
    trend4))
```

4.3 Adding parameter uncertainty with copulas

We can also use copulas to add parameter uncertainty to the natural mortality model, similar to the way we use them for the growth model in Section 3.3. As stated above these processes make use of the methods implemented for the *FLModelSim* class.

In the following example we'll use again Gislason's second estimator, $M_l = K(\frac{L_{\inf}}{l})^{1.5}$ and a triangle copula to model parameter uncertainty. The method `mvrtriangle()` is used to create 1000 iterations.

```
linf <- 60
k <- 0.4
# vcov matrix (make up some values)
mm <- matrix(NA, ncol = 2, nrow = 2)
# 10% cv
diag(mm) <- c((linf * 0.1)^2, (k * 0.1)^2)
# 0.2 correlation
mm[upper.tri(mm)] <- mm[lower.tri(mm)] <- c(0.05)
# a good way to check is using cov2cor
cov2cor(mm)


##        [,1]   [,2]
## [1,] 1.0000 0.2083
## [2,] 0.2083 1.0000


# create object
mgis2 <- FLModelSim(model = ~k * (linf/len)^1.5, params = FLPar(linf = linf,
    k = k), vcov = mm)
# set the lower, upper and (optionally) centre of the parameters (without
# the centre, the triangle is symmetrical)
pars <- list(list(a = 55, b = 65), list(a = 0.3, b = 0.6, c = 0.35))
mgis2 <- mvrtriangle(1000, mgis2, paramMargins = pars)
mgis2


## An object of class "FLModelSim"
## Slot "model":
## ~k * (linf/len)^1.5
##
## Slot "params":
```

```
## An object of class "FLPar"
## iters:  1000
##
## params
##             linf                k
## 60.05763(2.2026)  0.40862(0.0751)
## units:  NA
##
## Slot "vcov":
##       [,1]   [,2]
## [1,] 36.00 0.0500
## [2,]  0.05 0.0016
##
## Slot "distr":
## [1] "un t copula family  triangle"
```

The resulting parameter estimates and marginal distributions can be seen in Figure 10 and 11

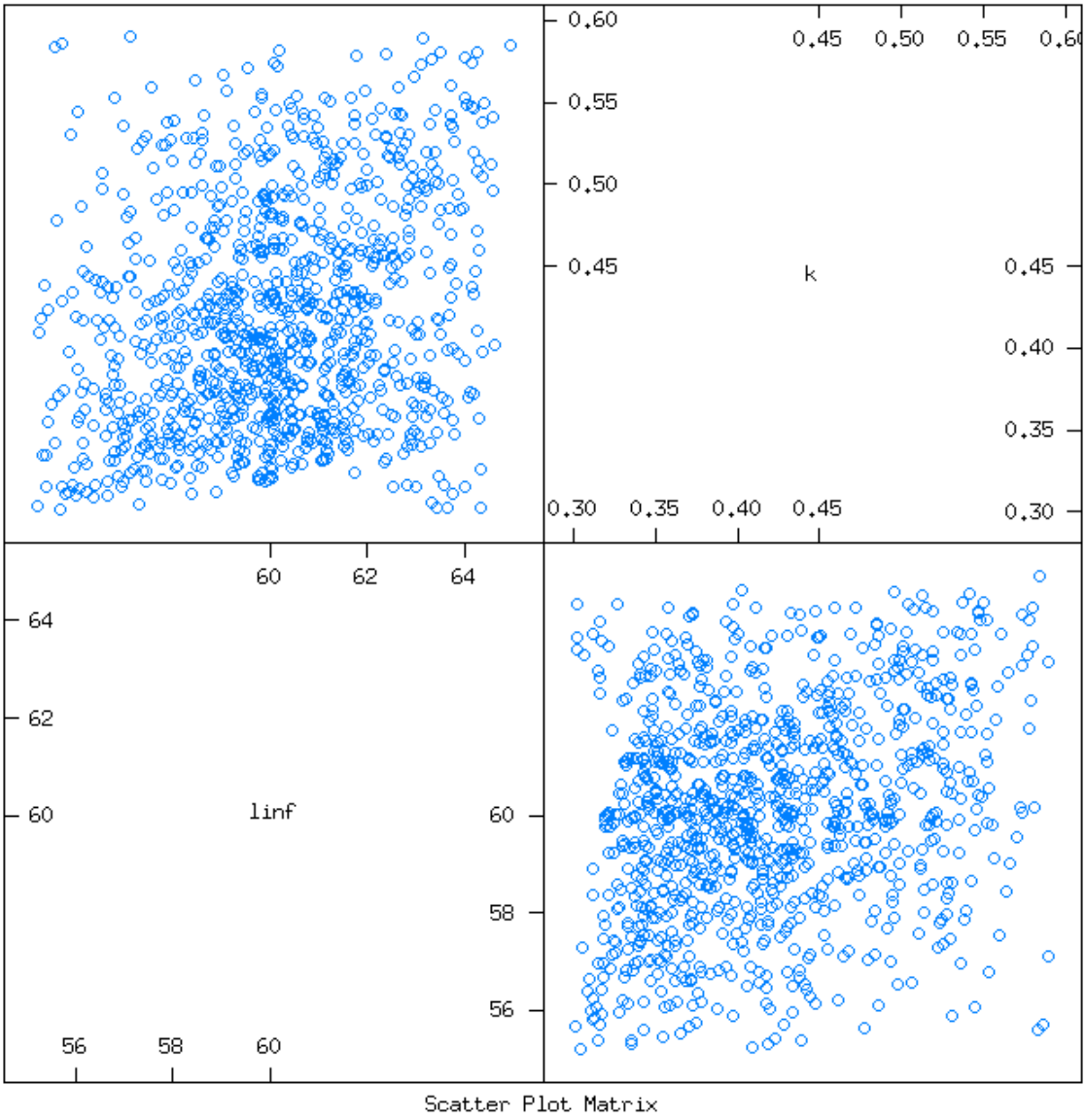

Figure 10: Parameter estimates for Gislason's second natural mortality model from using a triangle distribution.

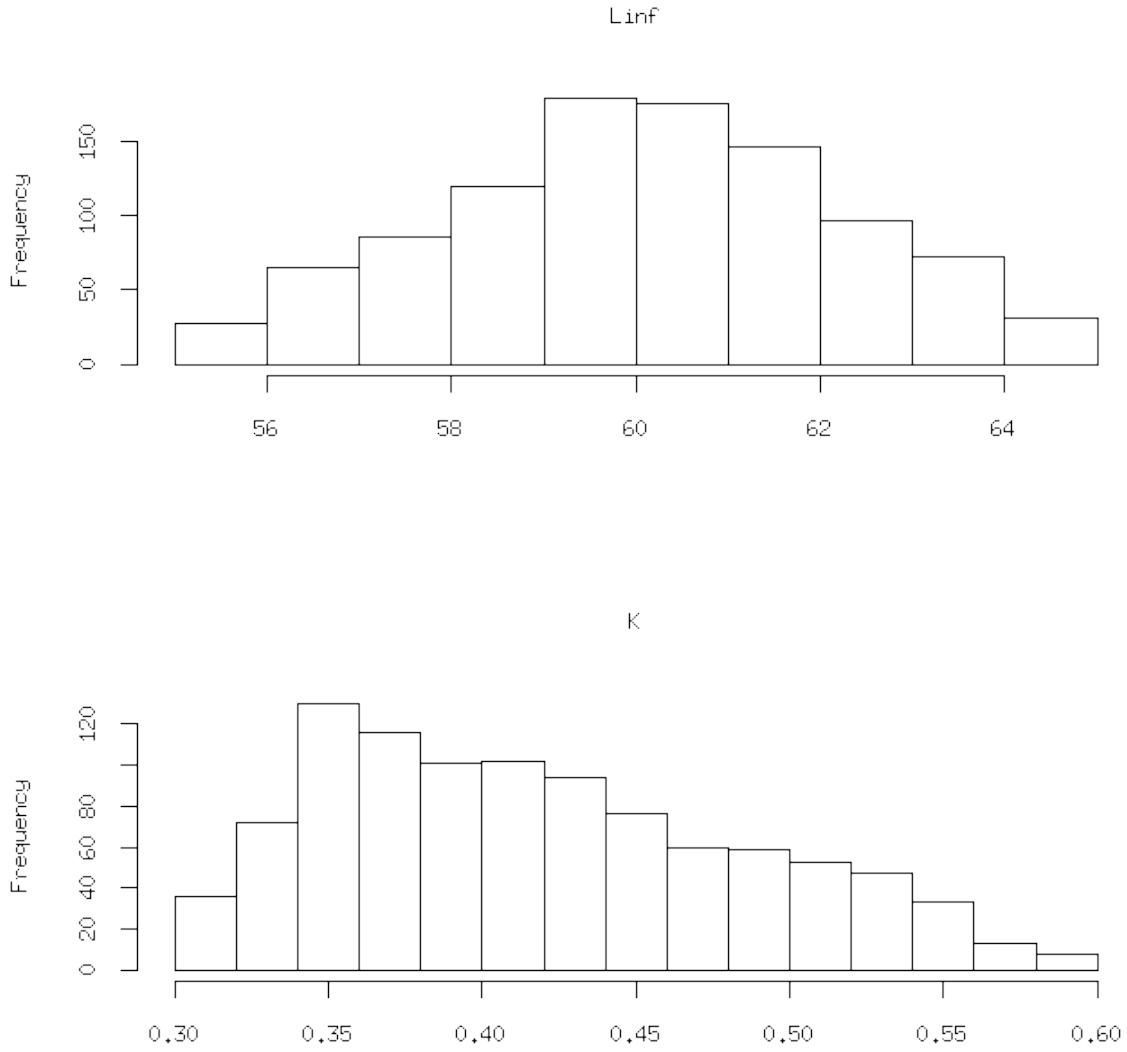

Figure 11: Marginal distributions of the parameters for Gislason's second natural mortality model using a triangle distribution.

We now have a new model that can be used for the `shape` model. You can use the constructor or the set method to add the new model. Note that we have a quite complex method now for M. A length based `shape` model from Gislason's work, Jensen's third model based on temperature `level` and a time `trend` depending on NAO. All of the component models have uncertainty in their parameters.

```
m5 <- a4aM(shape = mgis2, level = level4, trend = trend4)
# or
m5 <- m4
shape(m5) <- mgis2
```

4.4 The "m" method

Now that we have set up the natural mortality *a4aM* model and added parameter uncertainty to each component, we are ready to generate the *FLQuant* of natural mortality. For that we need the `m()` method.

The `m()` method is the workhorse method for computing natural mortality. The method returns an *FLQuant* that can be inserted in an *FLStock* for usage by the assessment method.

The size of the *FLQuant* object is determined by the `min`, `max`, `minyear` and `maxyear` elements of the `range` slot of the *a4aM* object. By default the values of these elements are set to 0. Giving an *FLQuant* with length 1 in the `quant` and `year` dimension. The `range` slot can be set by hand, or by using the `rngquant()` and `rngyear()` methods.

The name of the first dimension of the output *FLQuant* (e.g. 'age' or 'len') is determined by the parameters of the `shape` model. If it is not clear what the name should be then the name is set to 'quant'.

Here we demonstrate `m()` using the simple *a4aM* object we created above that has constant natural mortality:

```
# Start with the simplest model
m1

## a4aM object:
##   shape: ~1
##   level: ~a
##   trend: ~1

# Check the range
range(m1)

##       min       max plusgroup   minyear   maxyear   minmbar   maxmbar
##         0         0         0         0         0         0         0

# Simple - no ages or years
m(m1)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##      year
## quant 0
##     0 0.2
##
## units:  NA

# Set the quant range
rngquant(m1) <- c(0, 7)  # set the quant range
range(m1)

##       min       max plusgroup   minyear   maxyear   minmbar   maxmbar
##         0         7         0         0         0         0         0

m(m1)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##      year
## quant 0
##     0 0.2
##     1 0.2
##     2 0.2
```

```
##     3 0.2
##     4 0.2
##     5 0.2
##     6 0.2
##     7 0.2
##
## units:  NA

# Set the year range too
rngyear(m1) <- c(2000, 2010)  # set the year range
range(m1)

##       min       max plusgroup   minyear   maxyear   minmbar   maxmbar
##         0         7         0      2000      2010         0         0

m(m1)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##      year
## quant 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010
##     0 0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
##     1 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
##     3 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
##     5 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
##     7 0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2  0.2
##
## units:  NA

# Note the name of the first dimension is 'quant'
```

The next example has an age-based shape. As the `shape` model has 'age' as a variable which is not included in the *FLPar* slot it is used as the name of the first dimension of the resulting *FLQuant*. Note that in this case the `mbar` values in the range become relevant. `mbar` the range of quants (in this case, ages) that is used to compute the mean level. This mean level will match the value given by the `level` model. The `mbar` range can be changed with the `rngmbar()` method.

We illustrate this by making an *FLQuant* with age varying natural mortality:

```
# Remind ourselves of the model
m2

## a4aM object:
##   shape: ~exp(-age - 0.5)
##   level: ~1.5 * k
##   trend: ~1

# Simple with no ages or years - note that the first dimension will be 'age'
m(m2)

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

```
##
##    year
## age 0
##   0 0.6
##
## units:  NA

# With ages
rngquant(m2) <- c(0, 7)
m(m2)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##    year
## age 0
##   0 0.60000000
##   1 0.22072766
##   2 0.08120117
##   3 0.02987224
##   4 0.01098938
##   5 0.00404277
##   6 0.00148725
##   7 0.00054713
##
## units:  NA

# With ages and years
rngyear(m2) <- c(2000, 2003)
m(m2)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##    year
## age 2000       2001       2002       2003
##   0 0.60000000 0.60000000 0.60000000 0.60000000
##   1 0.22072766 0.22072766 0.22072766 0.22072766
##   2 0.08120117 0.08120117 0.08120117 0.08120117
##   3 0.02987224 0.02987224 0.02987224 0.02987224
##   4 0.01098938 0.01098938 0.01098938 0.01098938
##   5 0.00404277 0.00404277 0.00404277 0.00404277
##   6 0.00148725 0.00148725 0.00148725 0.00148725
##   7 0.00054713 0.00054713 0.00054713 0.00054713
##
## units:  NA

# Note that the level value is:
predict(level(m2))

##    iter
##       1
##   1 0.6

# Is the same as
m(m2)["0"]
```

```
## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##    year
## age 2000 2001 2002 2003
##   0 0.6  0.6  0.6  0.6
##
## units:  NA

# This is because the mbar range is currently set to '0' and '0'
range(m2)

##       min       max plusgroup   minyear   maxyear   minmbar   maxmbar
##         0         7         0      2000      2003         0         0

# The mean natural mortality value over this range is given by the level
# model We can change the mbar range
rngmbar(m2) <- c(0, 5)
range(m2)

##       min       max plusgroup   minyear   maxyear   minmbar   maxmbar
##         0         7         0      2000      2003         0         5

# This rescales the natural mortality at age:
m(m2)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##    year
## age 2000      2001      2002      2003
##   0 2.2812888 2.2812888 2.2812888 2.2812888
##   1 0.8392392 0.8392392 0.8392392 0.8392392
##   2 0.3087389 0.3087389 0.3087389 0.3087389
##   3 0.1135787 0.1135787 0.1135787 0.1135787
##   4 0.0417833 0.0417833 0.0417833 0.0417833
##   5 0.0153712 0.0153712 0.0153712 0.0153712
##   6 0.0056547 0.0056547 0.0056547 0.0056547
##   7 0.0020803 0.0020803 0.0020803 0.0020803
##
## units:  NA

# Check that the mortality over the mean range is the same as the level
# model
quantMeans(m(m2)[as.character(0:5)])

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##      year
## age   2000 2001 2002 2003
##   all 0.6  0.6  0.6  0.6
##
## units:  NA
```

The next example uses a time trend for the `trend` model. We use the `m3` model we made earlier. The `trend` model for this model has a covariate, 'nao'. This needs to be passed to the `m()` method. The year range of the 'nao' covariate should match that of the `range` slot.

```
# Simple, pass in a single nao value (only one year)
m(m3, nao = 1)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##    year
## age 0
##   0 0.9
##
## units:  NA

# Set some ages
rngquant(m3) <- c(0, 7)
m(m3, nao = 0)

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##    year
## age 0
##   0 0.60000000
##   1 0.22072766
##   2 0.08120117
##   3 0.02987224
##   4 0.01098938
##   5 0.00404277
##   6 0.00148725
##   7 0.00054713
##
## units:  NA

# With ages and years - passing in the NAO data as numeric (1,0,1,0)
rngyear(m3) <- c(2000, 2003)
m(m3, nao = as.numeric(nao[, as.character(2000:2003)]))

## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##    year
## age 2000       2001       2002       2003
##   0 0.90000000 0.60000000 0.90000000 0.60000000
##   1 0.33109150 0.22072766 0.33109150 0.22072766
##   2 0.12180175 0.08120117 0.12180175 0.08120117
##   3 0.04480836 0.02987224 0.04480836 0.02987224
##   4 0.01648407 0.01098938 0.01648407 0.01098938
##   5 0.00606415 0.00404277 0.00606415 0.00404277
##   6 0.00223088 0.00148725 0.00223088 0.00148725
##   7 0.00082069 0.00054713 0.00082069 0.00054713
##
## units:  NA
```

The final example show how `m()` can be used to make an *FLQuant* with uncertainty (see Figure 12). We

use the m4 object from earlier with uncertainty on the `level` and `trend` parameters.

```
# Simple - no time trend but with iterations
m(m4, nao = 1)

## An object of class "FLQuant"
## iters:  100
##
## , , unit = unique, season = all, area = unique
##
##    year
## age 0
##   0 2.9935(0.467)
##
## units:  NA

dim(m(m4, nao = 1))

##    age   year   unit season   area   iter
##      1      1      1      1      1    100

# With ages
rngquant(m4) <- c(0, 7)
m(m4, nao = 0)

## An object of class "FLQuant"
## iters:  100
##
## , , unit = unique, season = all, area = unique
##
##    year
## age 0
##   0 2.0103563(0.232595)
##   1 0.7395688(0.085567)
##   2 0.2720721(0.031478)
##   3 0.1000897(0.011580)
##   4 0.0368210(0.004260)
##   5 0.0135457(0.001567)
##   6 0.0049832(0.000577)
##   7 0.0018332(0.000212)
##
## units:  NA

dim(m(m4, nao = 0))

##    age   year   unit season   area   iter
##      8      1      1      1      1    100

# With ages and years
rngyear(m4) <- c(2000, 2003)
m(m4, nao = as.numeric(nao[, as.character(2000:2003)]))

## An object of class "FLQuant"
## iters:  100
##
## , , unit = unique, season = all, area = unique
```

```
##
##    year
## age 2000                2001                2002
##   0 2.9934683(0.467106) 2.0103563(0.232595) 2.9934683(0.467106)
##   1 1.1012354(0.171839) 0.7395688(0.085567) 1.1012354(0.171839)
##   2 0.4051219(0.063216) 0.2720721(0.031478) 0.4051219(0.063216)
##   3 0.1490360(0.023256) 0.1000897(0.011580) 0.1490360(0.023256)
##   4 0.0548273(0.008555) 0.0368210(0.004260) 0.0548273(0.008555)
##   5 0.0201698(0.003147) 0.0135457(0.001567) 0.0201698(0.003147)
##   6 0.0074201(0.001158) 0.0049832(0.000577) 0.0074201(0.001158)
##   7 0.0027297(0.000426) 0.0018332(0.000212) 0.0027297(0.000426)
##    year
## age 2003
##   0 2.0103563(0.232595)
##   1 0.7395688(0.085567)
##   2 0.2720721(0.031478)
##   3 0.1000897(0.011580)
##   4 0.0368210(0.004260)
##   5 0.0135457(0.001567)
##   6 0.0049832(0.000577)
##   7 0.0018332(0.000212)
##
## units:  NA
```

```
dim(m(m4, nao = as.numeric(nao[, as.character(2000:2003)])))
```

```
##    age   year   unit season   area   iter
##      8      4      1      1      1    100
```

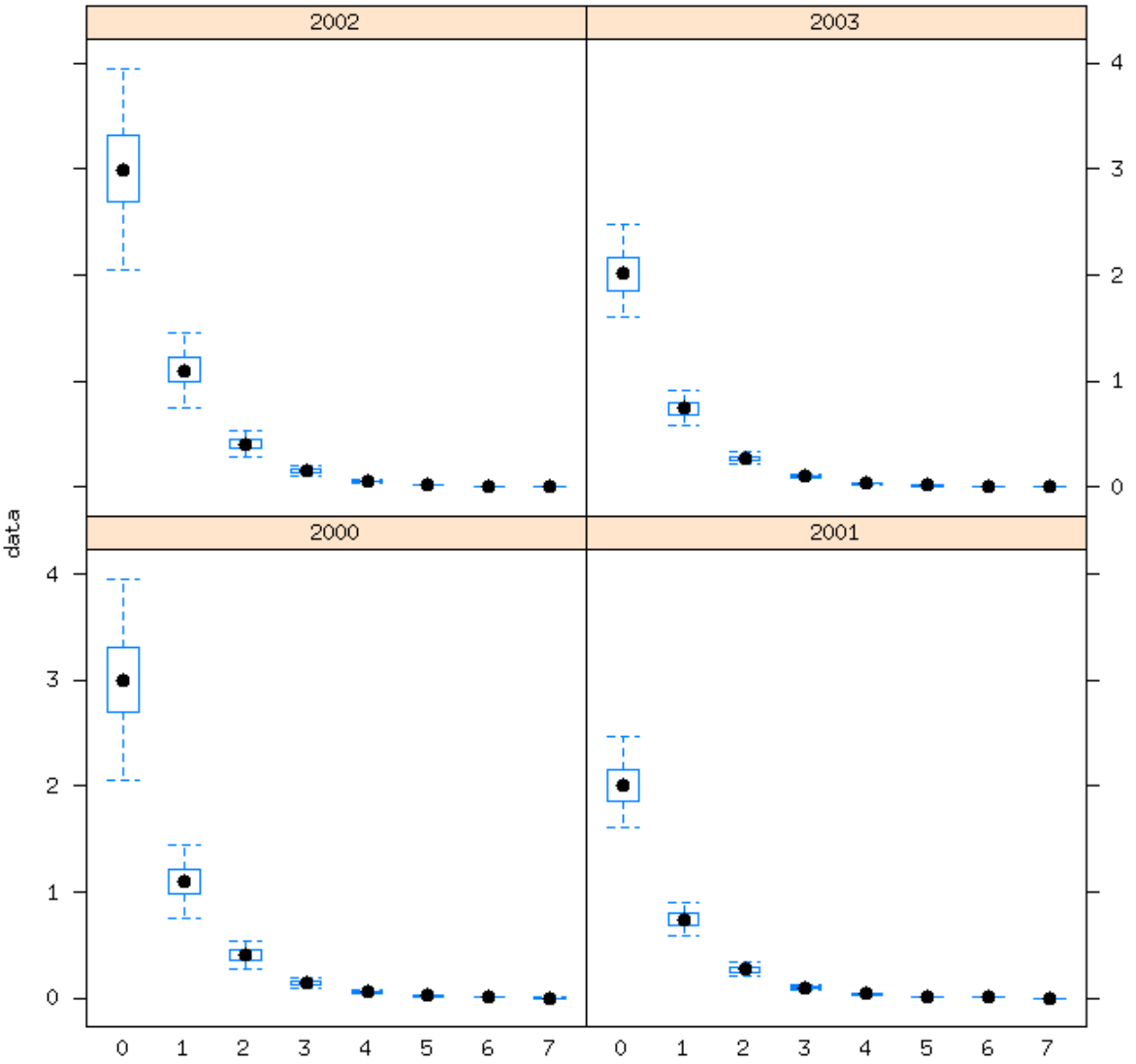

Figure 12: Natural mortality with age and year trend.

5 Running assessments

In the `a4a` assessment model, the model structure is defined by submodels, which are the different parts of a statistical catch at age model that require structural assumptions.

