Multispecies, trait-based and community size spectrum ecological modelling in R (`mizer`)

Finlay Scott[1,4], Julia L. Blanchard[2] and Ken. H. Andersen[3]

[1]Cefas, Lowestoft, UK
[2]University of Sheffield, UK
[3]DTU Aqua, Copenhagen, DK
[4]Maritime Affairs Unit, IPSC, European Commission Joint Research Centre, IT

`mizer` version 0.2 , 2015-02-20

Contents

[1] This document is included as a vignette (a LaTeX document created using the R package `knitr`) of the package `mizer`. It is automatically downloaded together with the package and can be accessed through R typing `vignette("mizer_vignette")`.

1 Summary

Size spectrum ecological models have emerged as a conceptually simple way to model a large community of individuals that grow and change trophic level during their lives. They are a subset of physiologically structured models where growth (and thus maturation) is food dependent, and processes are formulated in terms of individual level processes. A key feature is that of a *size spectrum*, where the total abundance of individuals at size scales negatively with size: there are more small things than big things.

`mizer` is a software package for implementing size spectrum ecological models using the R statistical programming environment. The package has been developed to model marine ecosystems that are subject to fishing.

Roughly speaking there are three versions of the size spectrum modelling framework of increasing complexity: The *community* model, in which only one "species" is resolved; the *trait-based* model, in which all the species-specific parameters are the same, except for the asymptotic size which determines other life-history parameters such as the size at maturity; and the *multispecies* model in which multiple "real" species are resolved, each of which can have differing species-specific traits. The community and trait-based models can be considered as simplifications of the multispecies model. `mizer` is able to implement all three model versions using the same set of tools.

`mizer` contains routines and methods to allow users to set up the model community, and then project it through time under different fishing strategies. The results of the simulations can then be explored using a range of plots and summary methods, including plots of size spectra and the calculation of community indicators such as mean weight in the community and the the slope of the size spectrum.

The models created can be quite flexible and there are many options for setting up and running simulations. For example, different stock-recruitment relationships can be implemented and fishing gears with different selectivity patterns can be set up so that different species are caught by different gears. However, `mizer` aims to make setting up the models relatively simple. For example, easy to use "wrapper" functions are provided so users can set up community and trait-based models with the minimum of R gymnastics. Additionally, most of the methods and functions in the package have default options and assumptions. These can be changed by the user once they are familiar with the models

This vignette starts by summarising the principles and assumptions of size spectrum models. The `mizer` package is then introduced by showing how to set up the simplest types of size spectrum models: the community and trait-based model. These sections give a basic overview of the classes and methods used by `mizer`. Simple examples are given that demonstrate how trophic-cascades can be simulated. There then follows a more detailed description of the `mizer` classes and methods using a multispecies model of the North Sea as an example. Finally, there is a detailed example of a multispecies size spectrum model of the North Sea, including running projections using historical fishing patterns.

2 Package installation and getting help

2.1 Installing `mizer`

`mizer` is a package for the R open-source statistical programming language. R is available from the CRAN website, which is also an excellent source of documentation and tutorials.

The easiest way to install `mizer` (assuming you have an active internet connection) is to start an R session and then type:

```
install.packages("mizer")
```

After installing `mizer`, to actually use it you need to load the package using the `library()` function. Note that whilst you only need to install the package once, it will need to be loaded every time you start a new R session.

```
library(mizer)
```

The source code for `mizer` is currently hosted at Github (https://github.com/drfinlayscott/mizer). If you are feeling brave and wish to try out a development version of `mizer` you can install the package from here using the R package `devtools` (which was used extensively in putting together `mizer`).

2.2 Getting help

As you probably know, to access documentation in R you can use the `help()` function. This can be used to access package documentation in a range of ways. For example:

```
help(package="mizer")
help(mizer)
help(project)
```

The first command gives a technical summary of the package, including the available functions. The second command gives a brief introduction to `mizer`. The third gives the documentation page for the method `project()`.

Some methods are associated with several R classes. For example, the `plot()` method is generic and can be used on a wide range of R objects. The `plot()` method has been overloaded in `mizer` to plot *MizerSim* objects. To select the help page for the appropriate plotting method you can either just use `help(plot)` and then select from a menu of packages or you can supply the package name in the `help()` command:

```
help(plot, package="mizer")
```

There are other ways of accessing package documentation that you may not be familiar with. To access the the help page of a particular method you can use: `method ?  method-name`. For example, to access the help page of the `getFeedingLevel()` method you can use:

```
method ? getFeedingLevel
```

This also works for getting information on a class. For example, to get the help page on the *MizerParams* class you can use:

```
class ? MizerParams
```

These two ways of getting help can be useful when a class and a method have the same name. For example, *MizerParams* is a class and also the name of the method for creating *MizerParams* objects.

3 Size spectrum modelling - concepts, processes and assumptions

Size spectrum models have emerged as a conceptually simple way to model a large community of individuals which grow and change trophic level during life. There is now a growing literature describing different types of size spectrum models (e.g. Andersen and Beyer, 2006; Andersen et al., 2008; Benoît and Rochet, 2004; Hartvig, 2011; Hartvig et al., 2011; Law et al., 2009). The models can be used to understand how marine communities are organised (Andersen and Beyer, 2006; Andersen et al., 2009a; Blanchard et al., 2009) and how they respond to fishing (Andersen and Pedersen, 2010; Andersen and Rice, 2010). This section introduces the central assumptions, concepts, processes, equations and parameters of size spectrum models.