There are 5 submodels in operation: a model for F-at-age, a model for the initial age structure, a model for recruitment, a (list) of model(s) for abundance indices catchability-at-age, and a list of models for the observation variance of catch-at-age and abundance indices. In practice, we fix the variance models and the initial age structure models, but in theory these can be changed.

The submodels form use linear models. This opens the possibility of using the linear modelling tools available in R: see for example gam formulas, or factorial design formulas using `lm()`. In R's linear modelling language, a constant model is coded as ~ 1, while a slope over age would simply be $\sim age$. For example, we can write a traditional year/age separable F model like $\sim factor(age) + factor(year)$.

There are two basic types of assessments available in `a4a`: the management procedure fit and the full assessment fit. The management procedure fit does not compute estimates of covariances and is therefore quicker to execute, while the full assessment fit returns parameter estimates and their covariances at the expense of longer fitting time.

5.1 Stock assessment model details

The statistical catch at age model is based on the well known Baranov catch equation:

$$e^{\mathrm{E}[\log C]} = \frac{\mathbf{F}}{\mathbf{F}+M}\left(1-e^{-\mathbf{F}-M}\right)\mathbf{R}e^{-\sum \mathbf{F}+M}$$

and the survival equation

$$e^{\mathrm{E}[\log I]} = \mathbf{Q}\mathbf{R}e^{-\sum \mathbf{F}+M}$$

where

$$\mathrm{Var}\left[\log C_{ay}\right] = \sigma^{\mathbf{2}}_{\mathbf{ay}} \qquad \mathrm{Var}\left[\log I_{ays}\right] = \tau^{\mathbf{2}}_{\mathbf{ays}}$$

The quantities $\log F$, $\log Q$, $\log R$, $\log$ *observation variances* and $\log$ *initial age structure* (in red in the equations above), need to be given a form, which is done using linear models. Recruitment is a special case. It is modelled as a fixed variance random effect, using the hard coded models Ricker, Beverton Holt, smooth hockeystick or geometric mean, which can use linear models for their parameters $\log a$ or $\log b$, where relevant. As an alternative the $\log R$ submodel can use a linear model like the other submodels

The 'language' of linear models has been developing within the statistical community for many years, and constitutes an elegant way of defining models without going through the complexity of mathematical representations. This approach makes it also easier to communicate among scientists

- 1965 J. A. Nelder, notation for randomized block design
- 1973 Wilkinson and Rodgers, symbolic description for factorial designs
- 1990 Hastie and Tibshirani, introduced notation for smoothers
- 1991 Chambers and Hastie, further developed for use in S

5.2 Quick and dirty

Here we show a simple example of using the assessment model using plaice in the North Sea. The default settings of the stock assessment model work reasonably well. It's an area of research that will improve with time. Note that because the survey index for plaice has missing values we get a warning saying that we assume these values are missing at random, and not because the observations were zero.

```
data(ple4)
data(ple4.indices)
fit <- sca(ple4, ple4.indices)

## Note:  The following observations are treated as being missing at random:
##       fleet year age
##     BTS-Isis 1997  1
##     BTS-Isis 1997  2
## BTS-Tridens 1997  1
## BTS-Tridens 1997  2
##         SNS 1997  1
##         SNS 1997  2
##         SNS 2003  1
##         SNS 2003  2
##         SNS 2003  3
##      Predictions will be made for missing observations.
```

The `residuals()` method will compute standardized residuals which can be plotted using a set of packed methods.

```
res <- residuals(fit, ple4, ple4.indices)
```

Figure 13 shows a scatterplot of residuals by age and survey, with a smoother to guide (or mis-guide ...) your visual analysis.

```
plot(res, main = "Residuals")
```

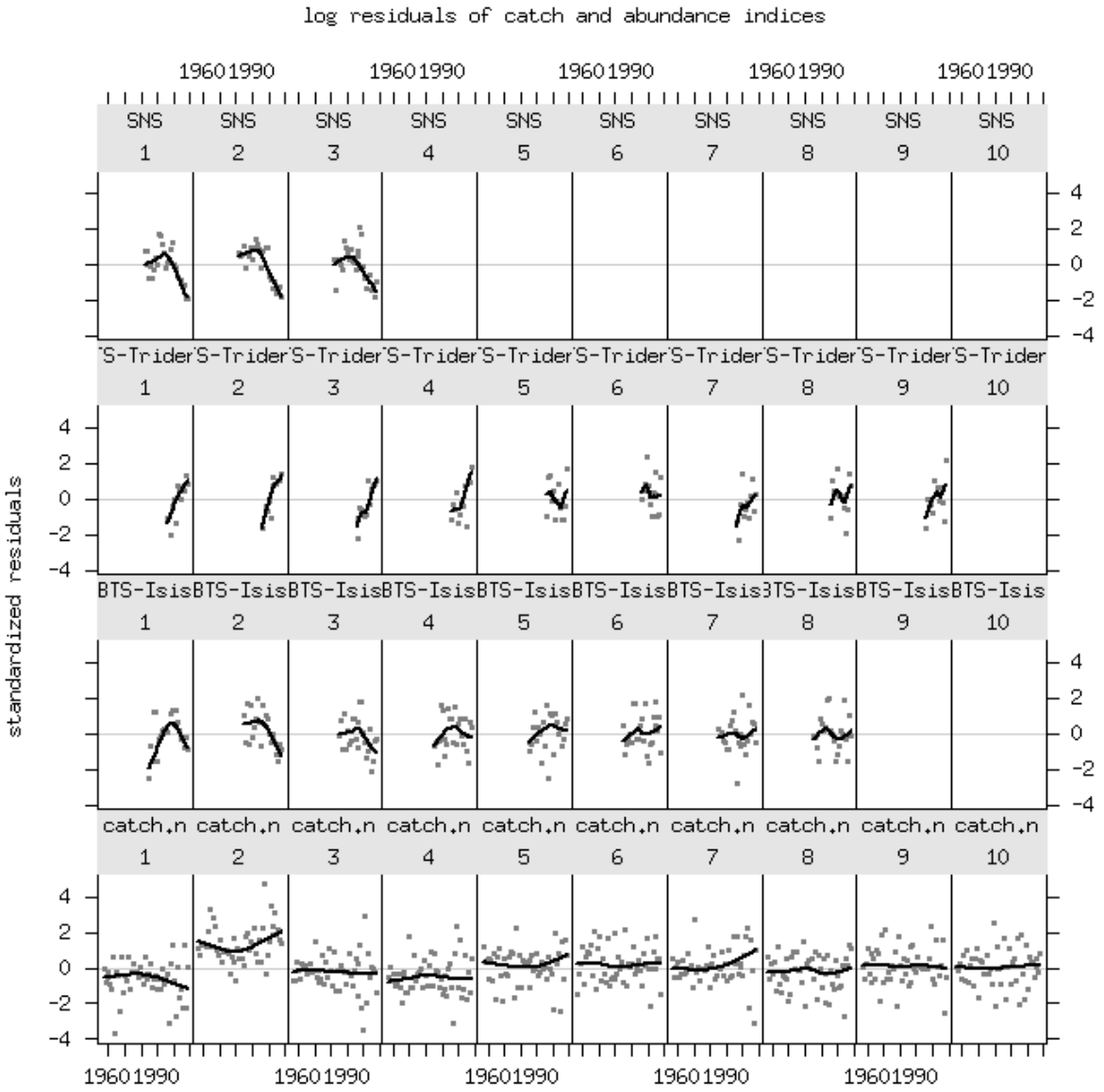

Figure 13: Standardized residuals

The common bubble plot by year and age for each survey are shown in Figure 14.

```
bubbles(res)
```

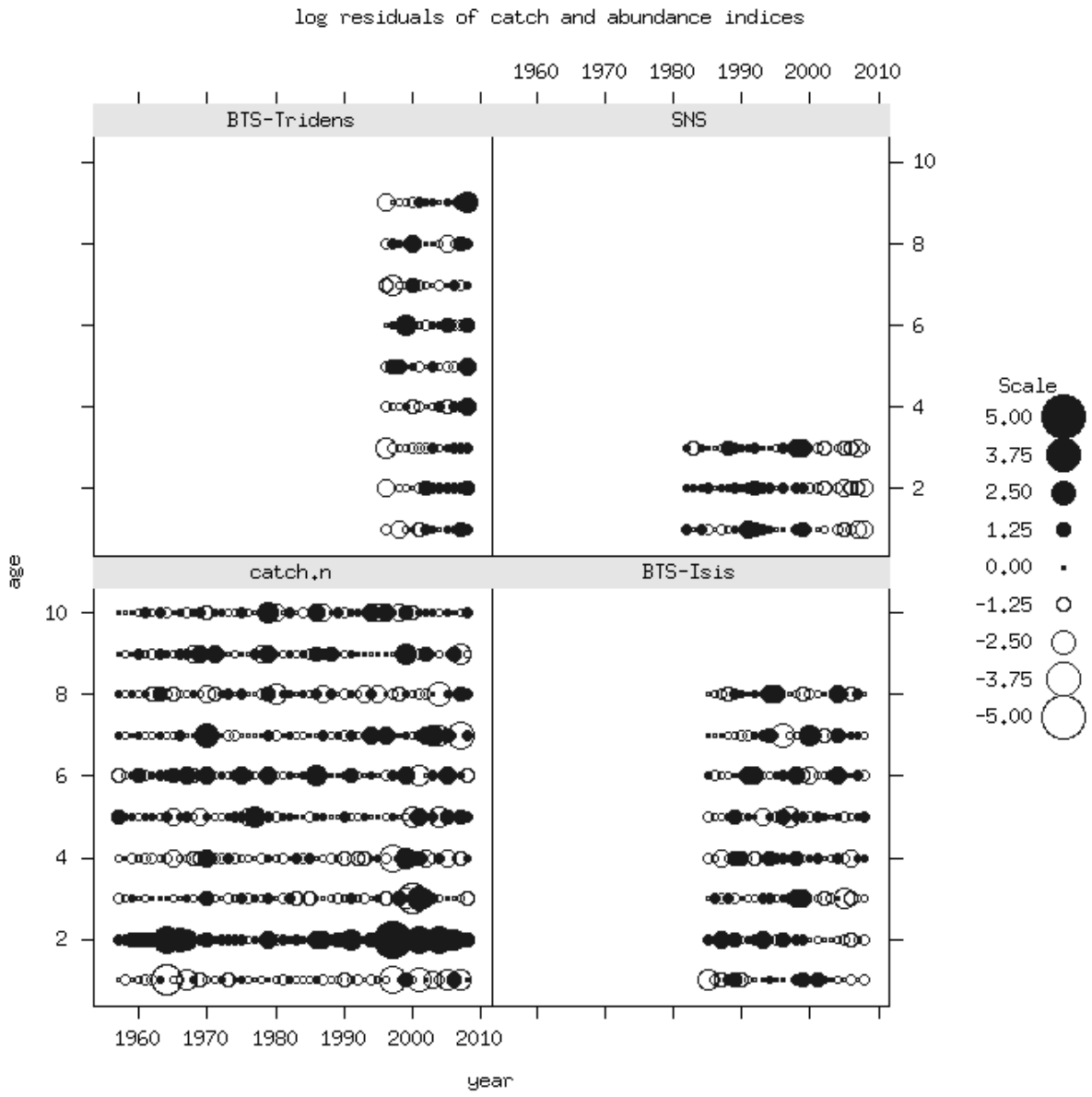

Figure 14: Bubbles plot of standardized residuals.

Finally, Figure 15 shows a quantile-quantile plot to assess how well do the residuals match the normal distribution.

```
qqmath(res)
```

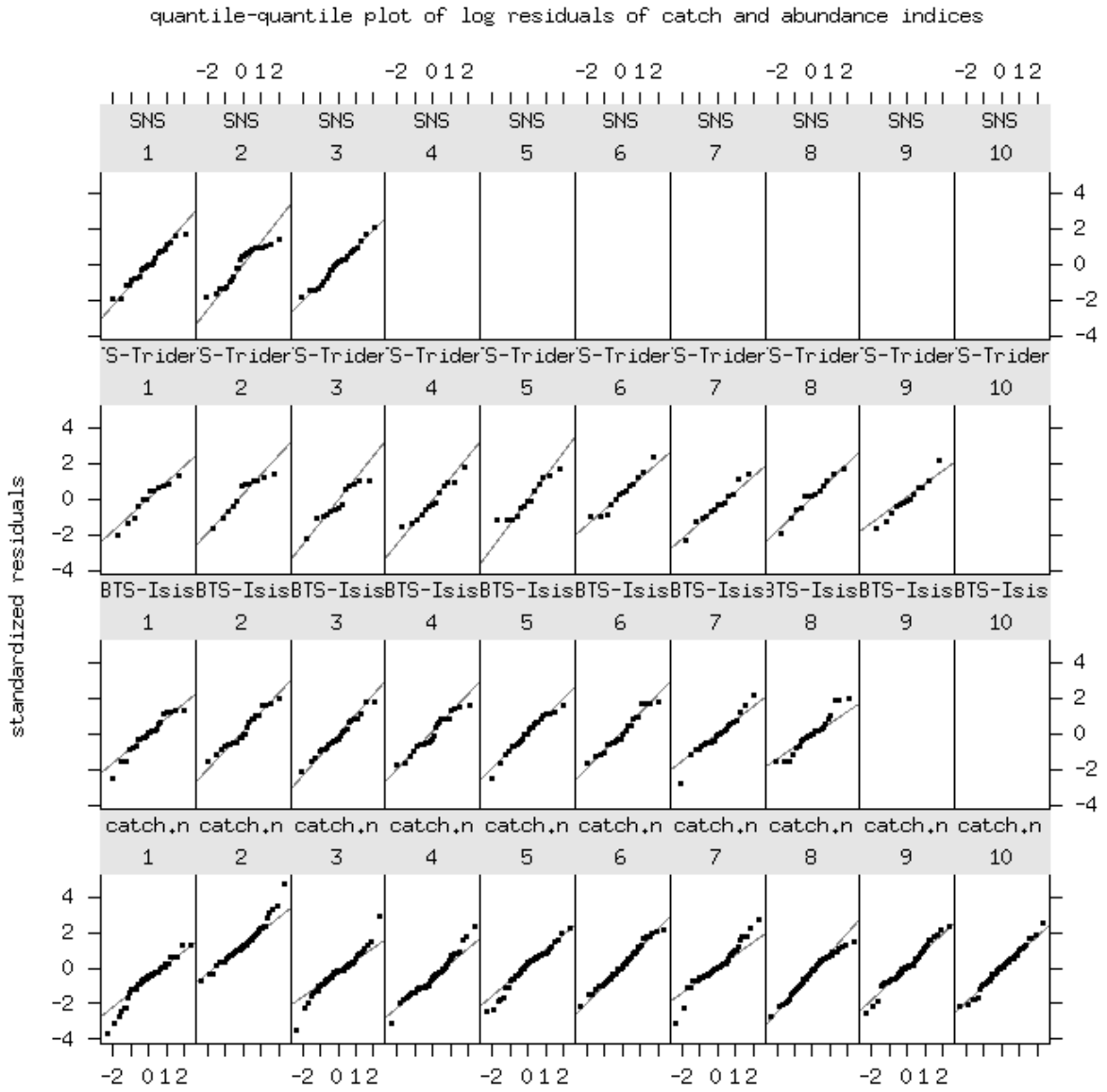

Figure 15: Quantile-quantile plot of standardized residuals.

To inspect the summaries (Fbar, SSB, catch and recruitment) of the fit the user may add the fit to the original stock object, using the method + and plot the result (Figure 16).

```
stk <- ple4 + fit
plot(stk)
```

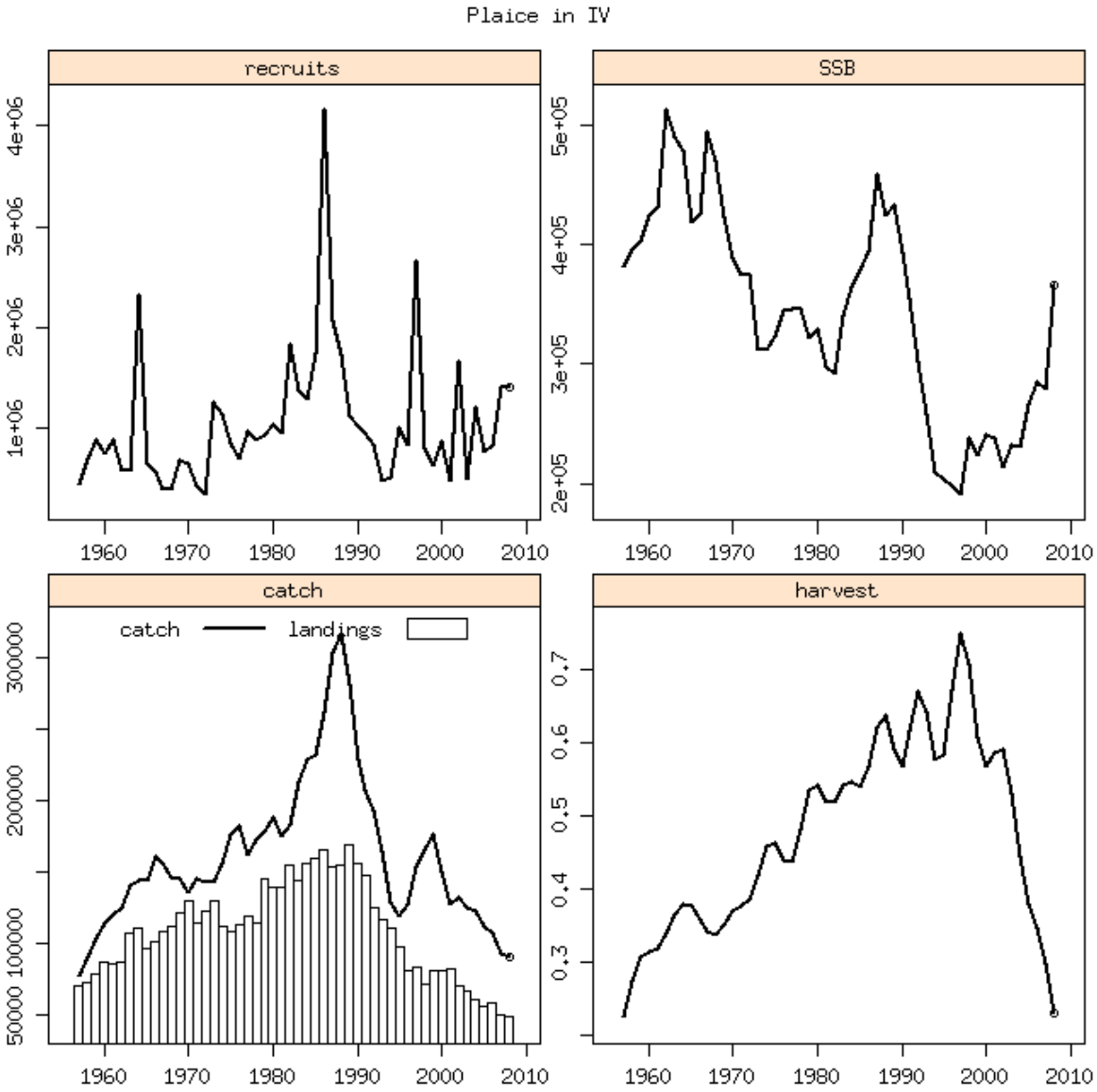

Figure 16: Stock summary

In more detail, one can plot a 3D representation of fishing mortality (Figure 17),

```
wireframe(data ~ age + year, data = as.data.frame(harvest(stk)), drape = TRUE,
    screen = list(x = -90, y = -45))
```

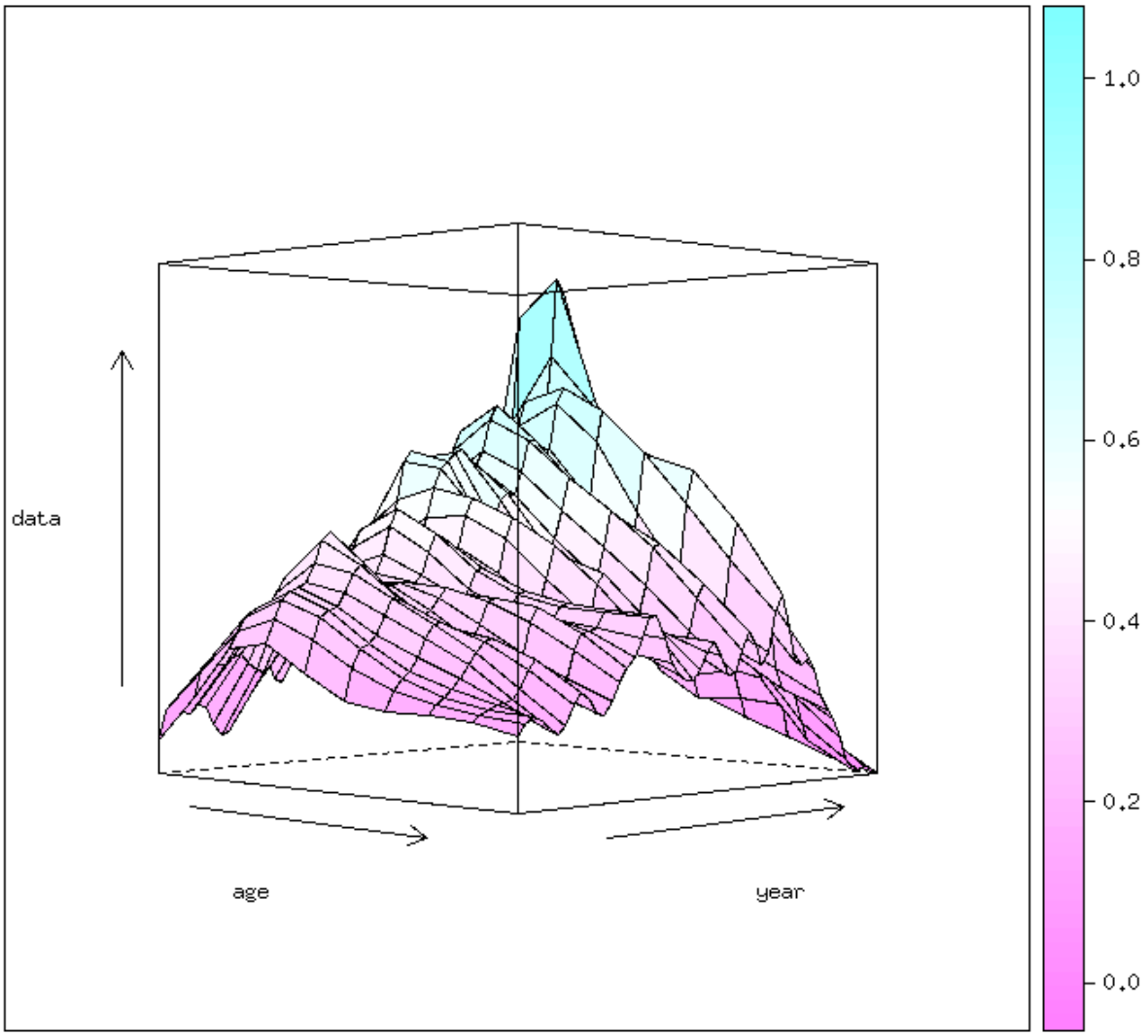

Figure 17: Fishing mortality

population abundance (Figure 18),

```
wireframe(data ~ age + year, data = as.data.frame(stock.n(stk)), drape = TRUE,
    screen = list(x = -90, y = -45))
```

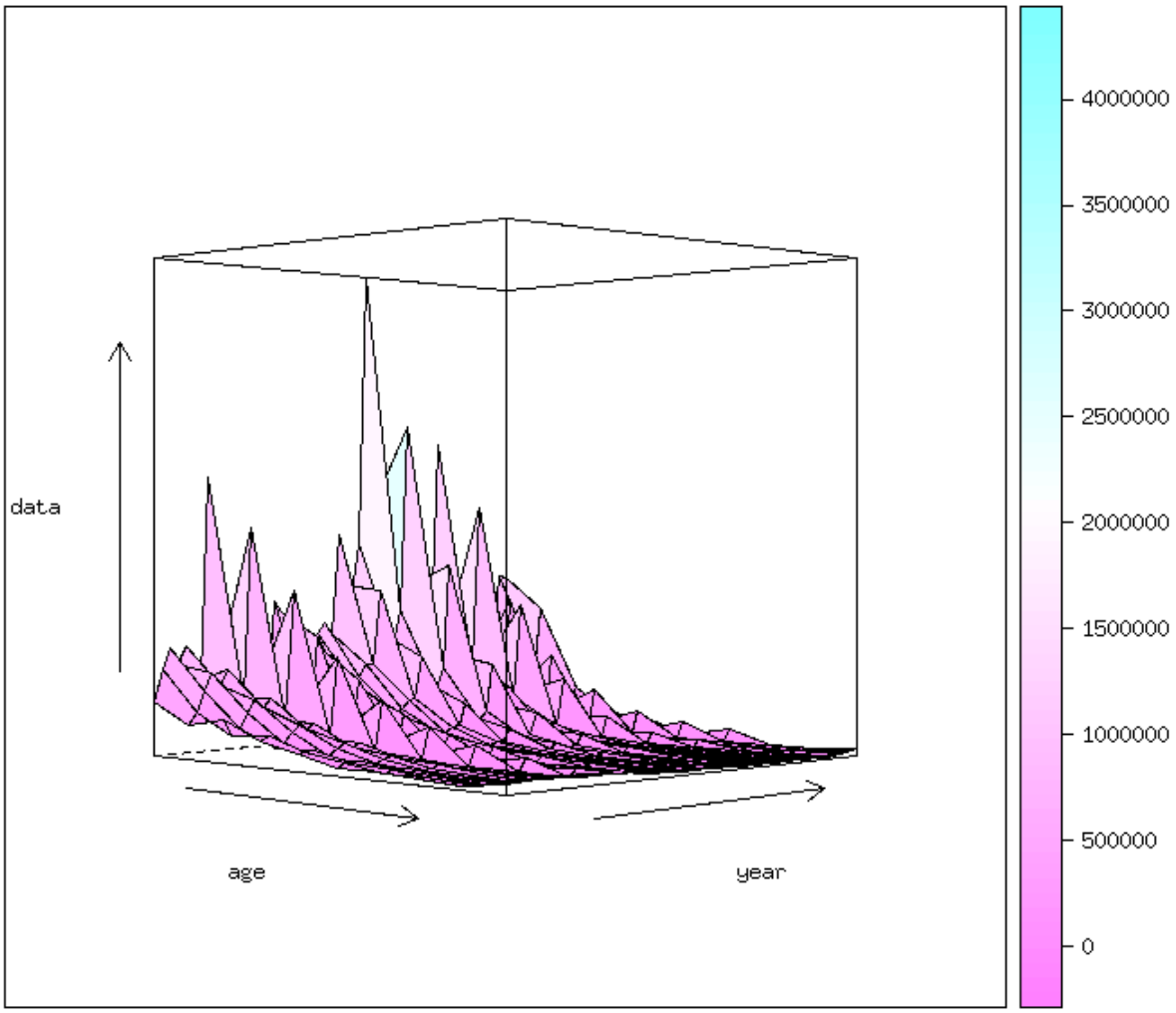

Figure 18: Population abundance

or catch-at-age (Figure 19).

```
wireframe(data ~ age + year, data = as.data.frame(catch.n(stk)), drape = TRUE)
```

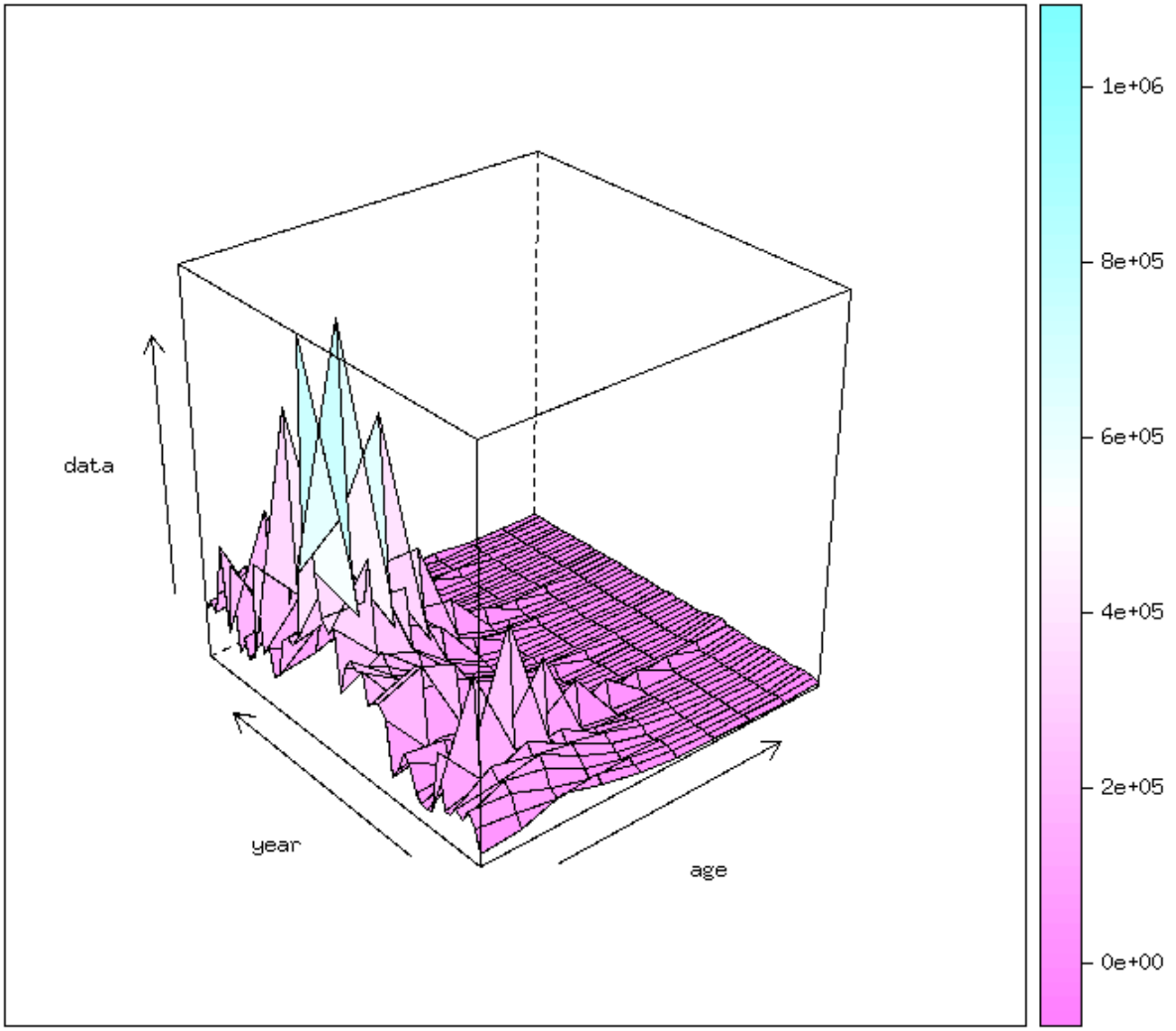

Figure 19: Catches

5.3 Data structures

As mentioned above, the output of the stock assessment method may be simpler, with or without all the information about the parameters of the model like the variance-covariance matrix. In the first case the class of the output object is *a4aFitSA* while in the second case is *a4aFit*.

This section will describe the data structures of these classes and the classes that compose them.

Starting with the basic model output class, *a4aFit*, the slots of this class are shown on the code below and Figure 20.

```
showClass("a4aFit")

## Class "a4aFit" [package "FLa4a"]
##
## Slots:
```

```
##
## Name:       call     clock   fitSumm   stock.n   harvest   catch.n
## Class:      call   numeric     array   FLQuant   FLQuant   FLQuant
##
## Name:      index      name      desc     range
## Class:  FLQuants character character   numeric
##
## Extends: "FLComp"
##
## Known Subclasses: "a4aFitSA"
```

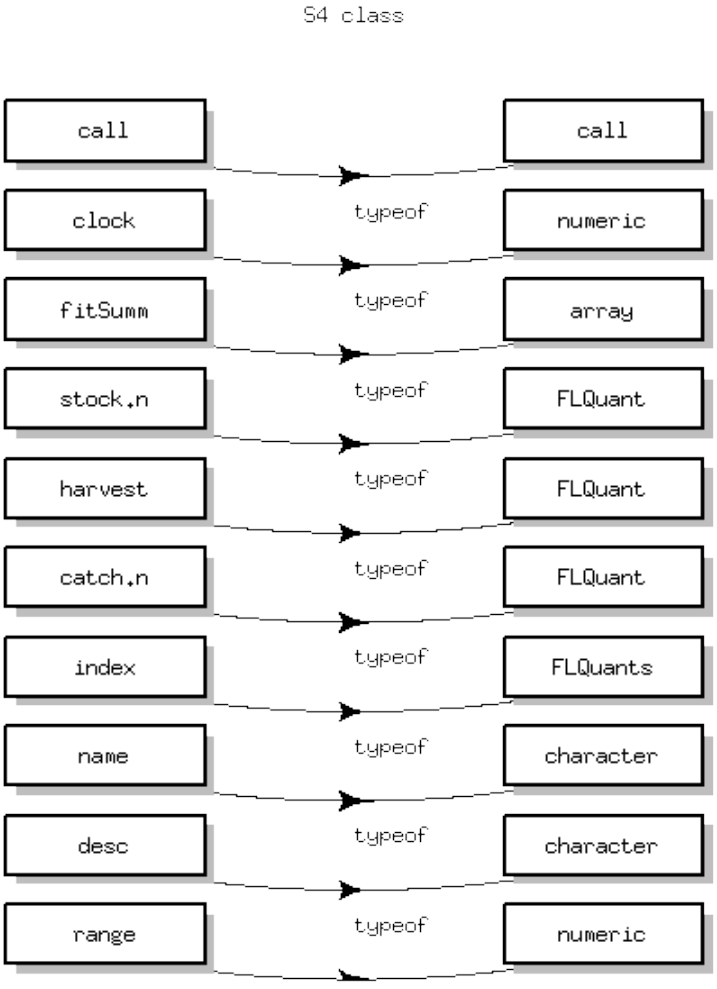

Figure 20: The a4aFit class

Fitted values are stored in the `stock.n`, `harvest`, `catch.n` and `index` slots. It also contains information carried over from the stock object used to fit the model, like the the name of the stock in `name`, any description provided in `desc` and the age and year range and mean F range in `range`. There is also a wall clock that has a breakdown of the time taken to run the model in `clock`.

The full assessment fit returns an object of class *a4aFitSA*, the slots of this class are shown on the code

below and Figure 21.

```
showClass("a4aFitSA")

## Class "a4aFitSA" [package "FLa4a"]
##
## Slots:
##
## Name:       pars      call     clock   fitSumm   stock.n   harvest
## Class:   SCAPars      call   numeric     array   FLQuant   FLQuant
##
## Name:    catch.n     index      name      desc     range
## Class:   FLQuant  FLQuants character character   numeric
##
## Extends:
## Class "a4aFit", directly
## Class "FLComp", by class "a4aFit", distance 2
```

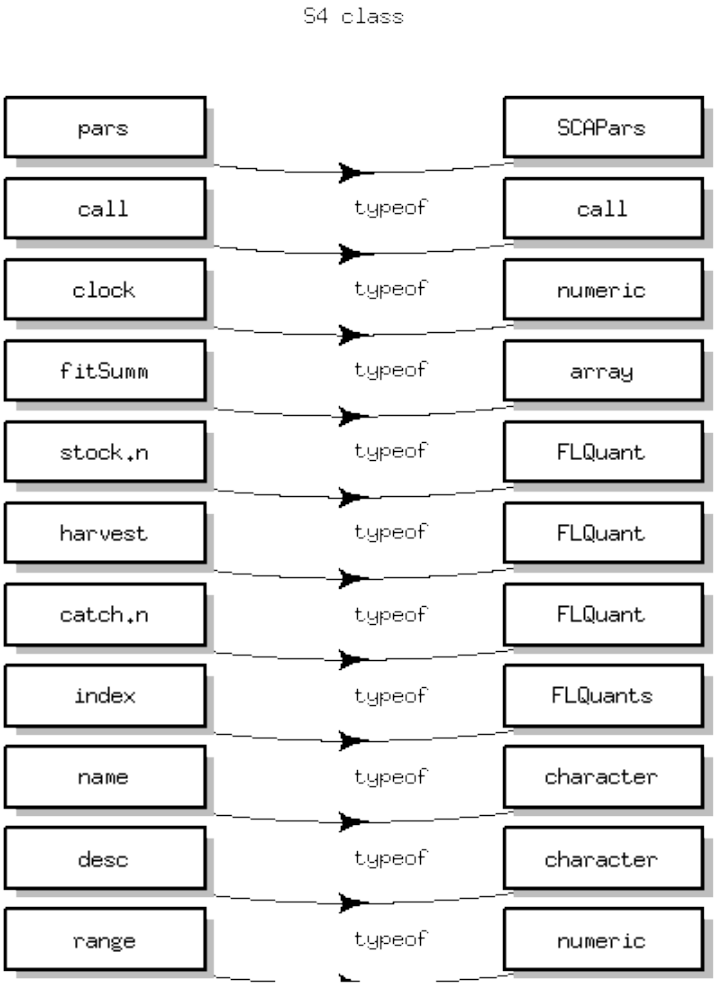

Figure 21: The a4aFitSA class

The additional slots in the assessment output are the `fitSumm` and `pars` slots, which are containers for model summaries and the model parameters. The `pars` slot is a class of type *SCAPars* (Figure 22) which is itself composed of sub-classes, designed to contain the information necessary to simulate from the model.

```
showClass("SCAPars")

## Class "SCAPars" [package "FLa4a"]
##
## Slots:
##
## Name:      stkmodel       qmodel       vmodel
## Class: a4aStkParams    submodels    submodels

showClass("a4aStkParams")

## Class "a4aStkParams" [package "FLa4a"]
##
## Slots:
##
## Name:       fMod     n1Mod     srMod    params      vcov centering
## Class:   formula   formula   formula     FLPar     array   numeric
##
## Name:      distr         m     units      name      desc     range
## Class: character   FLQuant character character character   numeric
##
## Extends: "FLComp"

showClass("submodel")

## Class "submodel" [package "FLa4a"]
##
## Slots:
##
## Name:        Mod    params      vcov centering     distr      name
## Class:   formula     FLPar     array   numeric character character
##
## Name:       desc     range
## Class: character   numeric
##
## Extends: "FLComp"
```

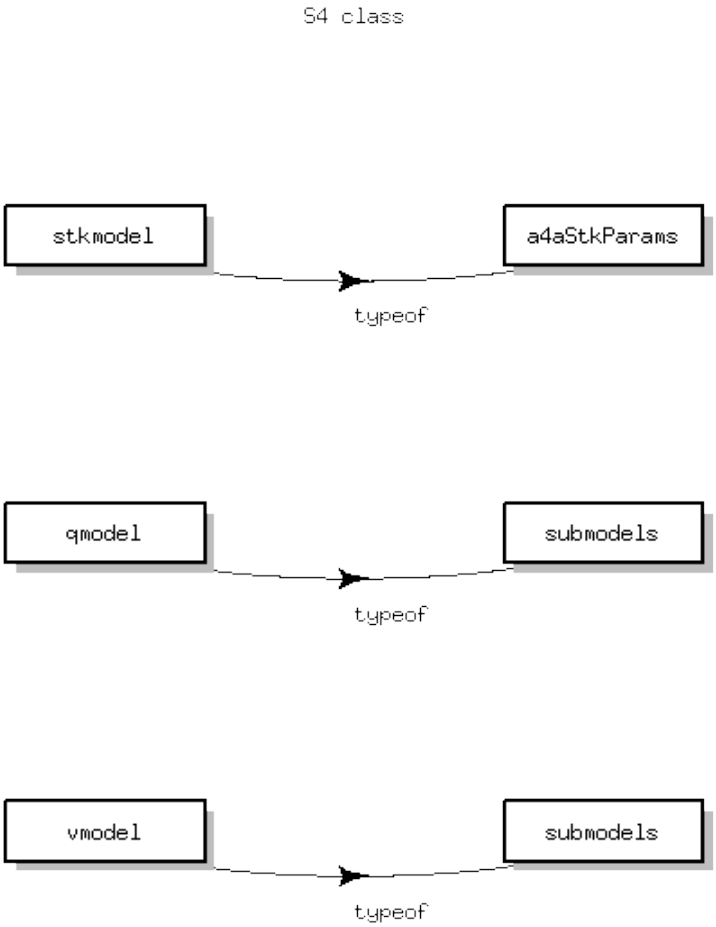

Figure 22: The SCAPars class

The *SCAPars* is built using objects of class *a4aStkParams* (Figure 23) and *submodel* (Figure 24). These classes have the following slots.

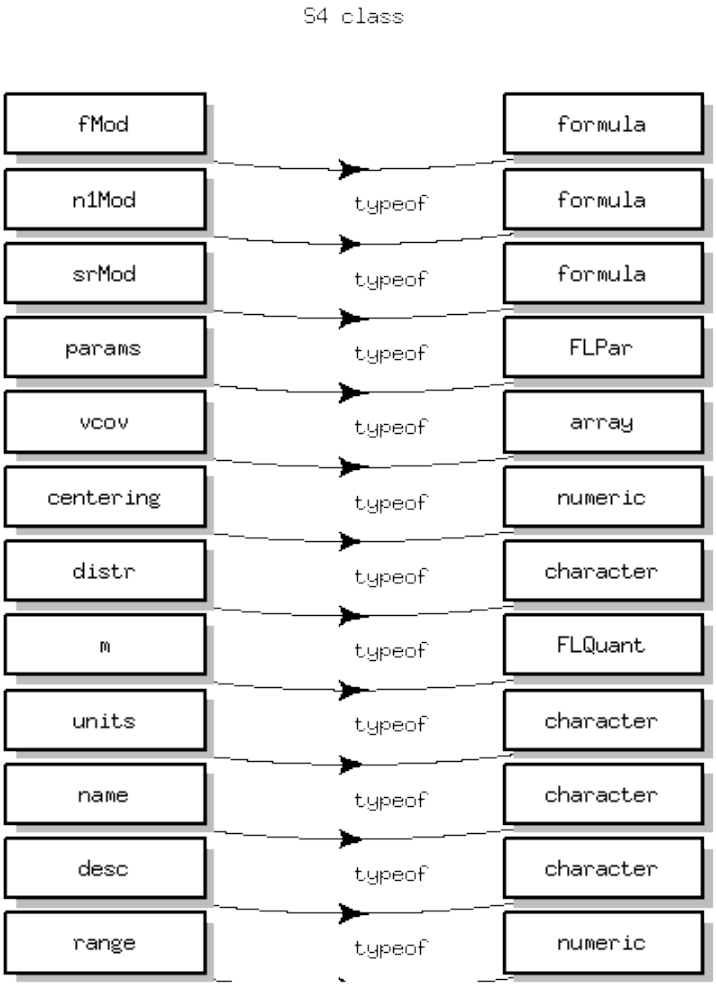

Figure 23: The a4aStkParams class

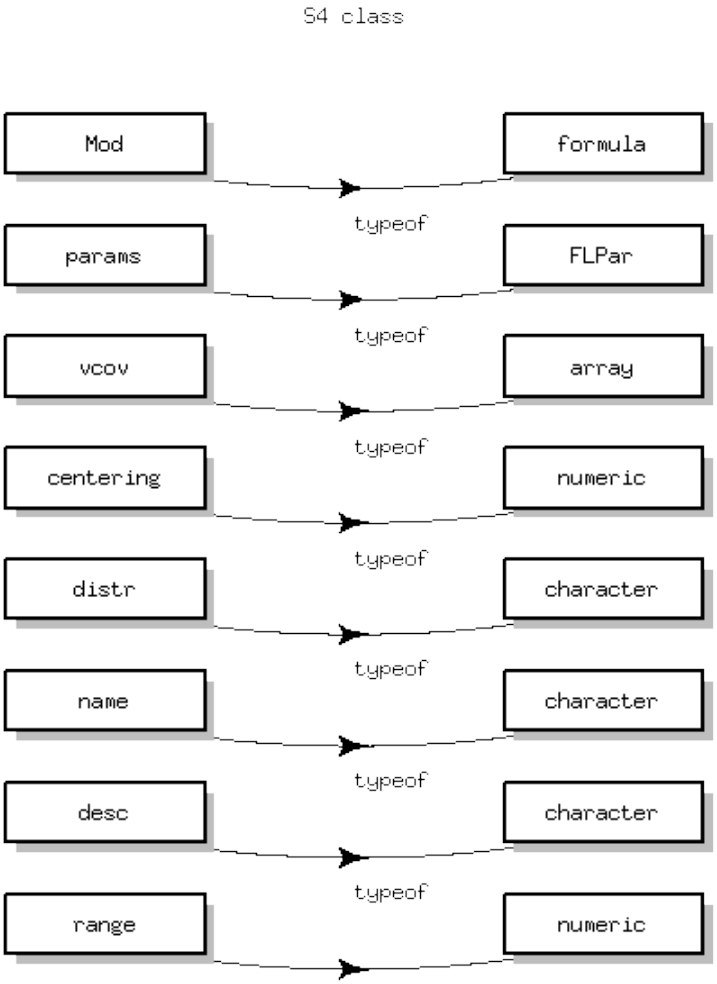

Figure 24: The submodel class

For example, all the parameters required so simulate a time-series of mean F trends is contained in the `stkmodel` slot, which is a class of type *a4aStkParams*. This class contains the relevant submodels (see later), their parameters `params` and the joint covariance matrix `vcov` for all stock related parameters.

5.4 The sca method - statistical catch-at-age

The `sca()` method used in the previous section with the default settings, can be parametrized to control other features of the stock assessment framework. The most interesting ones are the submodels for F, Q and R.

An important argument for `sca()` is the type of fit, which controls if a full assessment will be performed or a management procedure type of assessment. The argument is called `fit` and can have the values 'assessment' for a full assesemt or 'MP' for a simpler assessment. By default `sca()` uses `fit='MP'`.

We'll start by looking at the submodel for F, then Q and finally R.

Please note that each of these model *forms* have not been tuned to the data. The degrees of freedom of each model can be better tuned to the data by using model selection procedures such as AIC or BIC, etc.

5.4.1 Fishing mortality submodel

We will now take a look at some examples for F models and the forms that we can get. We'll fix the Q and R submodels.