Roughly speaking there are three versions of the size spectrum modelling framework of increasing complexity: The *community* size spectrum model (Benoît and Rochet, 2004; Blanchard et al., 2009; Law et al., 2009; Maury et al., 2007), the *trait-based* size spectrum model (Andersen and Beyer, 2006; Andersen and Pedersen, 2010), and the *multispecies* spectrum model (Hartvig et al., 2011). The community and trait-based models can be considered as simplifications of the multispecies model. This section focuses on the multispecies model but is also applicable to the community and trait-based models. `mizer` is able to implement all three types of model using similar commands.

3.1 Central concepts and assumptions

Size spectrum models are a subset of physiologically structured models (De Roos and Persson, 2001; Metz and Diekmann, 1986) as growth (and thus maturation) is food dependent, and processes are formulated in terms of individual level processes. All parameters in the size spectrum models are related to individual weight which makes it possible to formulate the model with a small set of general parameters (Table 1), which has prompted the label "charmingly simple" to the model framework (Pope et al., 2006).

The model framework builds on two central assumption and a number of lesser standard assumption.

The first central assumption is that an individual can be characterized by its weight w and its species number i only. The aim of the model is to calculate the size- and trait-spectrum $\mathcal{N}_i(w)$ which is the density of individuals such that $\mathcal{N}_i(w)dw$ is the number of individuals in the interval $[w : w + dw]$. The dimensions of the size spectrum are numbers per weight per volume. Scaling from individual-level processes of growth and mortality to the size spectrum of each trait group is achieved by means of the McKendrik-von Foerster equation, which is simply a conservation equation:

$$\frac{\partial N_i(w)}{\partial t} + \frac{\partial g_i(w) N_i(w)}{\partial w} = -\mu_i(w) N_i(w) \tag{3.1}$$

where individual growth $g_i(w)$ and mortality $\mu_i(w)$ are both determined by the availability of food from the other species plus a background resource, $N_R(w)$, and predation by the other species. The conservation equation is supplemented by a boundary condition at the boundary at weight w_0 where the flux of individuals (numbers per time) $g_i(w_0)N_i(w_0)$ is determined by the reproduction of offspring by mature individuals in the population R_i:

$$g_i(w_0) N_i(w_0) = R_i. \tag{3.2}$$

The second central assumption is that the preference of food is determined by individual weight combined with a species preference. The preference for prey weight is described by the log-normal

selection model (Ursin, 1973) which prescribes prey preference in terms of the ratio between the weight of predators w and prey of weight w_p:

$$\phi(w_p/w) = \exp\left[\frac{-(\ln(w/(w_p\beta_i)))^2}{2\sigma_i^2}\right], \tag{3.3}$$

where β_i is the preferred predator-prey mass ratio and σ_i the width of the weight selection function.

The rest of the formulation of the model rests on a number of "standard" assumptions from ecology and fisheries science about how encounters between predators and prey leads to growth $g_i(w)$ and recruitment R_i of the predators, and mortality of the prey $\mu_i(w)$. The remainder of this section looks in detail at these assumptions.

3.2 Predator-prey encounter

The encounter of food is based on the "Andersen-Ursin" encounter model which was developed as part of a North-Sea ecosystem model (see also Andersen and Beyer, 2006; Andersen and Ursin, 1977).

The available food (mass per volume) for a predator of weight w is determined by integrating over all species and the background resource weighted by the size selection function (Equation 3.3):

$$E_{a.i}(w) = \int \left(N_R(w) + \sum_j \theta_{ij} N_j(w)\right) \phi(w_p/w) w_p \, dw_p. \tag{3.4}$$

where θ_{ij} is the preference of species i for species j. The food actually encountered $E_{e.i}$ (mass per time) depends on the search rate (volume per time) which is assumed to scale with individual weight as $\gamma_i w^q$:

$$E_{e.i}(w) = \gamma_i w^q E_{a.i}. \tag{3.5}$$

3.3 Consumption

The encountered food is consumed subjected to a standard Holling functional response type II to represent satiation. This determines the *feeding level*, $f_i(w)$, which is a dimensionless number between 0 (no food) and 1 (fully satiated):

$$f_i(w) = \frac{E_{e.i}}{E_{e.i} + h_i w^n}, \tag{3.6}$$

where $h_i w^n$ is the maximum consumption rate.

3.4 Growth

The consumed food $f_i(w) h_i w^n$ is assimilated by an efficiency α and used to fuel the needs for standard metabolism and activity $k_{s.i} w^p$. The remaining available energy, $\alpha f_i(w) h_i w^n - k_{s.i} w^p$, is divided between growth and reproduction by a function of weight changing between zero around the weight of maturation to one at the asymptotic weight where all available energy is used for reproduction:

$$\psi_i(w) = \left[1 + \left(\frac{w}{w_{m.i}}\right)^{-10}\right]^{-1} \left(\frac{w}{W_i}\right)^{1-n}, \tag{3.7}$$

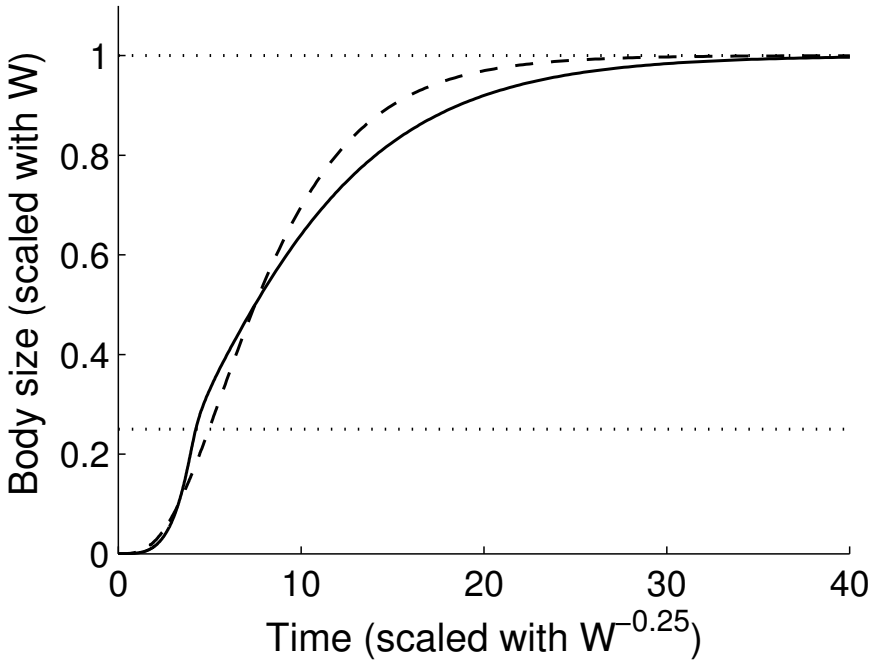

Figure 1: The growth curve of an individual found by solving (Equation 3.8) with a constant feeding level $f_0 = 0.6$ (solid line), compared to a von Bertalanffy growth curve with the von Bertalanffy growth parameter $K = (\alpha f_0 h w^n - k_s w^n) W^{n-1}/3$ (dashed line) (Andersen et al., 2009b).

where $w_{m.i}$ is the weight at maturation and W_i is the asymptotic (maximum) size. The term in the square bracket is a function which varies smoothly from 0 to 1 around the weight at maturation. The last term describes how the relative amount of energy invested in reproduction increases as the weight approaches the asymptotic weight. The somatic growth function therefore becomes:

$$g_i(w) = (\alpha f_i(w) h_i w^n - k_{s.i} w^p)(1 - \psi(w)). \tag{3.8}$$

The form of the allocation function (3.7) is chosen such that the growth curve approximates a von Bertalanffy growth curve if the feeding level is constant (see Figure 1 and Hartvig et al. (2011) for details about the derivation). The actual emerging growth curves from the model will depend on the feeding level, and may even result in stunted growth curves if the feeding level drops below the *critical feeding level* where the assimilated food is just enough to satisfy standard metabolism:

$$f_{c.i}(w) = k_{s.i} w^p / (\alpha h_i w^n). \tag{3.9}$$

If $n = p$ (which is the case for the set of default parameters chosen here), f_c is independent of w.