Lets start with a separable model in which age and year effects are modelled as dummy variables, using the `factor` coding provided by R (Figure 25).

```
qmod <- list(~factor(age))
fmod <- ~factor(age) + factor(year)
srmod <- ~factor(year)
fit <- sca(stock = ple4, indices = ple4.indices[1], fmodel = fmod, qmodel = qmod,
    srmodel = srmod)
```

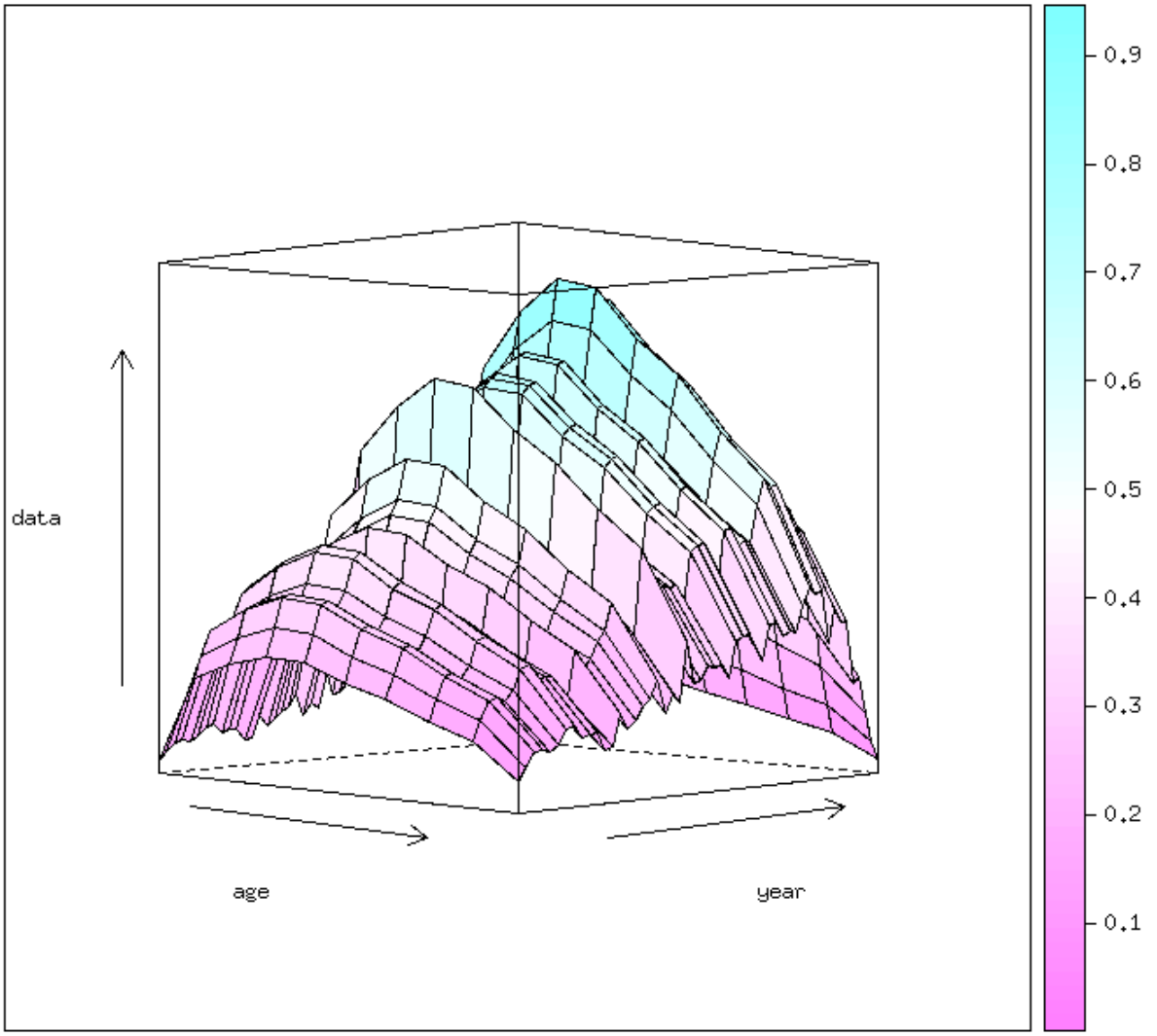

Figure 25: Fishing mortality separable model

Next we may make things a bit more interesting by using an (unpenalised) thin plate spline, where we'll borrow the method `s()` provided by package `mgcv`. We're using the North Sea Plaice data again, and since it has 10 ages we will use a simple rule of thumb that the spline should have fewer than $\frac{10}{2} = 5$

degrees of freedom, and so we opt for 4 degrees of freedom. We will also do the same for year and model the change in F through time as a smoother with 20 degrees of freedom. Note that this is still a separable model, it's a smoothed version of the previous model (Figure 26).

```
fmod <- ~s(age, k = 4) + s(year, k = 20)
fit1 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
```

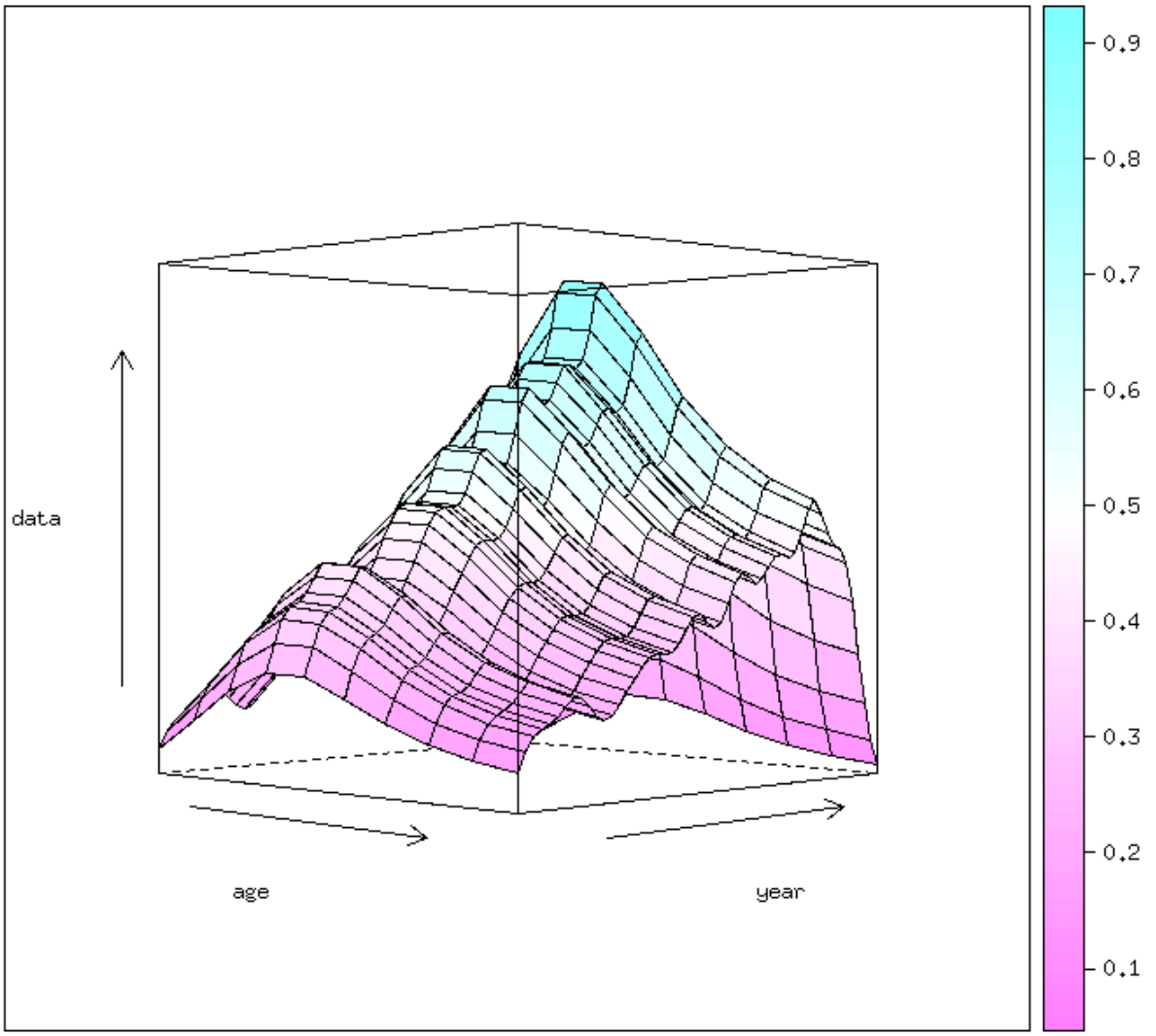

Figure 26: Fishing mortality smoothed separable model

A non-separable model, where we consider age and year to interact can be modeled using a smooth interaction term in the F model using a tensor product of cubic splines with the `te` method (Figure 27), again borrowed from `mgcv`.

```
fmod <- ~te(age, year, k = c(4, 20))
fit3 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
```

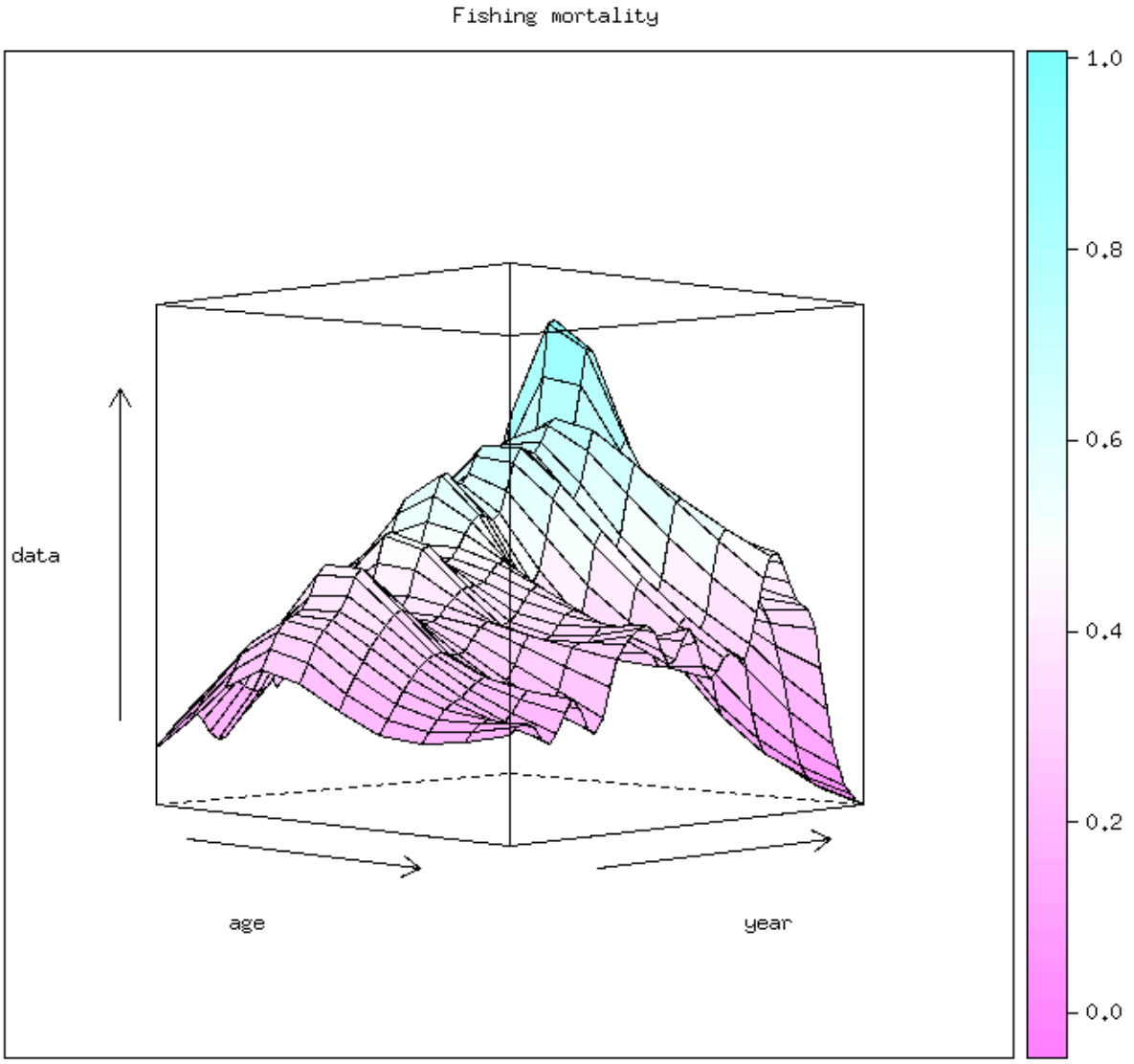

Figure 27: Fishing mortality smoothed non-separable model

In the last examples the Fs are linked across age and time. What if we want to free up a specific age class because in the residuals we see a consistent pattern. This can happen, for example, if the spatial distribution of juveniles is disconnected to the distribution of adults. The fishery focuses on the adult fish, and therefore the the F on young fish is a function of the distribution of the juveniles and could deserve a specific model. This can be achieved by adding a component for the year effect on age 1 (Figure 28).

```
fmod <- ~te(age, year, k = c(4, 20)) + s(year, k = 5, by = as.numeric(age ==
    1))
fit4 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
```

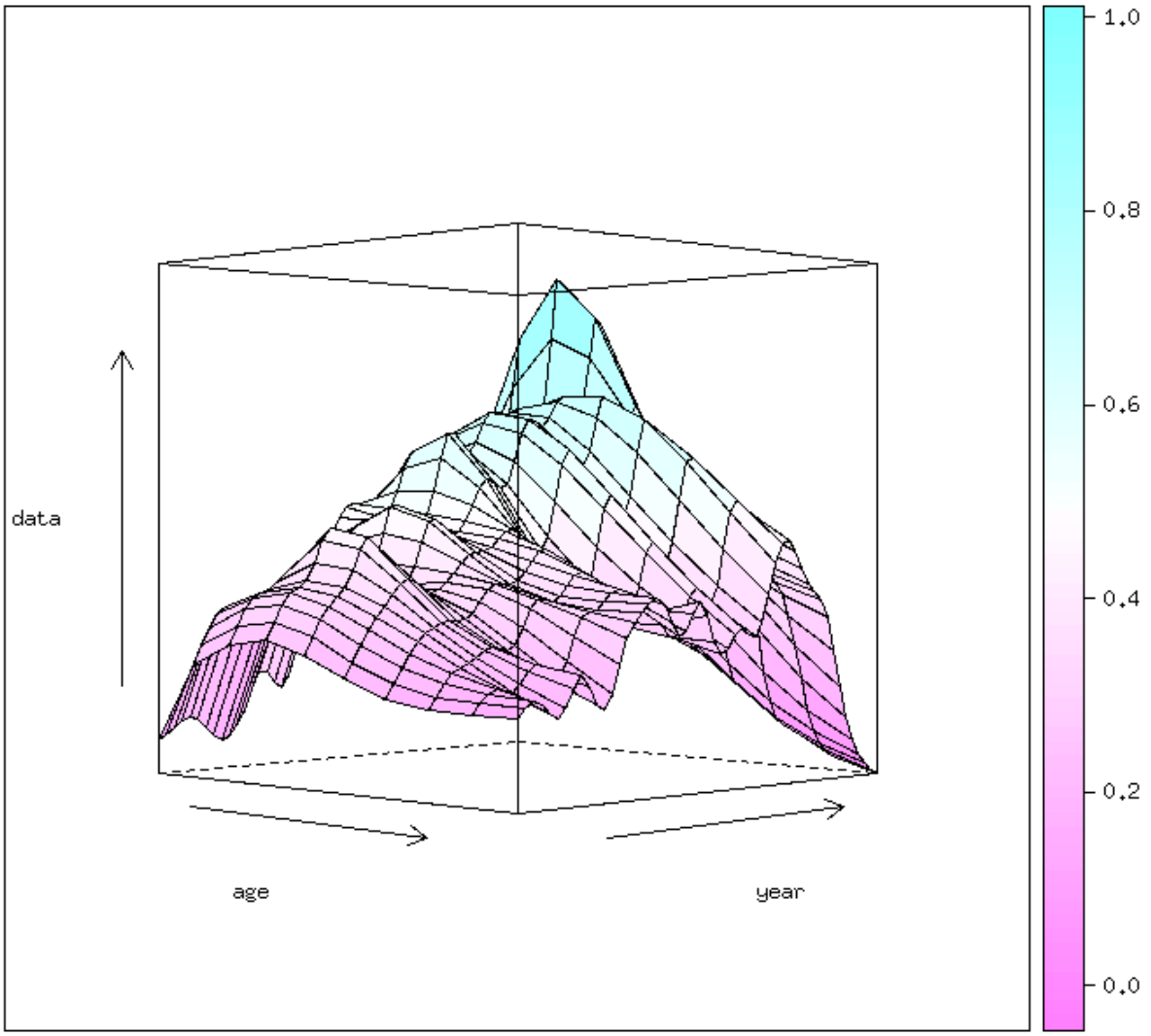

Figure 28: Fishing mortality age-year interaction model with extra age 1 smoother.

5.4.2 Catchability submodel

The catchability submodel is set up the same way as the F submodel and the tools available are the same. The only difference is that the submodel is set up as a list of formulas, where each formula relates with one abundance index.

We'll start by fixing the F and R models and compute the fraction of the year the index relates to, which will allows us to compute catchability at age and year.

```
sfrac <- mean(range(ple4.indices[[1]])[c("startf", "endf")])
fmod <- ~factor(age) + factor(year)
srmod <- ~factor(year)
```

A first model is simply a dummy effect on age, which means that a coefficient will be estimated for each age. Note that this kind of model considers that levels of the factor are independent (Figure 29).

```
qmod <- list(~factor(age))
fit <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)

# compute N for the fraction of the year the survey is carried out
Z <- (m(ple4) + harvest(fit)) * sfrac
lst <- dimnames(fit@index[[1]])
lst$x <- stock.n(fit) * exp(-Z)
stkn <- do.call("trim", lst)
```

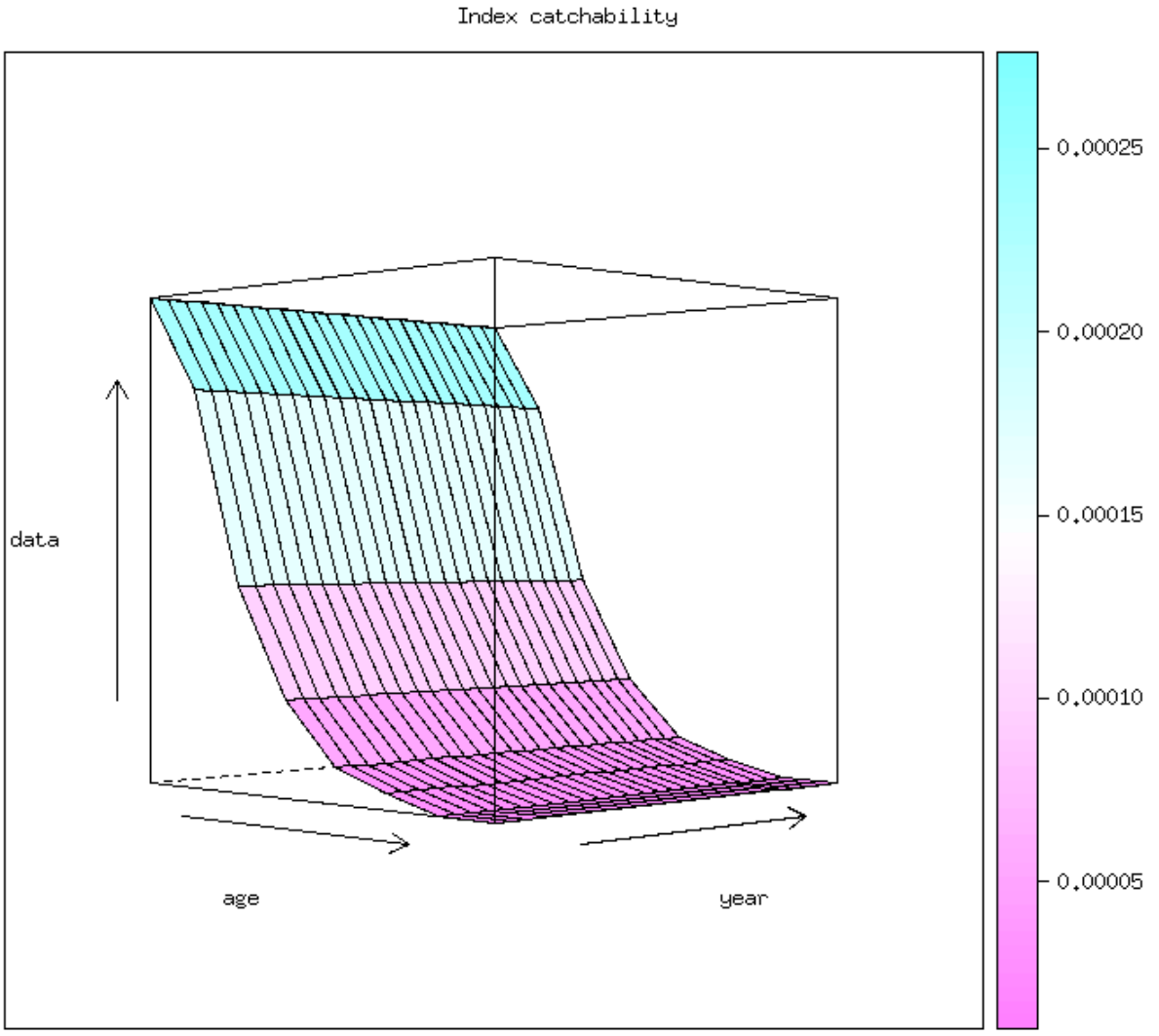

Figure 29: Catchability age independent model

If one considers catchability at a specific age to be dependent on catchability on the other ages, similar to a selectivity modelling approach, one option is to use a smoother at age, and let the data 'speak' regarding the shape (Figure 30).

```
qmod <- list(~s(age, k = 4))
fit1 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
```

```
# compute N for the fraction of the year the survey is carried out
Z <- (m(ple4) + harvest(fit1)) * sfrac
lst <- dimnames(fit1@index[[1]])
lst$x <- stock.n(fit1) * exp(-Z)
stkn <- do.call("trim", lst)
```

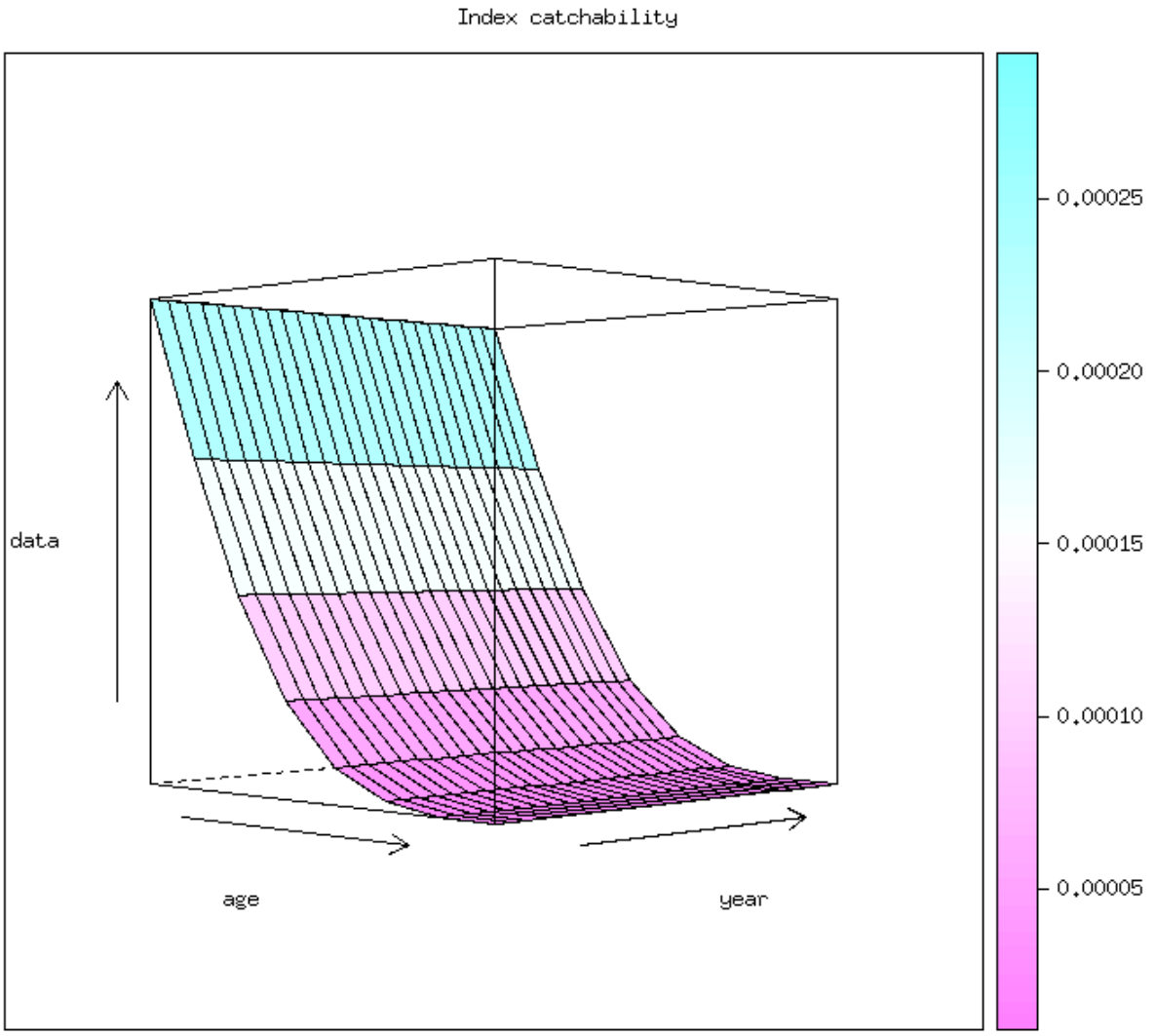

Figure 30: Catchability smoother age model

As in the case of F, one may consider catchability to be a process that evolves with age and year, including an interaction between the two effects. Such model can be modelled using the tensor product of cubic splines, the same way we did for the F model (Figure 31).

```
qmod <- list(~te(age, year, k = c(3, 40)))
fit2 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)

# compute N for the fraction of the year the survey is carried out
Z <- (m(ple4) + harvest(fit2)) * sfrac
lst <- dimnames(fit2@index[[1]])
```

```
lst$x <- stock.n(fit2) * exp(-Z)
stkn <- do.call("trim", lst)
```