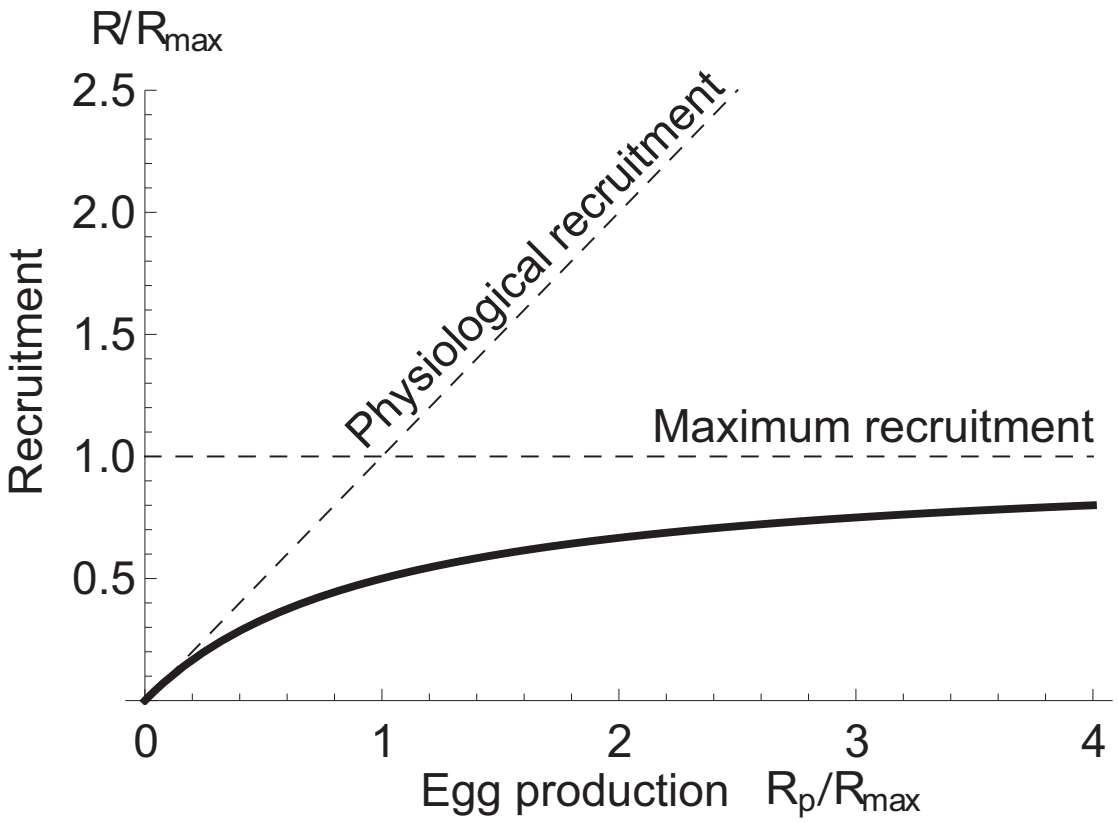

Figure 2: The Beverton-Holt recruitment function (thick line) which determines the recruitment R as a function of the egg production (the "physiological recruitment") R_p via (Equation 3.11).

3.5 Reproduction

The total production of eggs $R_{p.i}$ (numbers per time) is found by integrating the energy allocated to reproduction over all individuals:

$$R_{p.i} = \frac{\epsilon}{2w_0} \int N_i(w)(\alpha f_i(w) h_i w^n - k_{s.i} w^p)\psi_i(w)\, dw, \tag{3.10}$$

where w_0 is the egg weight, ϵ is the efficiency of reproduction, and the factor $1/2$ takes into account that only females reproduce.

3.6 Recruitment

One of the fundamental assumptions in fisheries science is that marine fish populations experience significant density dependence early in life (Ricker, 1954). This assumption has some empirical backing for selected, well-studied stocks (Myers and Cadigan, 1993) and is also supported by the general observation that marine fish do not experience strong, food-dependent growth. This is in contrast to fish in lakes where stunted growth is not uncommon. From a technical point of view, additional density dependence is needed to stabilize the model community to avoid some of the trait classes going extinct (Hartvig, 2011). An alternative way to ensure coexistence of many species is to introduce a random

food web matrix describing species-specific food preference (Hartvig et al., 2011) or an abstract notion of space (Hartvig, 2011).

In `mizer`, density dependence is modelled as a compensation on the egg production. This can be considered as the stock-recruitment relationship (SRR). The default functional form is such that the recruitment flux R_i (numbers per time) approaches a maximum recruitment as the egg production increases, modelled mathematically analogous to the Holling type II function response as a "Beverton-Holt" type of SRR:

$$R_i = R_{\max .i} \frac{R_{p.i}}{R_{p.i} + R_{\max .i}}, \tag{3.11}$$

where $R_{\max .i}$ is the maximum recruitment flux of each trait class (Figure 2).

The "Beverton-Holt" type of SRR is not the only density dependence model that `mizer` can use. Users are able to write their own model so it is possible to set a range of SRRs, e.g. fixed recruitment (as used in the community-type model) or "hockey-stick". This is explored in more detail in Section 8.4.

3.7 Mortality

The mortality rate of an individual $\mu_i(w)$ has three sources: predation mortality $\mu_{p.i}(w)$, starvation mortality $\mu_{s.i}(w)$, and a constant background mortality $\mu_{b.i}(w)$. The background mortality is needed to ensure that the largest individuals in the community also experience mortality as they are not predated upon by any individuals from the community spectrum. Predation mortality is calculated such that all that is eaten translates into corresponding predation mortalities on the ingested prey individuals (see Hartvig et al. (2011, Appendix A) for derivation):

$$\mu_{p.i}(w_p) = \sum_j \int \phi(w_p/w)(1 - f_j(w))\gamma_j w^q \theta_{ji} N_j(w)\, dw. \tag{3.12}$$

When food supply does not cover metabolic requirements $k_{s.i}w^p$, growth stops, i.e. there is no negative growth, and the individual is subjected to a starvation mortality. Starvation mortality is assumed proportional to the energy deficiency $k_{s.i}w^p - \alpha f_i(w) h_i w^n$, and is inversely proportional to lipid reserves, which are assumed proportional to body weight:

$$\mu_{s.i}(w) = \begin{cases} 0 & \alpha f_i(w) h_i w^n > k_{s.i} w^p \\ \frac{k_{s.i}w^p - \alpha f_i(w) h_i w^n}{\xi w} & \text{otherwise} \end{cases}. \tag{3.13}$$

Starvation is usually not an important process in the multispecies model with Beverton-Holt recruitment.

Mortality from sources other than predation and starvation is assumed to be constant within a species and inversely proportional to generation time (Peters, 1986):

$$\mu_{b.i} = \mu_0 W_i^{n-1}. \tag{3.14}$$

3.8 Resource dynamics

The background resource spectrum $N_R(w)$ represents food items for the smallest individuals (smaller than βw_0). The temporal evolution of each size group in the resource spectrum is described using semi-chemostatic growth:

$$\frac{\partial N_R(w,t)}{\partial t} = r_0 w^{p-1} \Big[\kappa_R w^{-\lambda} - N_R(w,t)\Big] - \mu_p(w) N_R(w,t), \tag{3.15}$$

where $r_0 w^{p-1}$ is the population regeneration rate (Fenchel, 1974; Savage et al., 2004) and $\kappa w^{-\lambda} = \kappa w^{-2-q+n}$ the carrying capacity.