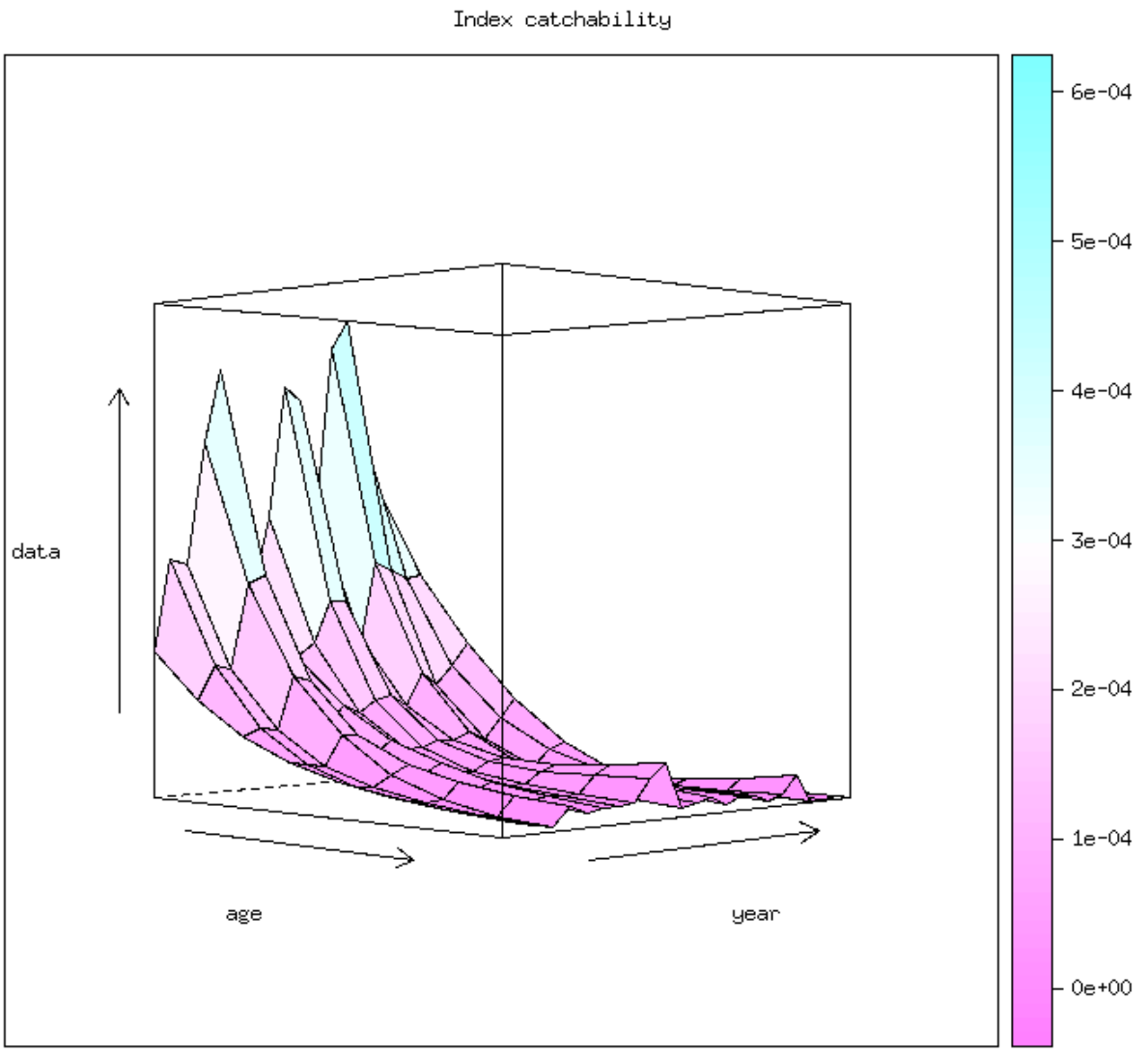

Figure 31: Catchability tensor product of age and year

Finally, one may want to investigate a trend in catchability with time, very common in indices built from CPUE data. In the example given here we'll use a linear trend in time, set up by a simple linear model (Figure 32).

```
qmod <- list(~s(age, k = 4) + year)
fit3 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)

# compute N for the fraction of the year the survey is carried out
Z <- (m(ple4) + harvest(fit3)) * sfrac
lst <- dimnames(fit3@index[[1]])
lst$x <- stock.n(fit3) * exp(-Z)
stkn <- do.call("trim", lst)
```

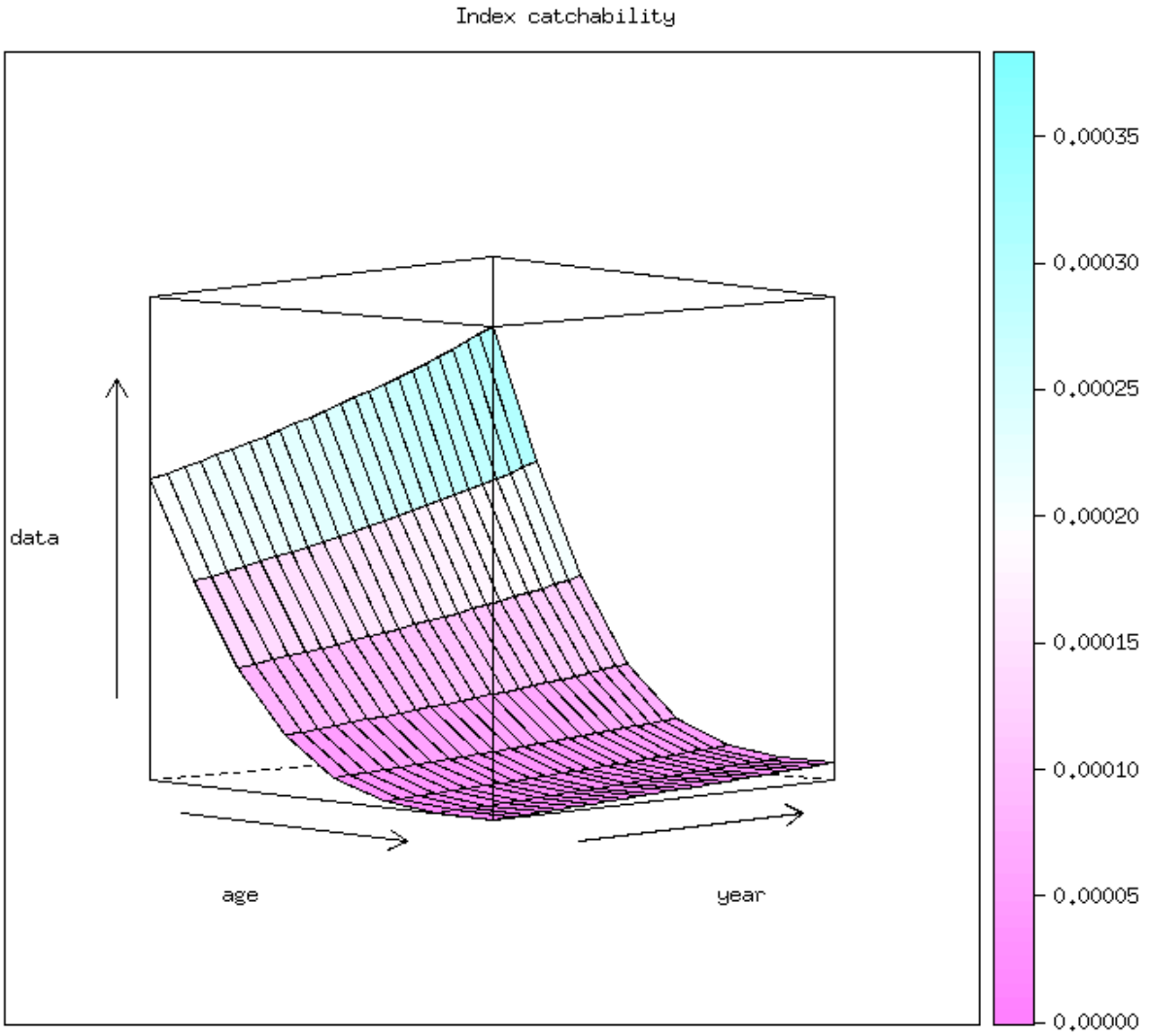

Figure 32: Catchability with a linear trend in year

5.4.3 Catchability submodel for age aggregated indices

The previous section was focused on age disaggregated indices, but age aggregated indices (biomass, DEPM, etc) may also be used to tune the total biomass of the population. In these cases the *FLIndex* has to be named "bio" in the *FLIndices* object that is passed to the fitting method. At the moment only one "bio" index is allowed. Note that in this case the qmodel should be set without age factors. It could have a "year" component though.

```
# creating an index
bioidx <- FLIndex(FLQuant(NA, dimnames = list(age = "all", year = range(ple4)["minyear"]:range(ple4)["m
index(bioidx) <- stock(ple4) * 0.001
index(bioidx) <- index(bioidx) * exp(rnorm(index(bioidx), sd = 0.1))
range(bioidx)[c("startf", "endf")] <- c(0, 0)

# fitting the model
fit <- sca(ple4, FLIndices(bio = bioidx), qmodel = list(~1))
```

```
# how is it fiting catchability ? not too well ...
index(fit)[[1]]/(quantSums(stock.n(fit) * stock.wt(ple4)))
```

```
## An object of class "FLQuant"
## , , unit = unique, season = all, area = unique
##
##      year
## age   1957       1958       1959       1960       1961       1962
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   1963       1964       1965       1966       1967       1968
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   1969       1970       1971       1972       1973       1974
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   1975       1976       1977       1978       1979       1980
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   1981       1982       1983       1984       1985       1986
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   1987       1988       1989       1990       1991       1992
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   1993       1994       1995       1996       1997       1998
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   1999       2000       2001       2002       2003       2004
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##      year
## age   2005       2006       2007       2008
##   all 5.2678e-05 5.2678e-05 5.2678e-05 5.2678e-05
##
## units:  kg
```

Residuals are not so good but one single biomass index is not very informative (Figure 33). Note that the results are not so bad because most information is coming from the catch at age information (Figure 34).

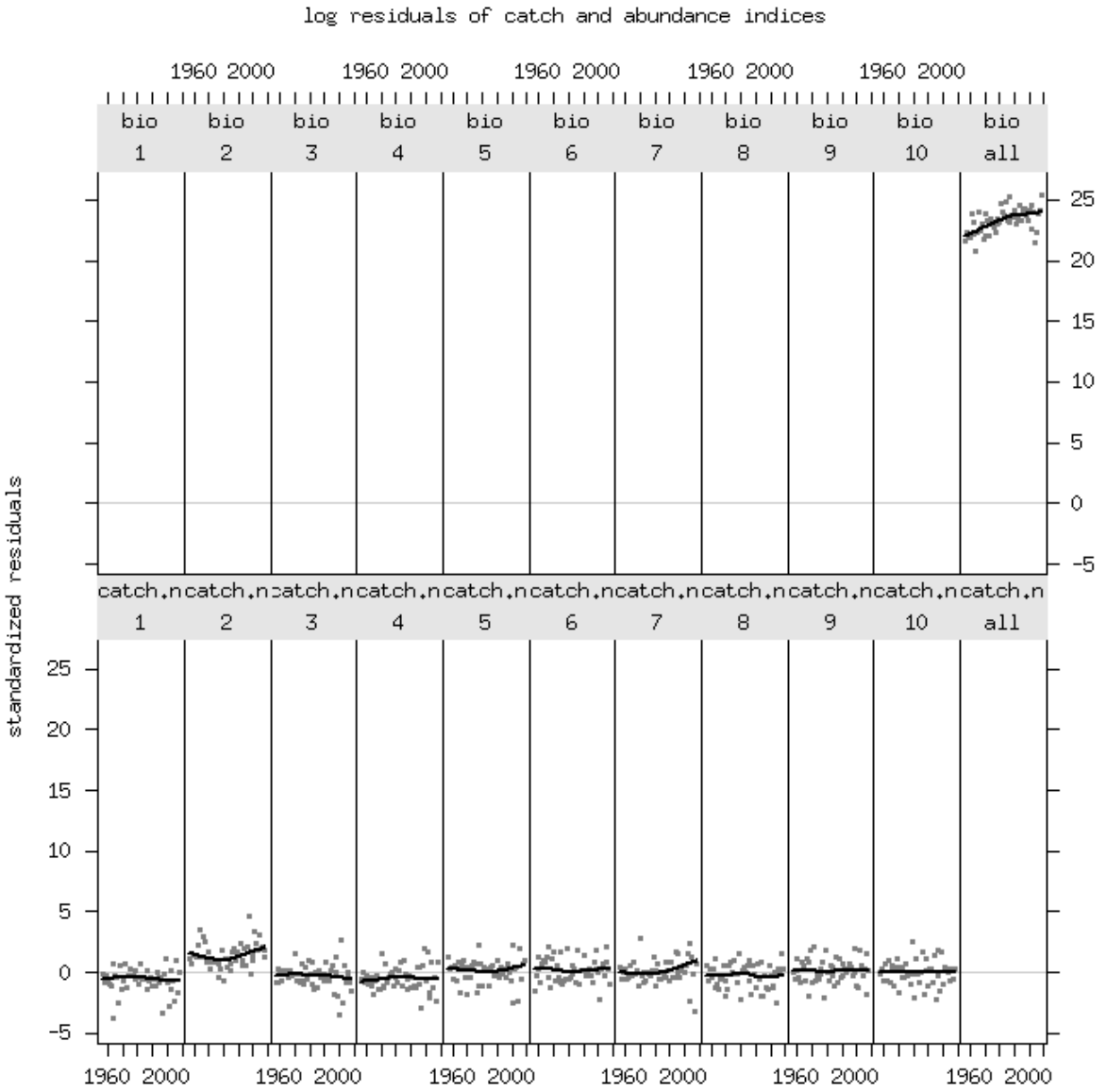

Figure 33: Catchability residuals for a biomass index

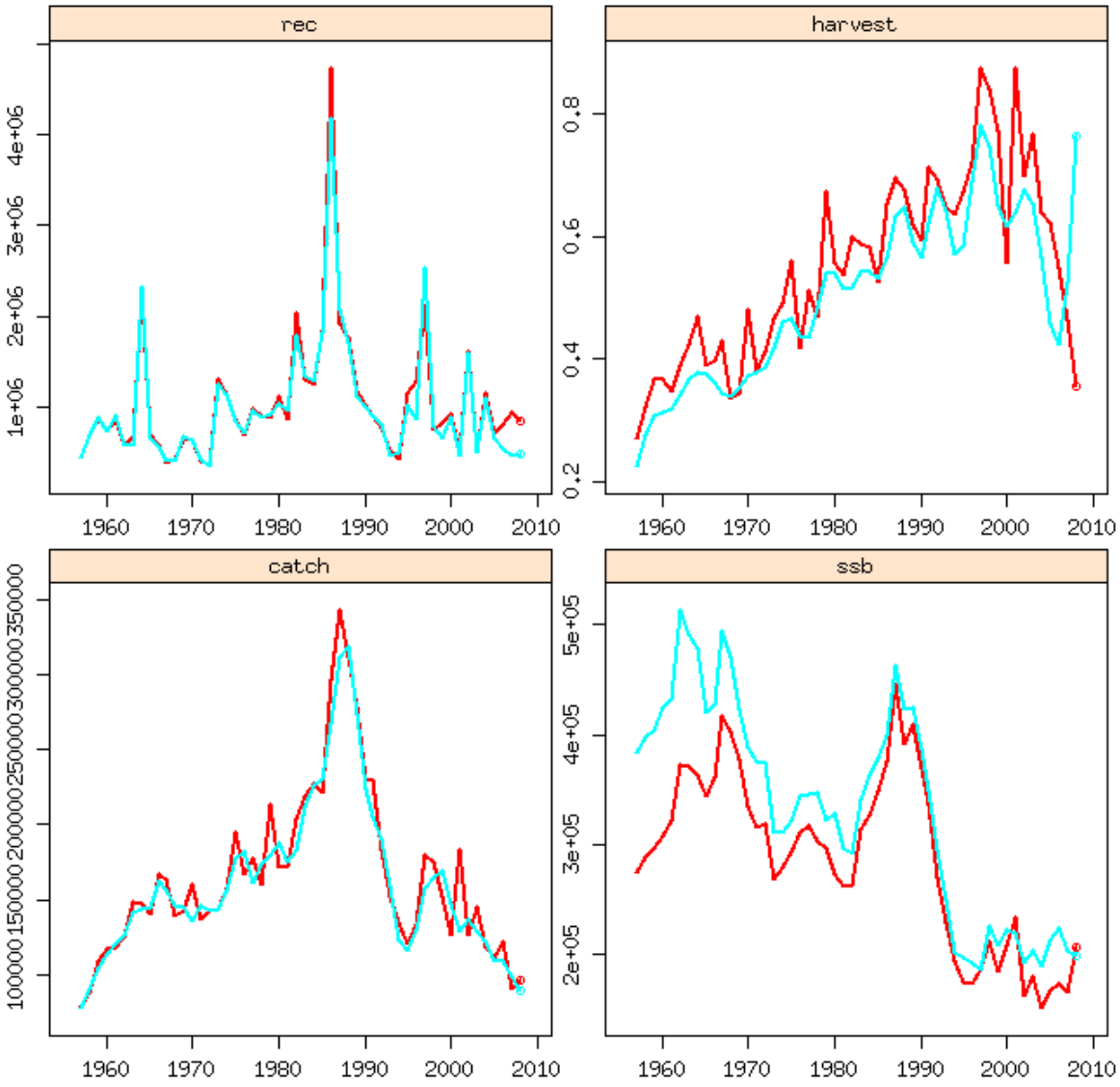

Figure 34: Stock summary

5.4.4 Stock-recruitment submodel

The S/R submodel is a special case, in the sense that it can be set up with the same linear tools as the F and Q models, but it can also use some hard coded models. The example shows how to set up a simple dummy model with `factor()`, a smooth model with `s()`, a Ricker model (`ricker()`), a Beverton and Holt model (`bevholt()`), a hockey stick model (`hockey()`), and a geometric mean model (`geomean()`). See Figure 35 for results. As mentioned before, the 'structural' models have a fixed variance, which must be set by defining the coefficient of variation. We now fix the F and Q submodels before fiddling around with the S/R model.

```
fmod <- ~s(age, k = 4) + s(year, k = 20)
qmod <- list(~s(age, k = 4))
```

```
srmod <- ~factor(year)
fit <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
srmod <- ~s(year, k = 20)
fit1 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
srmod <- ~ricker(CV = 0.05)
fit2 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
srmod <- ~bevholt(CV = 0.05)
fit3 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
srmod <- ~hockey(CV = 0.05)
fit4 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
srmod <- ~geomean(CV = 0.05)
fit5 <- sca(ple4, ple4.indices[1], fmod, qmod, srmod)
flqs <- FLQuants(fac = stock.n(fit)[1], smo = stock.n(fit1)[1], ric = stock.n(fit2)[1],
    bh = stock.n(fit3)[1], hs = stock.n(fit4)[1], gm = stock.n(fit5)[1])
```

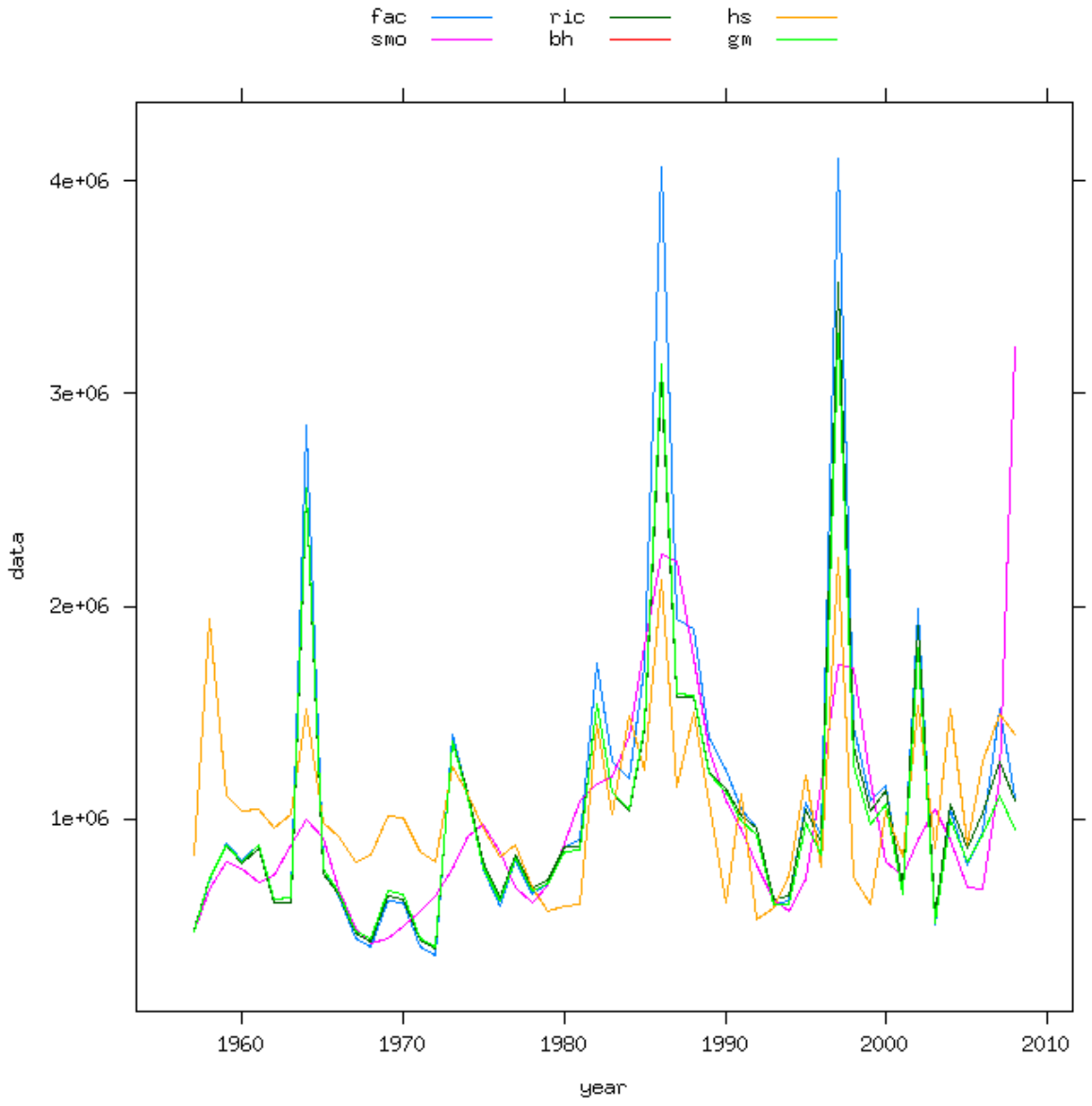

Figure 35: Recruitment models

5.5 The a4aSCA method - advanced features

A more advanced method for stock assessment can be used through the `a4aSCA()` method. This method gives access to the submodels for $N1$, σ^2_{ay} and I_{ays} as well as arguments to get the ADMB files, etc. Check the manual pages with `?a4aSCA` for more information. This method has 'assessment' as the default value for the `fit` argument, which means that the hessian is going to be computed and all the information about the parameters will be returned by default. Note that the default models of each submodel can be accessed with

```
# fmodel <- ~ s(age, k=4) + s(year, k = 20) qmodel <- list( ~ s(age, k=4) +
# year) srmodel <- ~s(year, k=20)
fit <- a4aSCA(ple4, ple4.indices[1])
submodels(fit)


##   fmodel: ~s(age, k = 3) + factor(year)
##  srmodel: ~factor(year)
##  n1model: ~factor(age)
##   qmodel:
##     BTS-Isis: ~1
##   vmodel:
##     catch:    ~s(age, k = 3)
##     BTS-Isis: ~1
```