3.9 Parameters

The default parameter values for the model have been derived on the basis of meta-analyses of data on marine fish (Table 1). Two parameters deserving special mention are the initial and critical feeding levels. These are used as physiological measures characterizing the productivity of the system and the metabolic requirements of the individuals.

The system is driven by the production of the resource. The ratio between the production and the maximum consumption will determine the feeding level f_0 of small individuals feeding mainly on the background resource. If f_0 is close to 1 the individuals are satiated, and growth will be largely independent of changes in food availability. If f_0 is in the linear range of the functional response, growth will be dependent on available food. If f_0 is close to the critical feeding level f_c, where the available food is only sufficient to cover standard metabolism, growth will be stunted. The initial feeding level f_0, calculated as the feeding level of an individual feeding on the resource at carrying capacity, is used as the control parameter for the resource productivity. The initial feeding level is then used to calculate the search rate parameter γ:

$$\gamma_i(f_0) = \frac{f_0 h_i \beta_i^{2-\lambda}}{(1-f_0)\sqrt{2\pi}\kappa\sigma_i}. \tag{3.16}$$

Similar to the way the initial feeding level f_0 is used to describe the relation between the production and the maximum consumption, the critical feeding level f_c (Equation 3.9) is used to describe the relation between the factors for standard metabolism k_s and maximum consumption h. A critical feeding level of $f_c \approx 0.2$ seems to be reasonable (Hartvig et al., 2011).

The free parameter κ may in principle be determined analytically, based on an assumption that all species have maximum recruitment, $R_i = R_{\max.i}$. However, the depletion of resources makes it difficult to find an analytical approximation of κ and until that has been achieved, κ has to be adjusted manually such that the resulting community spectrum forms a continuation of the resource spectrum.

4 Introducing `mizer`

With `mizer` is is possible to implement the three types of model mentioned above (with increasing complexity): community, trait-based and multispecies, using the same basic tools and methods. For the remainder of this vignette we present examples of how to set up, project and analyse all three types of model.

Using the package is relatively simple. There are three main stages to implementing a model:

- Setting the model parameters. This is done by creating an object of class *MizerParams*. This includes model parameters such as the life history parameters of each species, and the fishing gears. It is possible to create a *MizerParams* object directly using the class constructor or by using one of the convenient wrapper functions provided in the package.
- Running a simulation. This is done by calling the `project()` method on the model parameters. This produces an object of *MizerSim* which contains the results of the simulation. The `project()` method controls the length and time-step of the simulation, as well as the levels of fishing effort.

- Exploring the results. After a simulation has been run, the results can be examined using a range of plots and summaries.

These stages and the accompanying classes and methods are explained in detail in the rest of the document.

It is probably easier to learn the basics of `mizer` through examples. We start by looking at the simplest type of size spectrum model, the community model. We then move on to a more complex type of model, the trait-based model. Finally we look at the most complex type of model, the multispecies model.

5 Implementing a community-type model

5.1 Introduction

The simplest version of the size spectrum model is the community model, originally introduced by (Benoît and Rochet, 2004). In this model only one "species" is resolved. Reproduction is not considered, so $\psi(w) = 0$ and the recruitment flux R is set to be constant. The resource spectrum only extends to the start of the community spectrum. Standard metabolism is turned off by setting k_s to 0. Growth is therefore given by the simpler equation:

$$g(w) = \alpha f(w) h w^n \tag{5.1}$$

where $f(w)hw^n$ is the consumed food and α is the efficiency with which the consumed food is transformed into growth. The growth rate can also be written as:

$$g(w) = \bar{\epsilon}_I E_a \tag{5.2}$$

where $\bar{\epsilon}_I$ is the average growth efficiency which can be derived from the full model (see Andersen et al., 2009a; Zhang et al., 2013).

In this section we describe how a community model can be set up and projected through time. We then use a community model to illustrate the idea of a 'trophic cascade'. Due to the relative simplicity of this type of model they are useful for gently introducing some of the concepts behind the `mizer` package. Consequently, this section should hopefully serve as an introduction to using `mizer` and some of the main classes and methods.

5.2 Setting up a community model

As mentioned above, the first stage in implementing a model using `mizer` is to create an object of class *MizerParams*. This class contains the model parameters including the life-history parameters of the species in the model, the stock-recruitment relationships, the fishing selectivity functions and the parameters of the resource spectrum. The class is fully described in Section 8.

To avoid having to make a *MizerParams* object directly, a wrapper function, `set_community_model()`, has been provided that conveniently creates a *MizerParams* object specifically for a community model. The documentation for the function can be seen by entering:

```
?set_community_model
```

As can be seen in the help page, the function can take many arguments. These are passed on to the *MizerParams* constructor when the function is called. We can ignore most of these arguments for the

moment as they almost all come with default values. Full details of those arguments can be seen in Section 8.

The arguments that you should pay attention to are: `z0` (the level of background mortality), `alpha` (the assimilation efficiency of the community), `f0` (the average feeding level of the community which is used to calculate γ in equation 3.16) and `recruitment` (the level of constant recruitment).

Although default values for these parameters are provided, you are encouraged to explore how changing the values affects the simulated community. For example, the default value of `z0` is 0.1. Increasing this value effectively 'shortens' the length of the community spectrum. The value of the constant recruitment should be set so that the community spectrum is a continuation of the background spectrum. This can be done with trial and error. A reasonable value for `alpha` is 0.2 (Andersen et al., 2008).

The `set_community_model()` function is called by passing in the arguments by name. Any parameter that is not passed in is set to the default value. For example, the following line sets up the parameters with `z0` = 1, `f0` = 0.7, `alpha` = 0.2 and `recruitment` = 4e7. All other parameters will have their default value:

```
params <- set_community_model(z0 = 0.1, f0 = 0.7, alpha = 0.2, recruitment = 4e7)
```

Calling the function creates and returns an object of type *MizerParams*. We can check this using the `class()` function.

```
class(params)

## [1] "MizerParams"
## attr(,"package")
## [1] "mizer"
```

If you are going through this vignette for the first time, it is likely that you have no idea what a *MizerParams* object actually is. In "R-speak" it is an `S4` object, which means it is an object made up of "slots". Slots are essentially containers that store the object data. The names of these slots can be seen by calling the `slotNames()` method on the object:

```
slotNames(params)

##  [1] "w"              "dw"             "w_full"         "dw_full"
##  [5] "psi"            "intake_max"     "search_vol"     "activity"
##  [9] "std_metab"      "pred_kernel"    "rr_pp"          "cc_pp"
## [13] "species_params" "interaction"    "srr"            "selectivity"
## [17] "catchability"
```

As you can see, the `params` object is made up of lots of slots. They are discussed in detail in Section 8. A quick description can be found in the *MizerParams* help page (run `class ?  MizerParams`. The slots of an object are accessed by using the `@` operator. For example, to access the `w` slot (which contains a vector of the size bins in the model) you would use:

```
params@w
```

Rather than picking through the slots to find out what is in a particular *MizerParams* object, a summary of the object can be seen by calling the `summary()` method on it:

```
summary(params)

## An object of class "MizerParams"
## Community size spectrum:
##  minimum size: 0.001
##  maximum size: 1e+06
##  no. size bins: 100
## Background size spectrum:
##  minimum size: 1e-10
##  maximum size: 1e+06
##  no. size bins: 130
## Species details:
##     species w_inf w_mat beta sigma
## 1 Community 9e+05    NA  100     2
## Fishing gear details:
##  Gear Target species
##   Community   Community
```

In the summary you can see that the size range of the community spectrum has been set from 0.001 to 10^6 and these are spread over 100 size bins. Similar information is available for the background resource spectrum. Additionally, the community is made up of only one species, called "Community", which has an asymptotic size of 9×10^5 and a preferred predator prey mass ratio of 100. The `w_mat` parameter has been set to `NA` as it is not used when running a community model. These values have all been set by default using the `set_community_model()` function. If you want to set different values for these, you will need to call the `set_community_model()` function and pass in the the desired argument values.

5.3 Running the community model

By using the `set_community_model()` method we now have a *MizerParams* object that contains all the information we need about the model community. We can use this to perform a simulation and project the community through time. In the `mizer` package, projections are performed using the `project()` method. You can see the help page for `project()` for more details and it is described fully in Section 9. We will ignore the details for the moment and just use `project()` to run some simple projections. The arguments for `project()` that we need to be concerned with are `effort`, which determines the fishing effort (and therefore fishing mortality) through time, and `t_max`, which is the length of the simulation. Initial population abundances are set automatically by the `get_initial_n()` function. It is possible to set your own initial abundances but we will not do this here.

To run a projection for 50 time steps, with no fishing effort (i.e. we want to model an unexploited community) we run:

```
sim <- project(params, t_max=50, effort = 0)
```

The resulting object, `sim`, is of type *MizerSim*. This class holds the results of the simulation, including the community and background resource abundances at size through time, as well as the original model parameters. It is explained in detail in Section 9.

After running the projection, it is possible to explore the results using a range of plots and analyses. These are described fully in Section 10. To quickly look at the results of the projection you can call

the generic `plot()` method. This plots the feeding level, predation mortality, fishing mortality and abundance by size in the last time step of the simulation, and the biomass through time (Figure 3) Each of the plots can be show individually if desired.

```
plot(sim)
```

In Figure 3 there are several things going on that are worth talking about. Looking at the total biomass of the community against time, you can see that the biomass quickly reaches a stable equilibrium. The other panels show what is happening at the last time step in the simulation, which in this case is when the community is at equilibrium. Fishing mortality is 0 because we set the `effort` argument to 0 when running the simulation. The predation mortality rate (M2) is clearly a function of size, with the smallest sizes experiencing the highest levels of predation. The feeding level describes how satiated an individual is, with 0 being unfed, and 1 being full satiated. The feeding level at size will be strongly affected by the values of the `f0` and `alpha` arguments passed to the `set_community_model()` function. The background resource and community spectra are shown in the bottom panel of the plot (the plotted background resource spectrum has been truncated to make for a better plot, but really extends all the way back to 10^{-10} g). You can see that the community spectrum forms a continuum with the resource spectrum. This is strongly affected by the level of fixed recruitment (the `recruitment` argument passed to `set_community_model()`).

Note the "hump" in the biomass at the largest end of the community spectrum. This is because the size spectrum model can be broadly described as 'big things eating little things'. Given this, what is eating the very biggest things? Without fishing pressure, the mortality of the largest individuals is only from the background mortality (determined by the `z0` argument) and the mortality from predation is almost 0. This is difficult to see in the plot due to the M2 being so high for the smaller individuals.