5.5.1 N1 model

The submodel for the stock number at age in the first year of the time series is set up with the usual linear tools (Figure 36), but bare in mind that the year effect does not make sense here.

```
n1mod <- ~s(age, k = 4)
fit1 <- a4aSCA(ple4, ple4.indices[1], n1model = n1mod)
flqs <- FLQuants(smo = stock.n(fit1)[, 1], fac = stock.n(fit)[, 1])
```

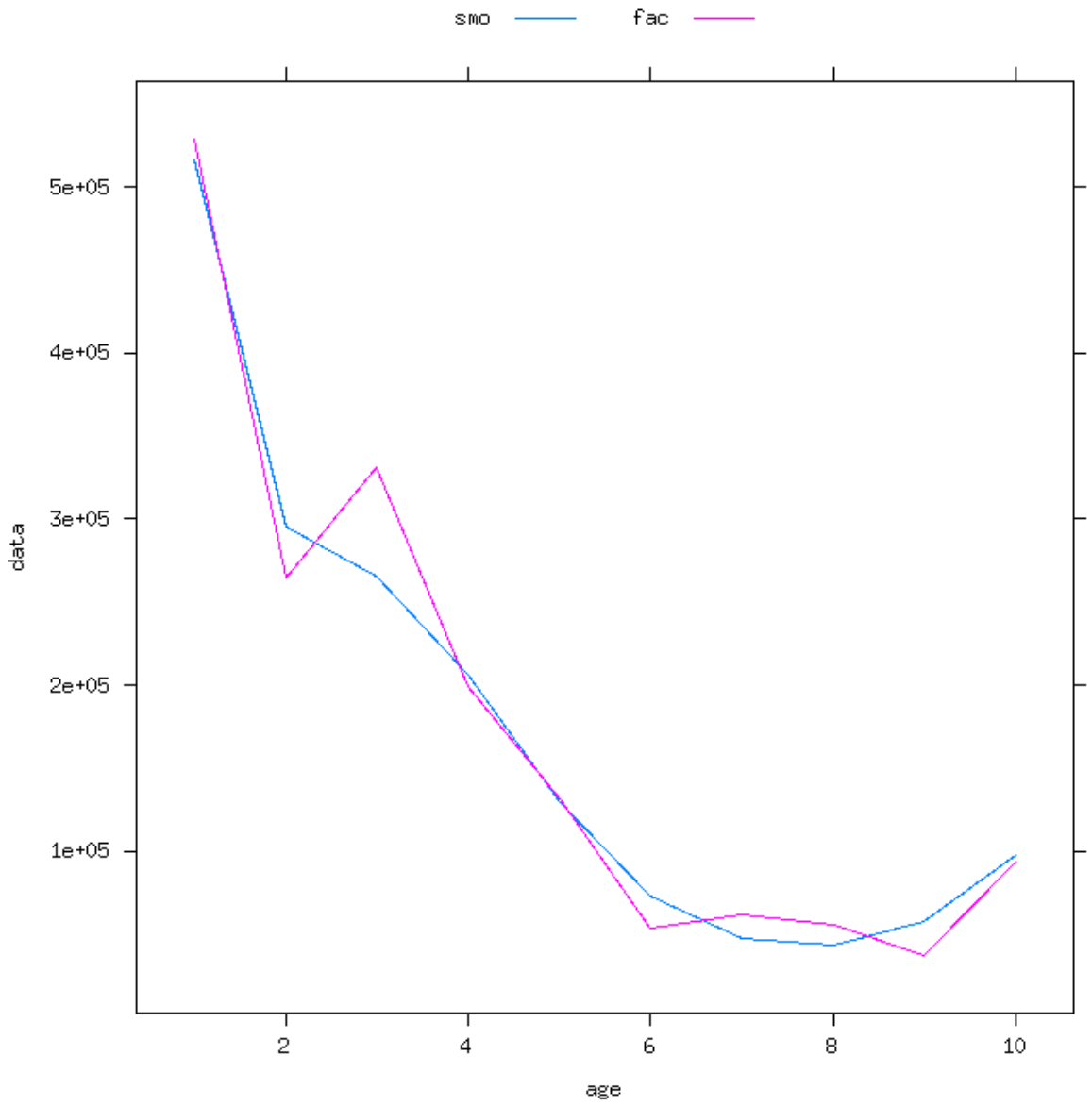

Figure 36: Nay=1 models

5.5.2 Variance model

The variance model allows the user to set up the shape of the observation variances σ^2_{ay} and I_{ays}. This is an important subject related with fisheries data used for input to stock assessment models. It's quite common to have more precision on the most represented ages and less precision on the less frequent ages. This is due to the fact that the last ages do not appear as often at the auction markets, in the fishing operations or on survey samples.

By default the model assumes constant variance over time and ages (1 model) but it can use other models specified by the user. As with the other submodels, R linear model capabilities are used (Figure 37).

```
vmod <- list(~1, ~1)
fit1 <- a4aSCA(ple4, ple4.indices[1], vmodel = vmod)
vmod <- list(~s(age, k = 4), ~1)
fit2 <- a4aSCA(ple4, ple4.indices[1], vmodel = vmod)
flqs <- FLQuants(cts = catch.n(fit1), smo = catch.n(fit2))
```

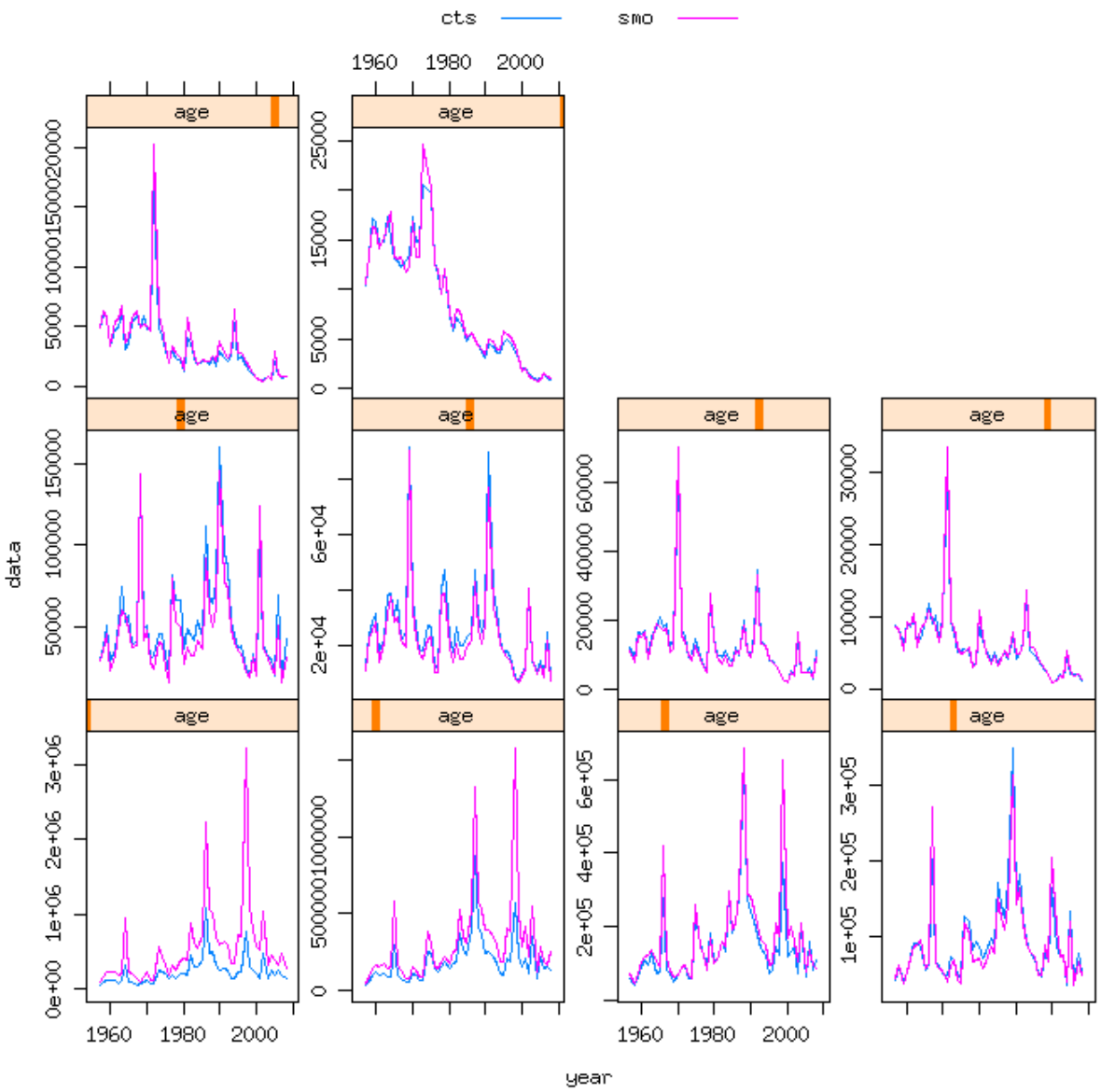

Figure 37: Population estimates using two different variance models

5.5.3 Working with covariates

In linear model one can use covariates to explain part of the variance observed on the data that the 'core' model does not explain. The same can be done in the a4a framework. The example below uses the North Atlantic Oscillation (NAO) index to model recruitment.

```
nao <- read.table("http://www.cdc.noaa.gov/data/correlation/nao.data", skip = 1,
    nrow = 62, na.strings = "-99.90")
dnms <- list(quant = "nao", year = 1948:2009, unit = "unique", season = 1:12,
    area = "unique")
nao <- FLQuant(unlist(nao[, -1]), dimnames = dnms, units = "nao")
nao <- seasonMeans(trim(nao, year = dimnames(stock.n(ple4))$year))
nao <- as.numeric(nao)
```

First by simply assuming that the index drives recruitment (Figure 38).

```
srmod <- ~nao
fit2 <- sca(ple4, ple4.indices[1], qmodel = list(~s(age, k = 4)), srmodel = srmod)
flqs <- FLQuants(simple = stock.n(fit)[1], covar = stock.n(fit2)[1])
```

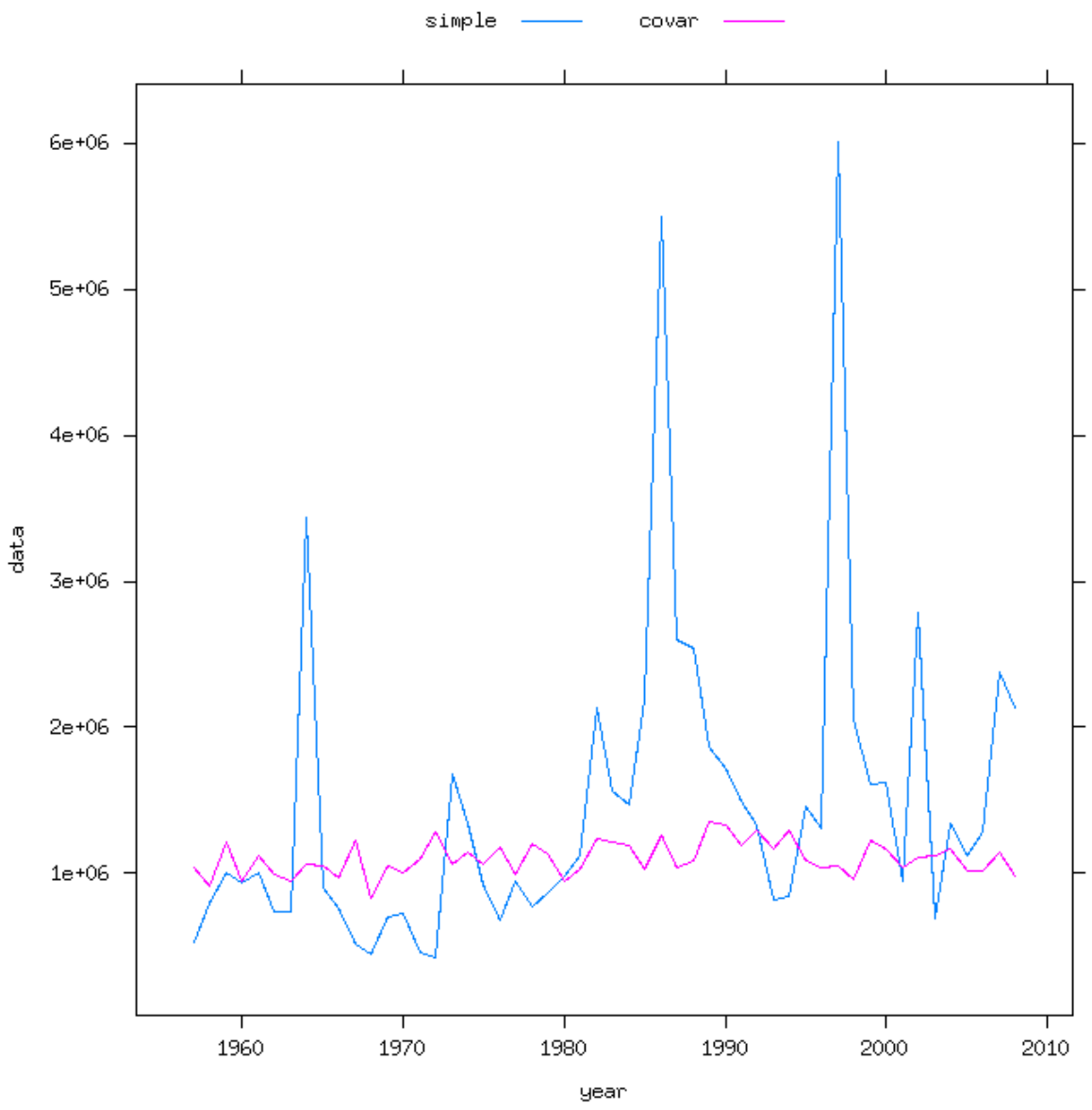

Figure 38: Recruitment model with covariates

In a second model we're using the NAO index not to model recruitment directly but to model one of the parameters of the S/R function (Figure 39).

```
srmod <- ~ricker(a = ~nao, CV = 0.1)
fit3 <- sca(ple4, ple4.indices[1], qmodel = list(~s(age, k = 4)), srmodel = srmod)
flqs <- FLQuants(simple = stock.n(fit)[1], covar = stock.n(fit3)[1])
```

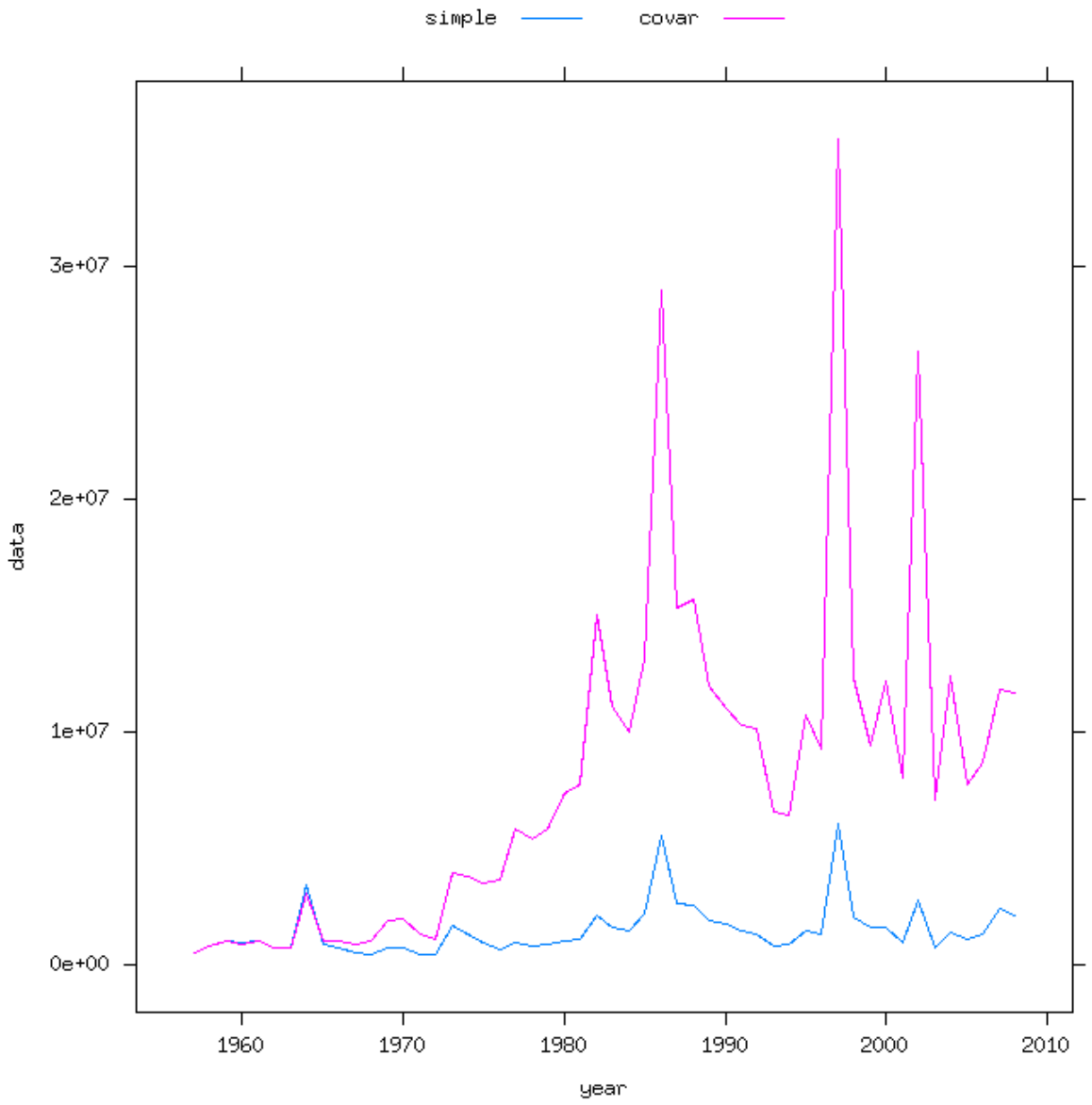

Figure 39: Recruitment model with covariates

Note that covariates can be added to any submodel using the linear model capabilities of R.

5.5.4 Assessing ADMB files

The framework gives access to the files produced to run the ADMB fitting routine through the argument `wkdir`. When set up all the ADMB files will be left in the directory. Note that the ADMB tpl file is distributed with the FLa4a. One can get it from your R library, under the folder `myRlib/FLa4a/admb/`.

```
fit1 <- a4aSCA(ple4, ple4.indices, wkdir = "mytest")

## Model and results are stored in working directory [mytest]
```

5.6 Predict and simulate

Predicting and simulating are important methods for statistical models. `R` uses the methods `predict()` and `simulate()`, which were implemented in `FLa4a` in the same fashion.

```
fit <- sca(ple4, ple4.indices[1], fit = "assessment")
```

5.6.1 Predict

Predict simply computes the quantities of interest using the estimated coefficients.

```
fit.pred <- predict(fit)
lapply(fit.pred, names)

## $stkmodel
## [1] "harvest" "rec"     "ny1"
##
## $qmodel
## [1] "BTS-Isis"
##
## $vmodel
## [1] "catch"    "BTS-Isis"
```

5.6.2 Simulate

Simulate uses the variance-covariance matrix computed from the Hessian returned by `ADMB` and the fitted parameters, to parametrize a multivariate normal distribution. The simulations are carried out using the method `mvrnorm()` provided by the R package `MASS`. Figure 40 shows a comparison between the estimated values and the medians of the simulation, while Figure 41 presents the stock summary of the simulated and fitted data.

```
fits <- simulate(fit, 100)
flqs <- FLQuants(sim = iterMedians(stock.n(fits)), det = stock.n(fit))
```