We can see this more clearly by extracting the predation mortality information from the *MizerSim* object, `sim`, that we created above. This is easily done by using the `getM2()` method (see the help page for more details). There are several methods that can be used for extracting information from a *MizerSim* object, e.g. `getFeedingLevel()` and `getFMort()`. For more information see Section 10. Here we just call `getM2()` using the `sim` object:

```
m2 <- getM2(sim)
```

This `m2` object is a array that contains the predation mortality at time by species by size. Here we only have one species so the species dimension is dropped, leaving us with a two dimensional array of time by size. We projected the model for 50 time steps but the length of the time dimension is 51 as the initial population is also included as a time step.

To pull out the predation mortality at size in the final time step we use:

```
m2[51,]
```

If you plot this predation mortality on a log-log scale you can see how the predation mortality declines to almost zero for the largest sizes (Figure 4).

5.4 Example of a trophic cascade with the community model

It is possible to use the community model to simulate a trophic cascade, similar to those seen in Andersen and Pedersen (2010). To do this we need to perform two simulations, one with fishing and one without.

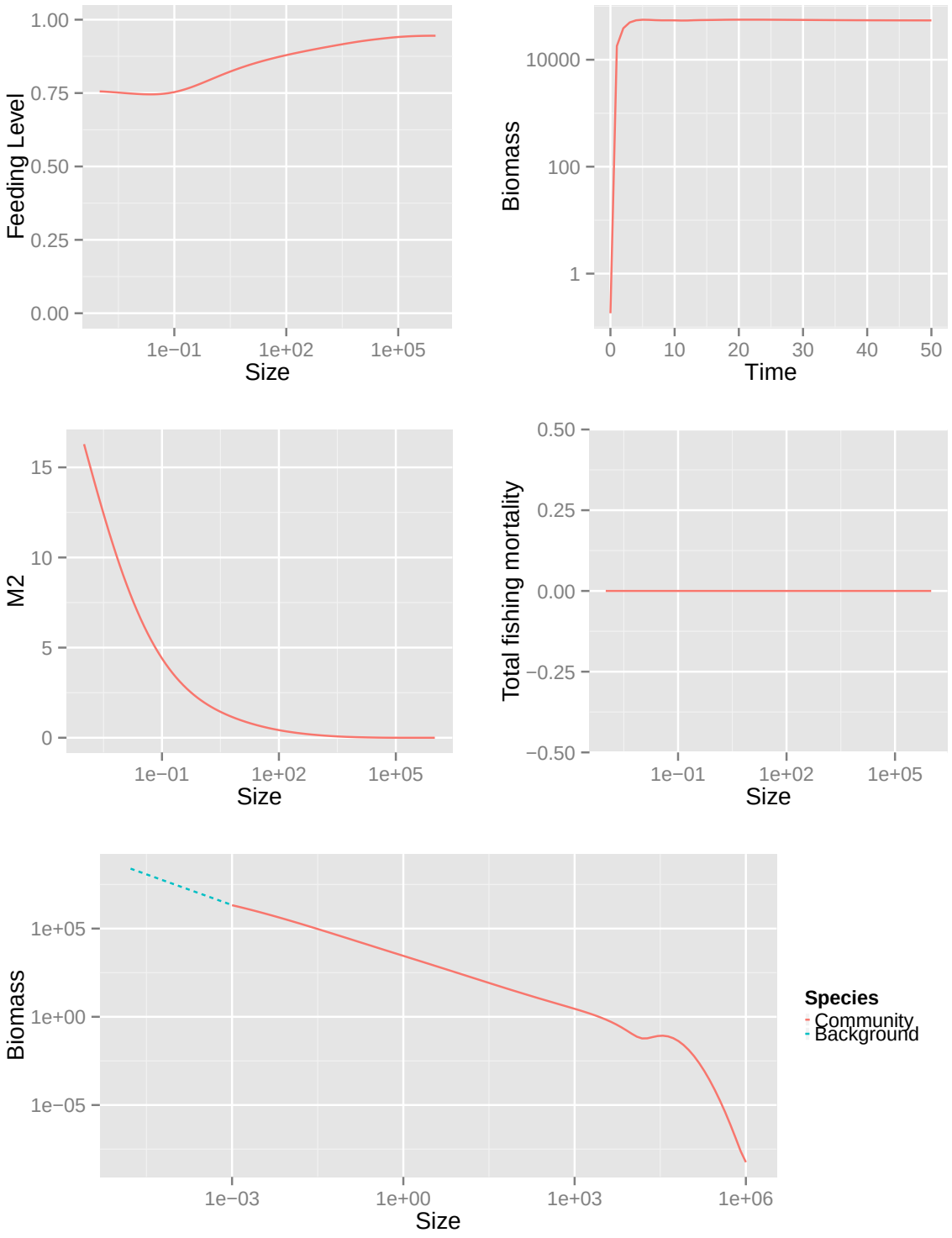

Figure 3: Example plot of the community model.