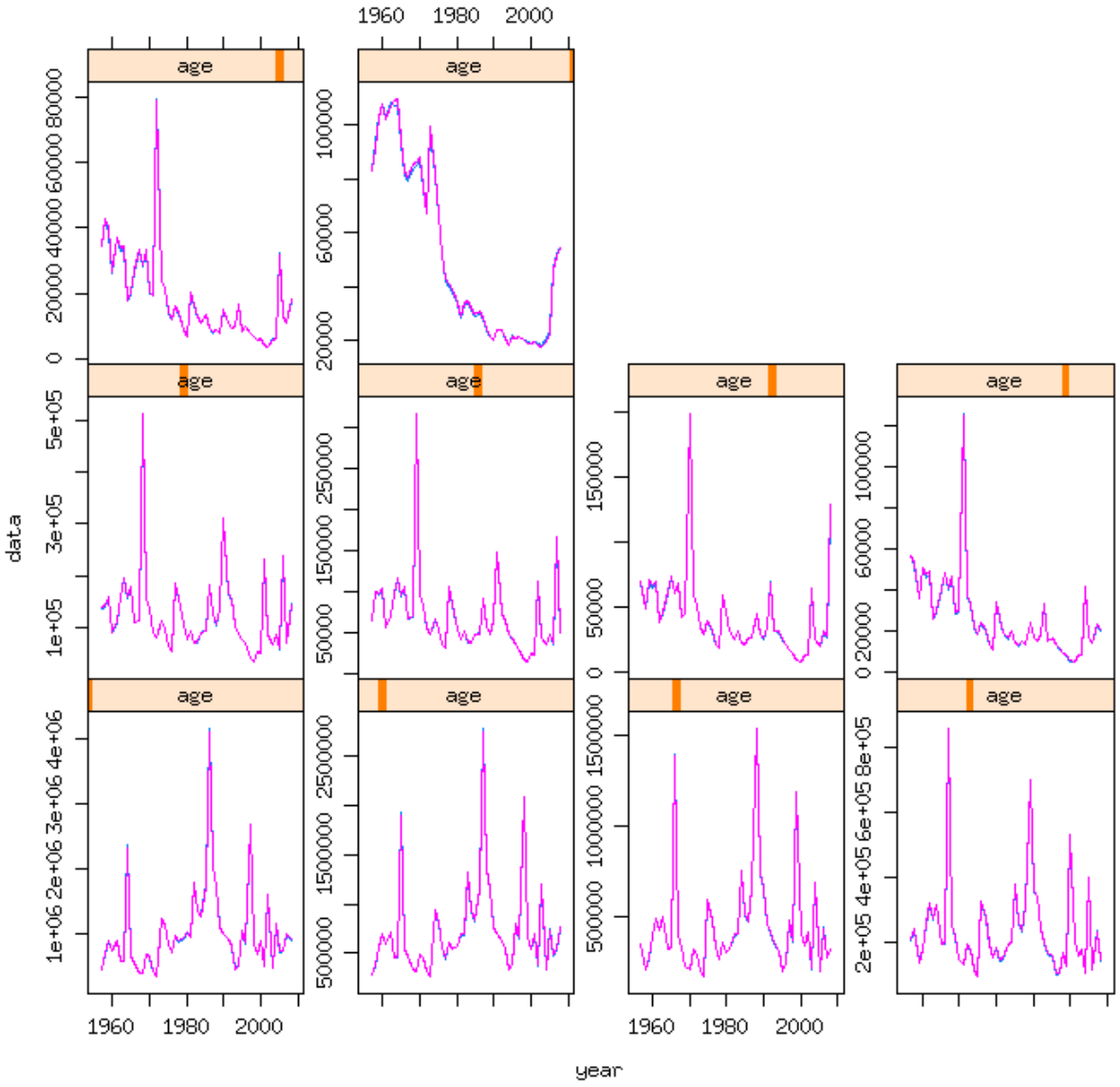

Figure 40: Median simulations VS fit

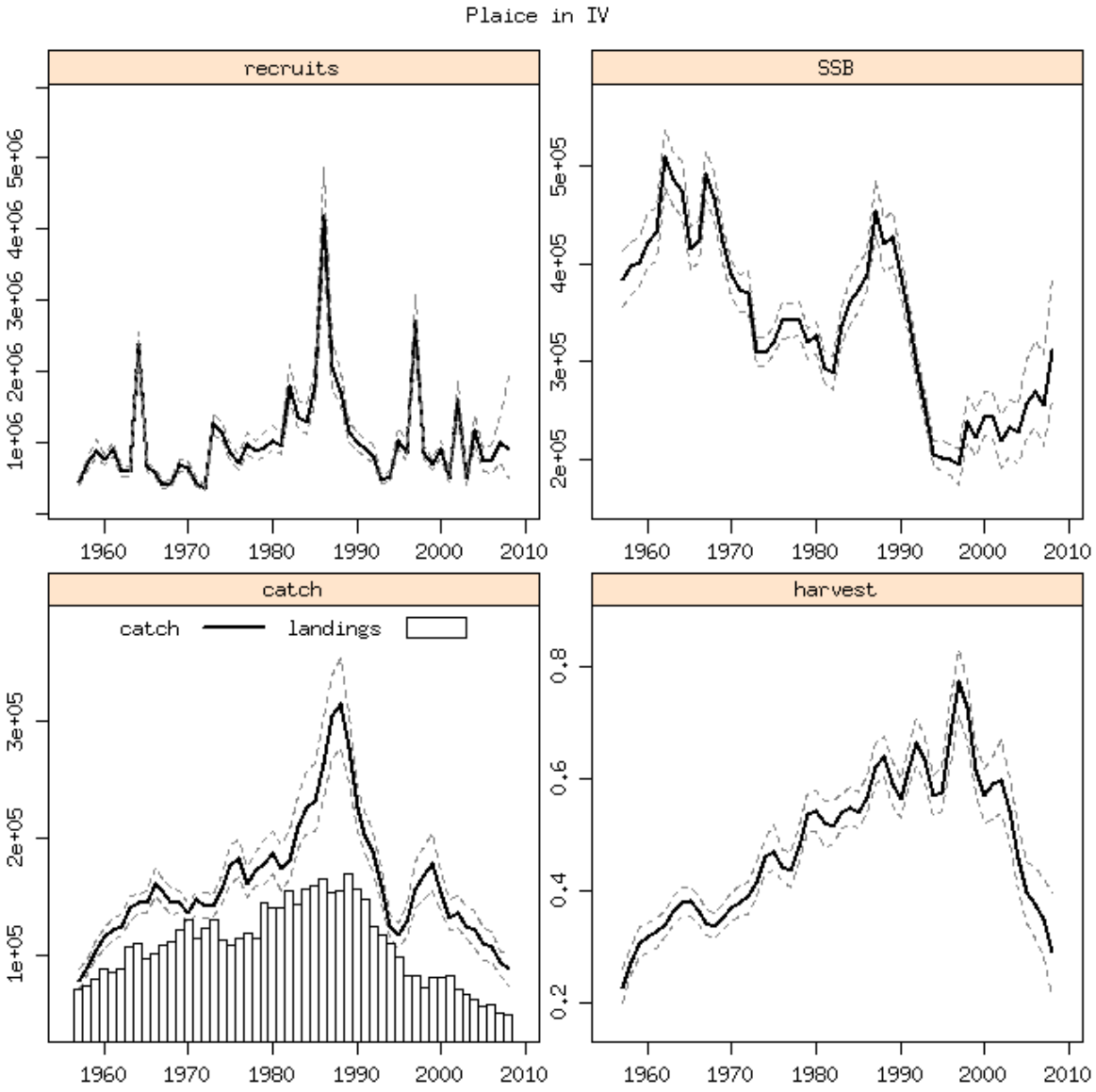

Figure 41: Stock summary of the simulated and fitted data

5.7 Geeky stuff

A lot more can be done with the `a4a` framework. The next sections will describe methods that are more technical. What we'd categorize as 'matters for geeks', in the sense that these methods usually will require the users to 'dive' into `R` a bit more.

```
fit <- sca(ple4, ple4.indices[1], fit = "assessment")
```

5.7.1 External weigthing of likelihood components

By default the likelihood components are weighted using inverse variance. However, the user may change the weights by setting the variance of the input parameters. This is done by adding a variance matrix to the `catch.n` and `index.n` slots of the stock and index objects. These variances will be used to penalize the data during the likelihood computation. The values should be given as coefficients of variation on the

log scale, so that variance is $\log(CV^2 + 1)$. Figure 42 shows the results of two fits with distinct likelihood weightings.

```
stk <- ple4
idx <- ple4.indices[1]
# variance of observed catches
varslt <- catch.n(stk)
varslt[] <- 0.4
catch.n(stk) <- FLQuantDistr(catch.n(stk), varslt)
# variance of observed indices
varslt <- index(idx[[1]])
varslt[] <- 0.1
index.var(idx[[1]]) <- varslt
# run
fit1 <- a4aSCA(stk, idx)
flqs <- FLQuants(nowgt = stock.n(fit), extwgt = stock.n(fit1))
```

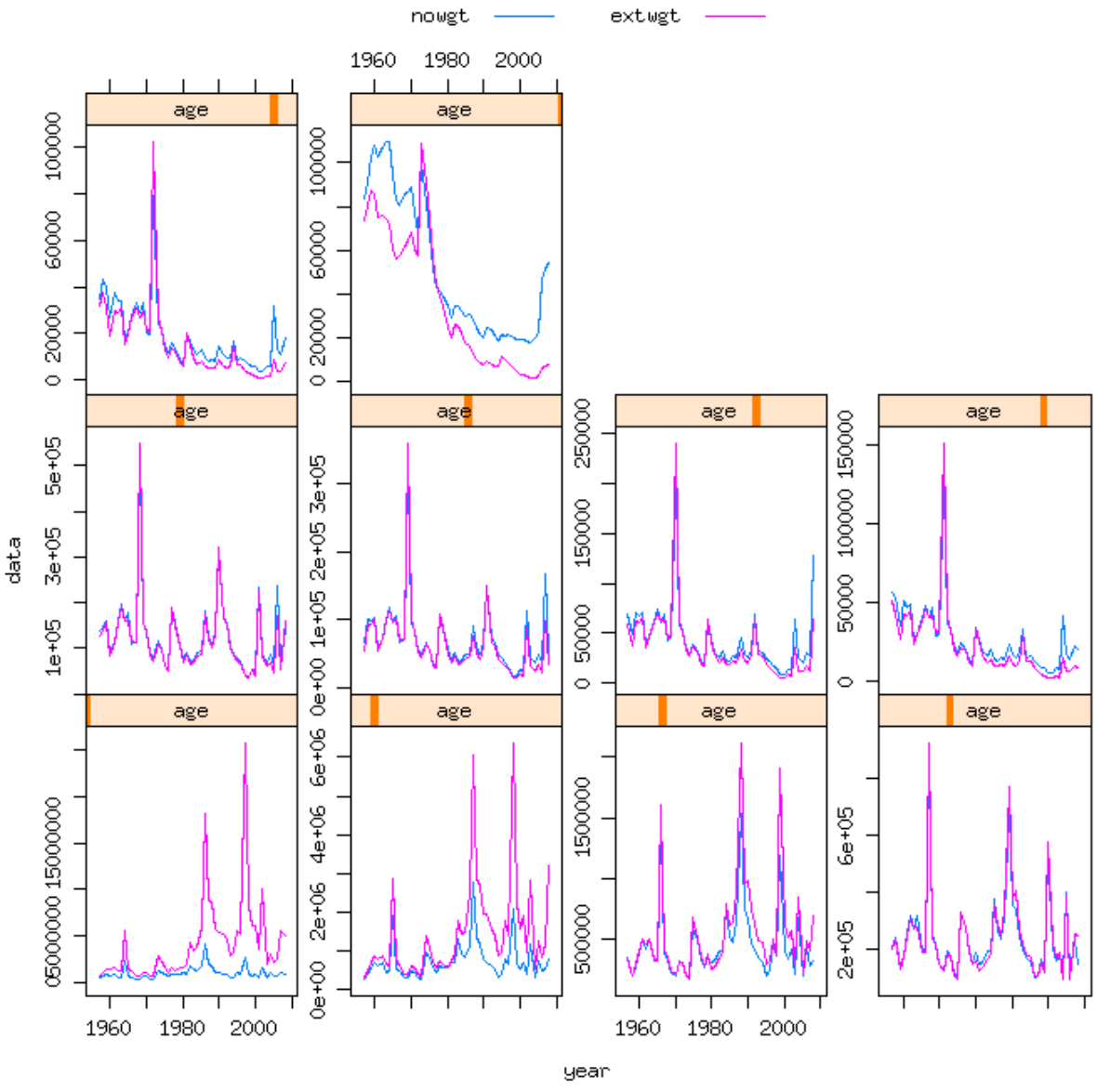

Figure 42: Stock summary of distinct likelihood weightings

5.7.2 More models

There's a set of methods that allow the user to have more flexibility on applying the models referred before. For example to break the time series in two periods, using the method `breakpts()`, or fixing some parts of the selection pattern by setting F to be the same for a group of ages, using `replace()`.

The example below (Figure 43) replaces all ages above 5 by age 5, which means that a single coefficient is going to be estimated for age 5-10.

```
fmod <- ~s(replace(age, age > 5, 5), k = 4) + s(year, k = 20)
fit <- sca(ple4, ple4.indices, fmod)
```

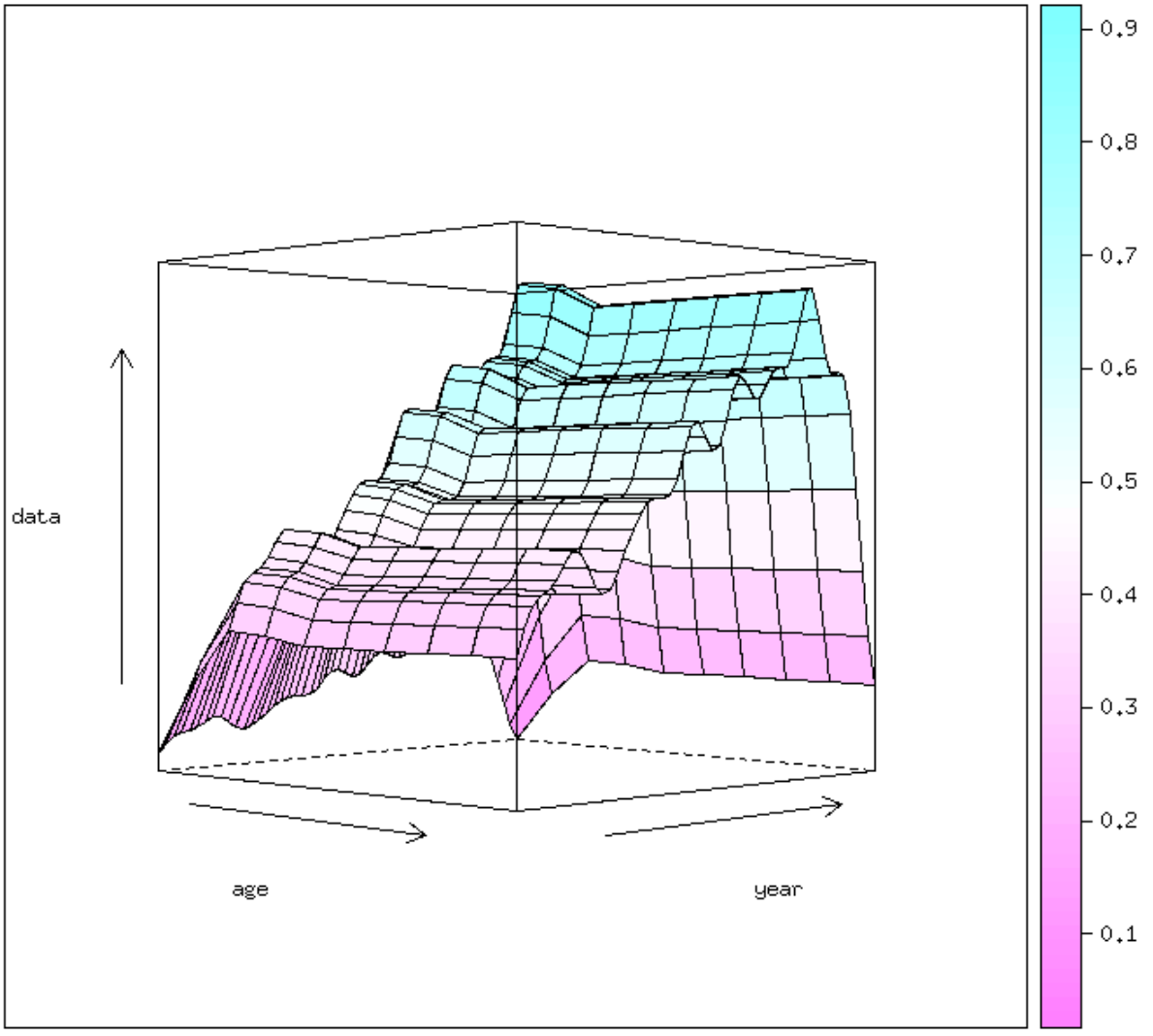

Figure 43: F-at-age fixed above age 5

In the next case we'll use the `breakpts()` to split the time series at 1990, although keeping the same shape in both periods, a thin plate spline with 3 knots (Figure 44).

```
fmod <- ~s(age, k = 3, by = breakpts(year, 1990))
fit <- sca(ple4, ple4.indices, fmod)
```

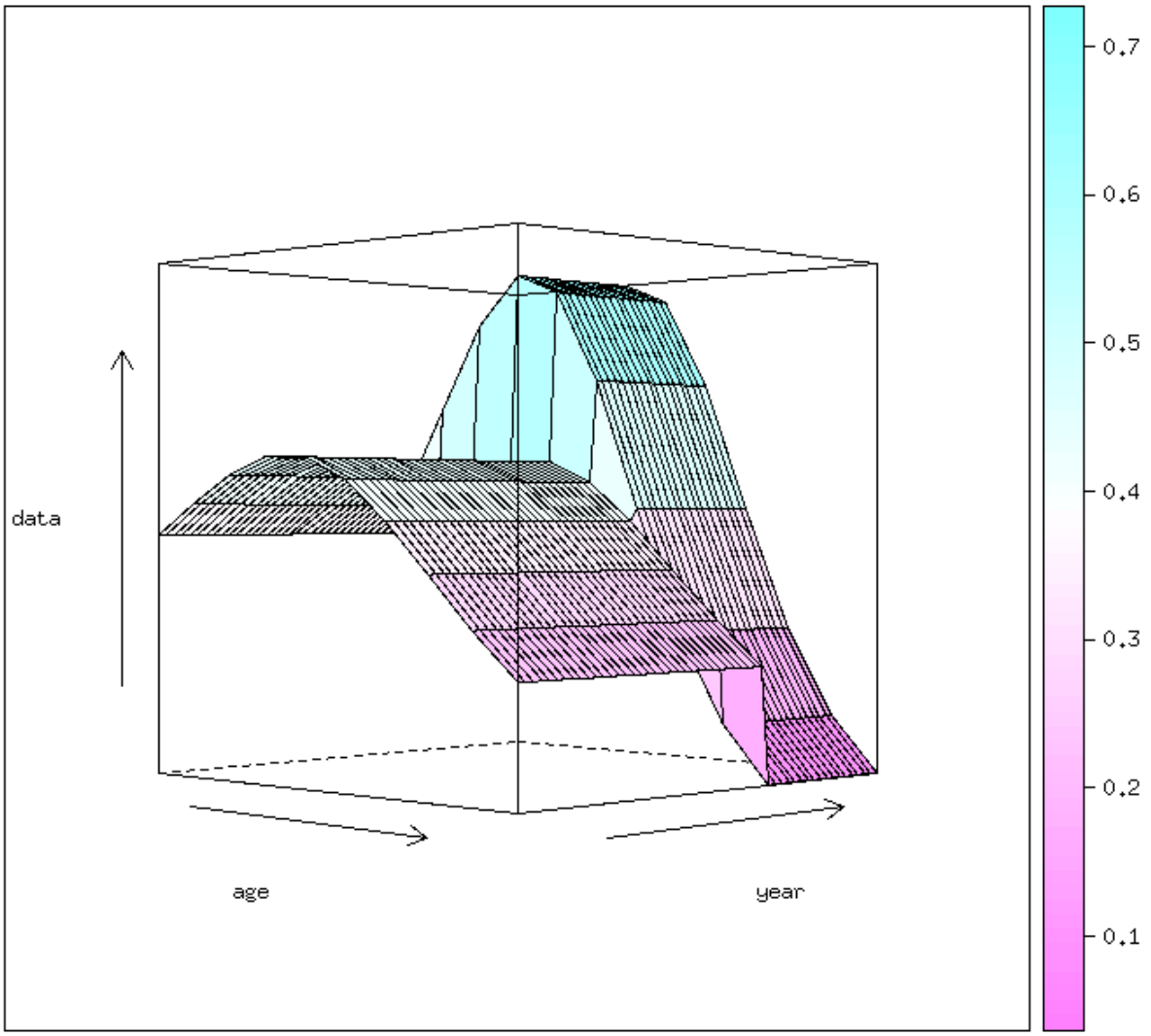

Figure 44: F-at-age in two periods using in both cases a thin plate spline with 3 knots

More complicated models can be built with these tools. For example, Figure 45 shows a model where the age effect is modelled as a smoother (the same thin plate spline) throughout years but independent from each other.

```
fmod <- ~factor(age) + s(year, k = 10, by = breakpts(age, c(2:8)))
fit <- sca(ple4, ple4.indices, fmod)
```

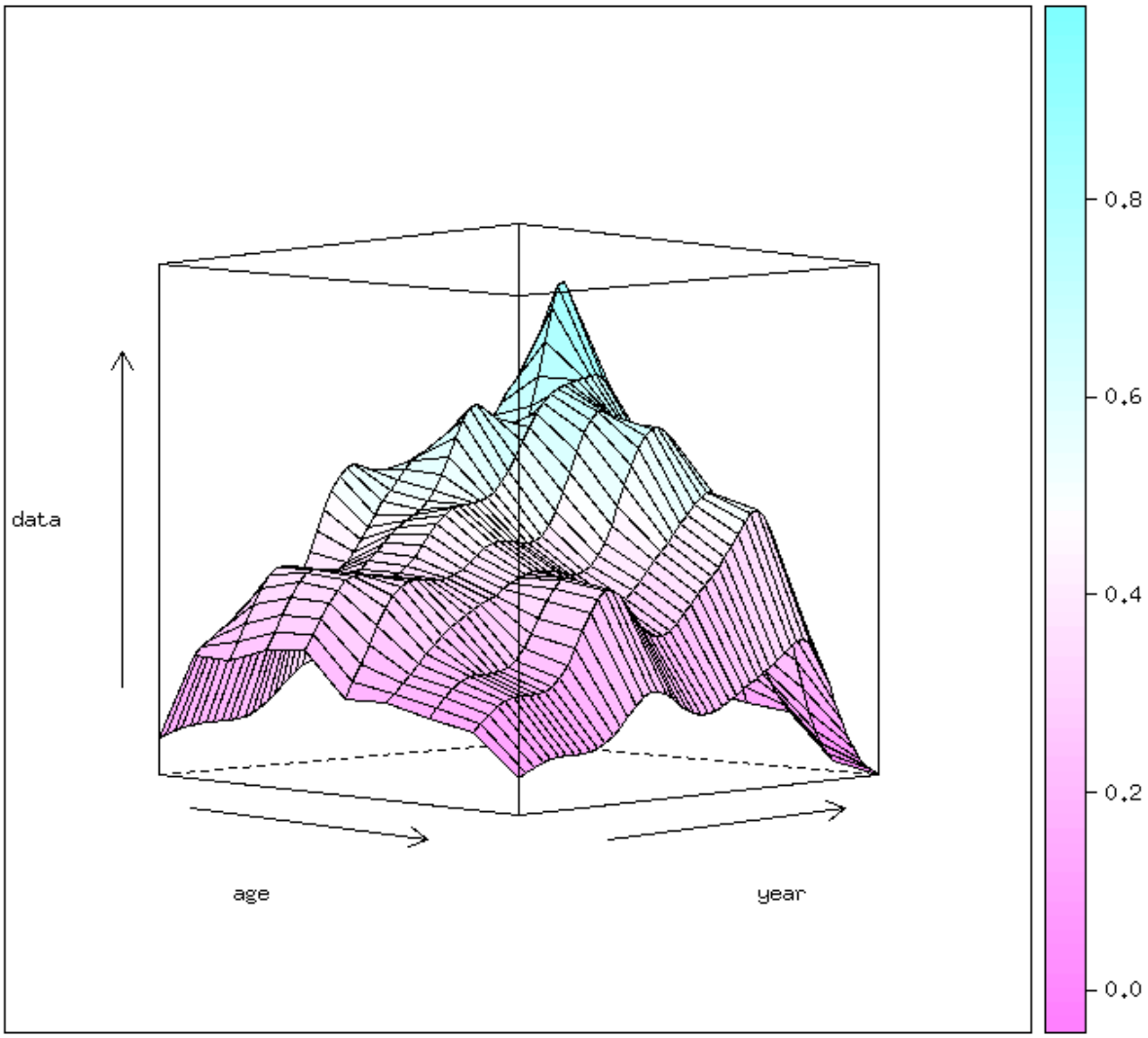

Figure 45: F-at-age as thin plate spline with 3 knots for each age

5.7.3 Propagate natural mortality uncertainty

In this section we give an example of how uncertainty in natural mortality, set up using the `m()` method and the class *a4aM* (see Section 4), is propagated through the stock assessment. We'll start by fitting the default model to the data.

```
data(ple4)
data(ple4.indices)
fit <- sca(ple4, ple4.indices)
```

Using the `a4a` methods we'll model natural mortality using a negative exponential model by age, Jensen's estimator for the level and a constant trend with time. We include multivariate normal uncertainty using the `mvrnorm()` method and create 25 iterations.

```
nits <- 25

shape <- FLModelSim(model = ~exp(-age - 0.5))
level <- FLModelSim(model = ~k^0.66 * t^0.57, params = FLPar(k = 0.4, t = 10),
    vcov = matrix(c(0.002, 0.01, 0.01, 1), ncol = 2))
trend <- FLModelSim(model = ~b, params = FLPar(b = 0.5), vcov = matrix(0.02))

m4 <- a4aM(shape = shape, level = level, trend = trend)
m4 <- mvrnorm(nits, m4)
range(m4)[] <- range(ple4)[]
range(m4)[c("minmbar", "maxmbar")] <- c(1, 1)
flq <- m(m4)[]
quant(flq) <- "age"
stk <- propagate(ple4, nits)
m(stk) <- flq
```

We fit the same model to the new stock object which has uncertainty in the natural mortality. The assessment is performed for each of the 25 iterations.

```
fit1 <- sca(stk, ple4.indices)
```

And compare the two results (Figure 46). It's quite easy to run these kind of tests and a large part of our effort is to create the tools to do so.

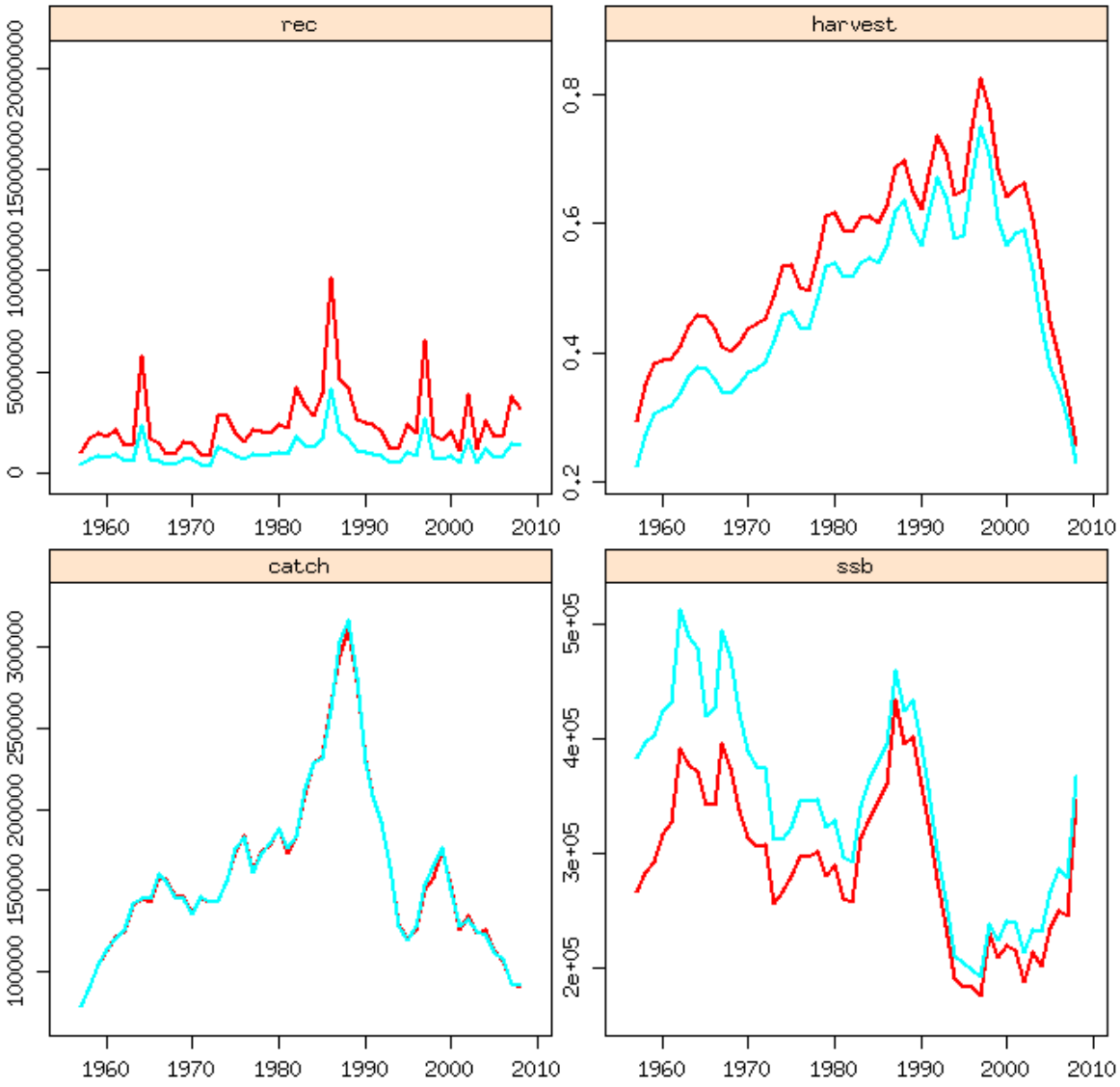

Figure 46: Stock summary for two M models

5.7.4 WCSAM exercise - replicating itself

The World Conference on Stock Assessment Methods (WCSAM) promoted a workshop where a large simulation study was used to test the performance of distinct stock assessment models. The first criteria used was that the models should be able to reproduce itself. The process involved fitting the model, simulating observation error using the same model, and refitting the model to each iteration. The final results should be similar to the fitted results before observation error was added (see Deroba, et.al, 2014 for details). The following analysis runs this analysis and Figure 47 presents the results.

```
nits <- 25
fit <- a4aSCA(ple4, ple4.indices[1])
stk <- ple4 + fit
set.seed(1234)
fits <- simulate(fit, nits)
stks <- ple4 + fits
idxs <- ple4.indices[1]
```

```
index(idxs[[1]]) <- index(fits)[[1]]
lst <- lapply(split(1:nits, 1:nits), function(x) {
    out <- try(a4aSCA(iter(stks, x), FLIndices(iter(idxs[[1]], x)), fit = "MP"))
    if (is(out, "try-error"))
        NULL else out
})

stks2 <- stks
for (i in 1:nits) {
    iter(catch.n(stks2), i) <- catch.n(lst[[i]])
    iter(stock.n(stks2), i) <- stock.n(lst[[i]])
    iter(harvest(stks2), i) <- harvest(lst[[i]])
}
catch(stks2) <- computeCatch(stks2)
stock(stks2) <- computeStock(stks2)
```

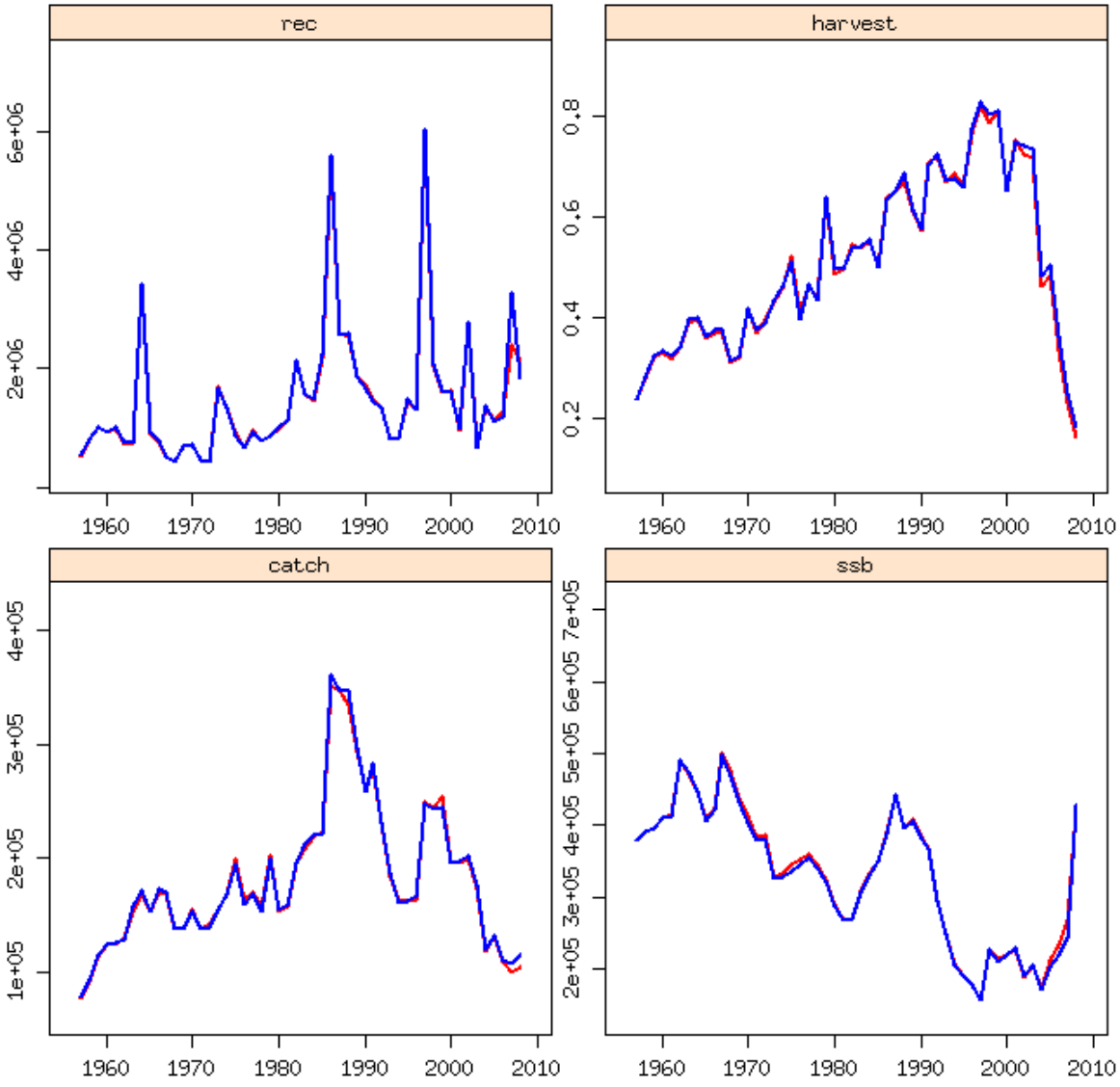

Figure 47: Replicating the stock assessment model (WCSAM approach)

5.7.5 Parallel computing

This is an example of how to use the `parallel` R package to run assessments. In this example each iteration is a dataset, including surveys, and we'll run one assessment for each iteration. Afterwards the data is pulled back together in an *FLStock* object and plotted (Figure 48). Only 20 iterations are run to avoid taking too long. Also note that we're using 4 cores. This parameter depends on the computer being used. These days almost all computers have at least 2 cores.

Finally, compare this code with the one for replicating WCSAM and note that it's exactly the same, except that we're using `mclapply()` from package `paralell` instead of `lapply()`.

```
data(ple4)
data(ple4.indices)
nits <- 25
fit <- a4aSCA(ple4, ple4.indices[1])
stk <- ple4 + fit
set.seed(1234)
fits <- simulate(fit, nits)
stks <- ple4 + fits
idxs <- ple4.indices[1]
index(idxs[[1]]) <- index(fits)[[1]]
library(parallel)
options(mc.cores = 4)
lst <- mclapply(split(1:nits, 1:nits), function(x) {
    out <- try(a4aSCA(iter(stks, x), FLIndices(iter(idxs[[1]], x)), fit = "MP"))
    if (is(out, "try-error"))
        NULL else out
})

stks2 <- stks
for (i in 1:nits) {
    iter(catch.n(stks2), i) <- catch.n(lst[[i]])
    iter(stock.n(stks2), i) <- stock.n(lst[[i]])
    iter(harvest(stks2), i) <- harvest(lst[[i]])
}
catch(stks2) <- computeCatch(stks2)
stock(stks2) <- computeStock(stks2)
stks3 <- FLStocks(orig = stk, sim = stks, fitsim = stks2)
```

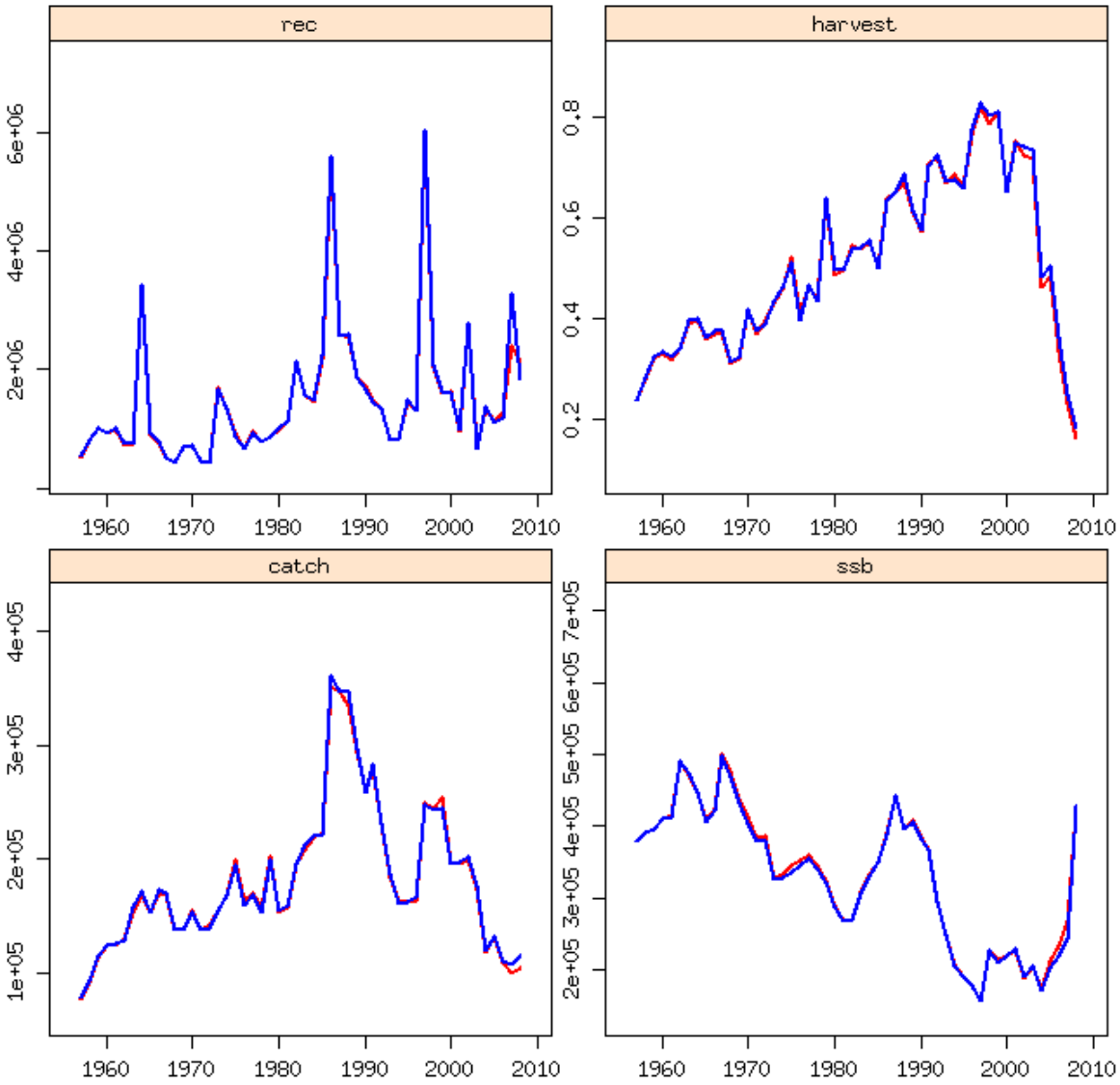

Figure 48: Replicating the stock assessment model (WCSAM approach) using parallel computing

5.8 Model averaging

To merge results from several fits, using distinct models or datasets, we follow Millar, et.al, 2014. The method `ma()` is the wrapper to the distinct methods, although for now only the AIC averaging is implemented. Figures 49 and 50 show the results.

```
data(ple4)
data(ple4.indices)
f1 <- sca(ple4, ple4.indices, fmodel = ~factor(age) + s(year, k = 20), qmodel = list(~s(age,
    k = 4), ~s(age, k = 4), ~s(age, k = 3)), fit = "assessment")
f2 <- sca(ple4, ple4.indices, fmodel = ~factor(age) + s(year, k = 20), qmodel = list(~s(age,
    k = 4) + year, ~s(age, k = 4), ~s(age, k = 3)), fit = "assessment")
stock.sim <- ma(a4aFitSAs(list(f1 = f1, f2 = f2)), ple4, AIC, nsim = 100)
stks <- FLStocks(f1 = ple4 + f1, f2 = ple4 + f2, ma = stock.sim)
```

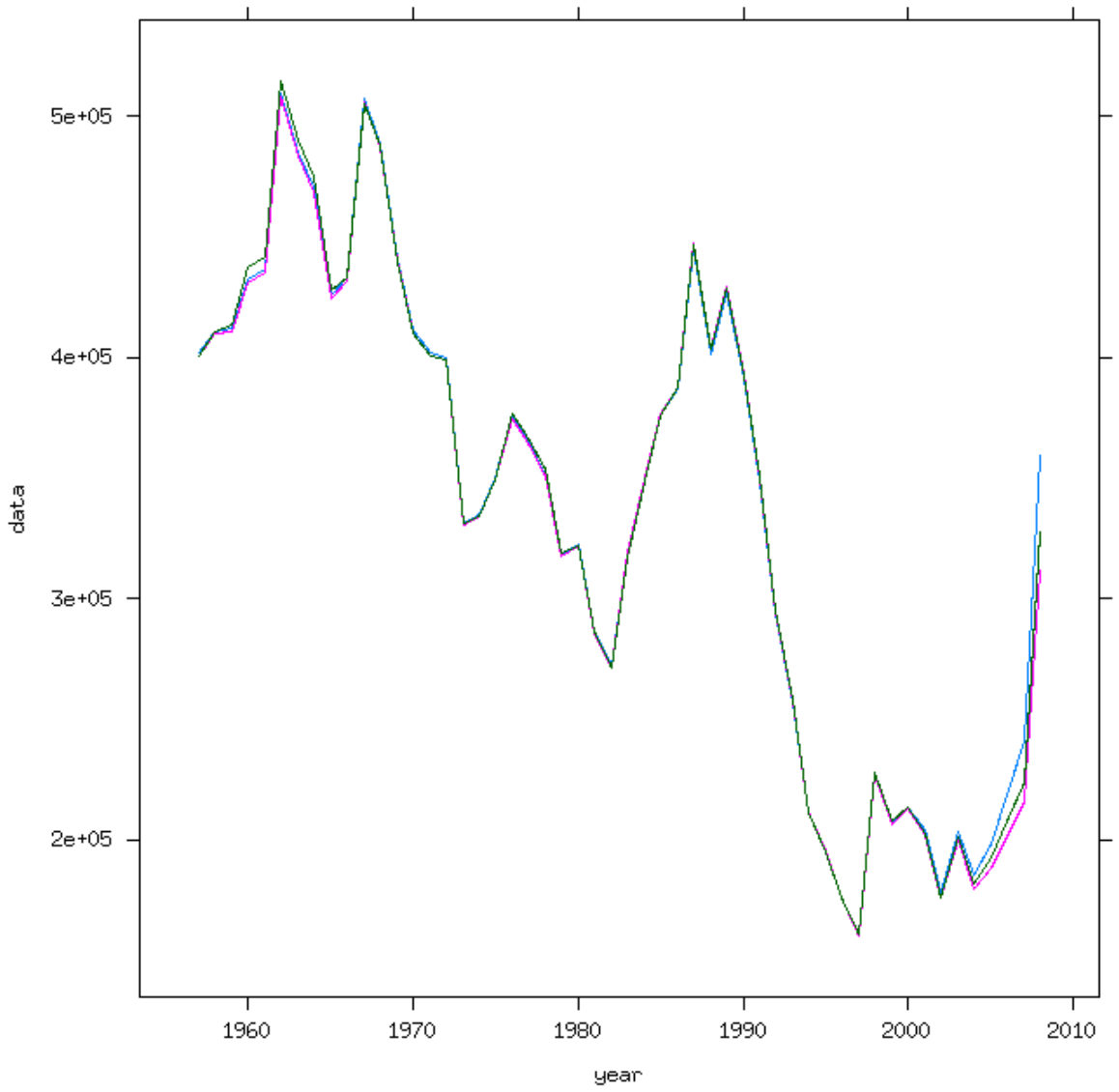

Figure 49: SSB of the two models and their average

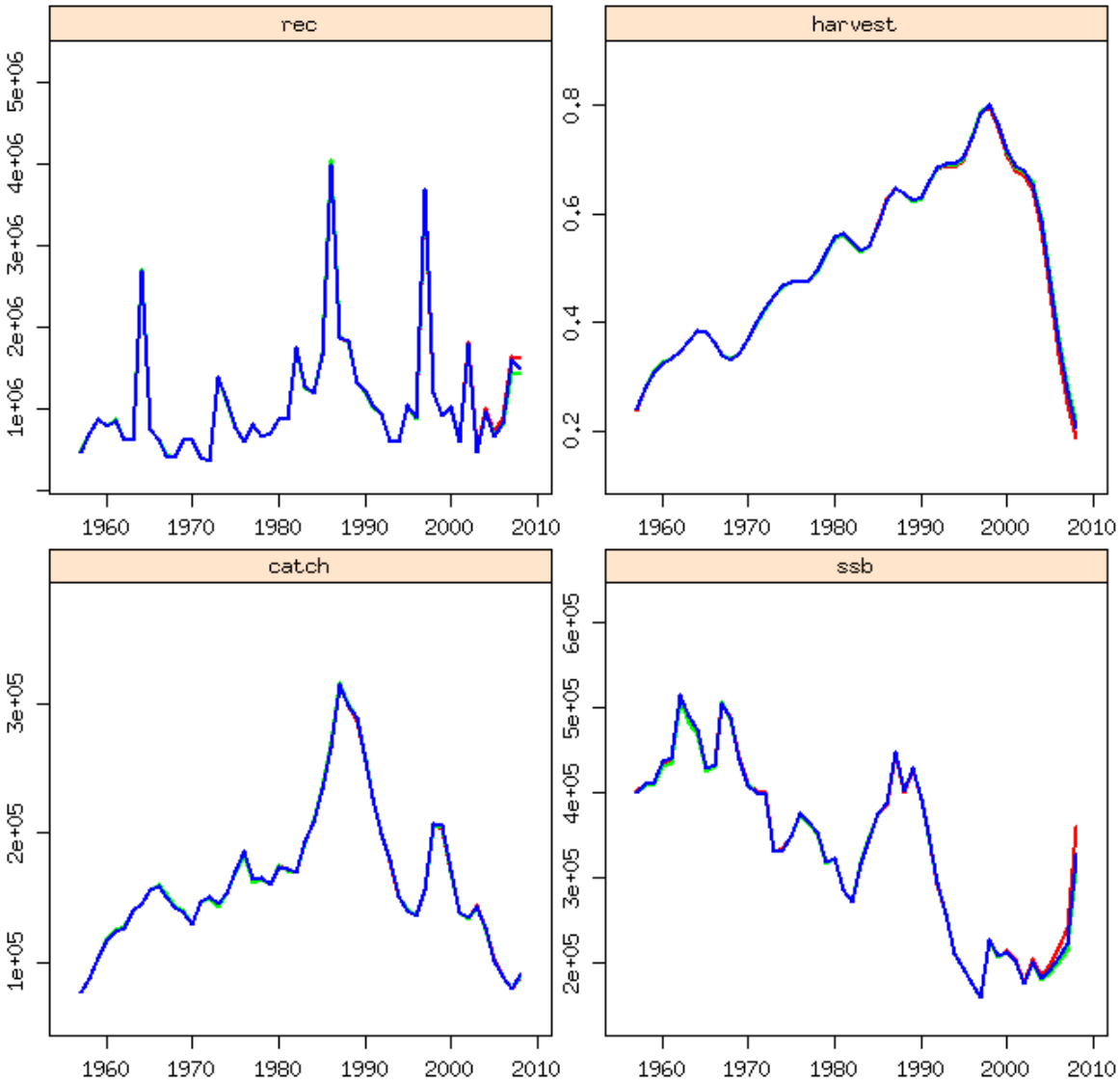

Figure 50: Stock summaries of the two models and their average