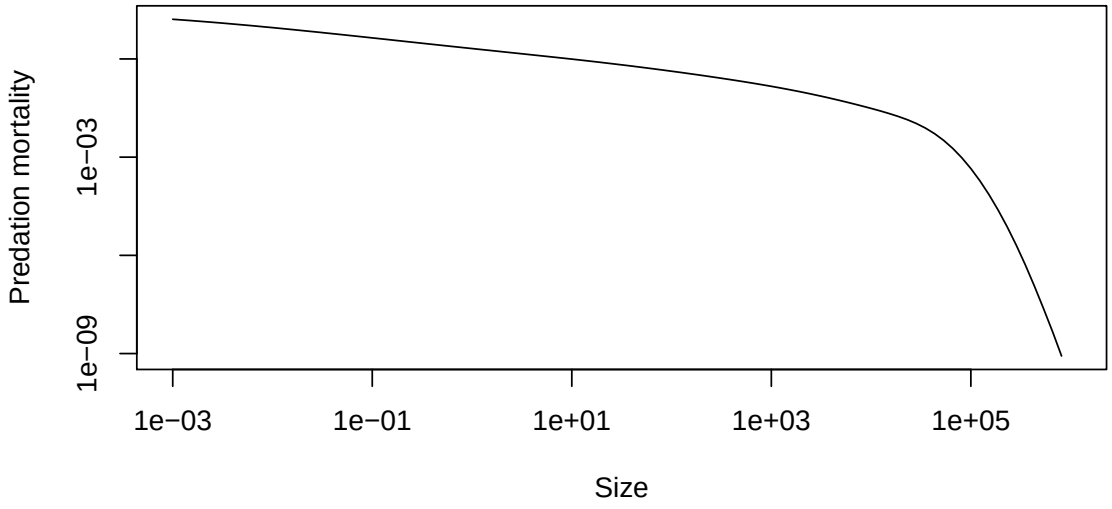

Figure 4: Predation mortality without fishing in the community model (note the log scales for both axes).

This means we need to consider how fishing is handled in `mizer`. The `set_community_model()` function automatically sets the fishing selectivity to have a knife-edge shape, with only individuals larger than 1 kg selected (the size at the knife-edge can be changed by setting the `knife_edge_size` argument). Although it is possible to change the selectivity function, here we will use the default knife-edge selectivity. Here we set up the parameter object exactly as before, but now we are explicitly setting the size at which individuals are selected by the fishing gear.

```
params_knife <- set_community_model(z0 = 0.1, recruitment = 4e7,
    alpha = 0.2, f0 = 0.7, knife_edge_size = 1000)
```

First we perform a simulation without fishing in the same way we did above by setting the `effort` argument to 0:

```
sim0 <- project(params_knife, effort = 0, t_max = 50)
```

Now we want to simulate again, this time with fishing. In the simulations, fishing mortality is calculated as the product of the fishing selectivity, effort and catchability (see Section 8.3 for more details). By default catchability is set to 1. This means that a fishing effort of 1 will result in a fishing mortality of 1 for fully selected sizes. Here we run a simulation with fishing effort set to 1 for the duration of the simulation:

```
sim1 <- project(params_knife, effort = 1, t_max = 50)
```

You can compare the difference between these scenarios by using the `plot()` method as before (Figure 5). Of particular interest is the fishing mortality at size. The knife-edge selectivity at 1000 g can be clearly seen and an effort of 1 has resulted in a fishing mortality of 1 for the fully selected sizes.

```
plot(sim1)
```

To explore the presence of a trophic cascade, we are interested in looking at the relative change in abundance when the community is fished compared to when it is not fished. To do this we need to get the abundances at size from the simulation objects. The abundances are store in the `n` slot of the *MizerSim* objects. The `n` slot contains a three dimensional array with dimensions time x species x size. Here we have 51 time steps (50 from the simulation plus one which stores the initial population), 1 species and 100 sizes:

```
dim(sim0@n)

## [1]  51   1 100
```

We want the abundances in the final time step, and we can use these to calculate the relative abundances:

```
relative_abundance <- sim1@n[51,,] / sim0@n[51,,]
```

This can then be plotted (Figure 6) using basic R plotting commands. The sizes are stored in the `params@w` slot (a slot of a slot!).

```
plot(x=sim0@params@w, y=relative_abundance, log="x", type="n",
    xlab = "Size (g)", ylab="Relative abundance")
lines(x=sim0@params@w, y=relative_abundance)
lines(x=c(min(sim0@params@w),max(sim0@params@w)), y=c(1,1),lty=2)
```

The impact of fishing on species larger than 1000g can be clearly seen. As described in (Andersen and Pedersen, 2010), the fishing pressure lowers the abundance of large fish (the decrease in relative abundance at 1000 g). This then relieves the predation pressure on their smaller prey (the preferred predator-prey size ratio is given by the β parameter, which is set to 100 by default), leading to an increase in their abundance. This in turn increases the predation mortality on their smaller prey, which reduces their abundance and so on.

5.5 The impact of changing σ

As described above, the σ parameter determines the width of the predator prey size preference. Here we take a look at how changing the value of σ can affect the dynamics of the community. In the examples above, σ is set in the `set_community_model()` function by default to a value of 2. When projected through time, the community abundances converge to a stable equilibrium. What happens if we reduce the value of σ, for example by setting it to 1.0? We can do this by passing in the new value of σ into `set_community_model()`.

```
params_sigma1 <- set_community_model(z0 = 0.1,
     f0 = 0.7, alpha = 0.2, recruitment = 4e7, sigma = 1)
```

We want to project this new model through time using the `project()` method. Note that we have introduced a new argument: *dt*. This is the step size of the solver. It does not have anything to do with the biology in the model. It only affects the internal engine of `project()` that performs

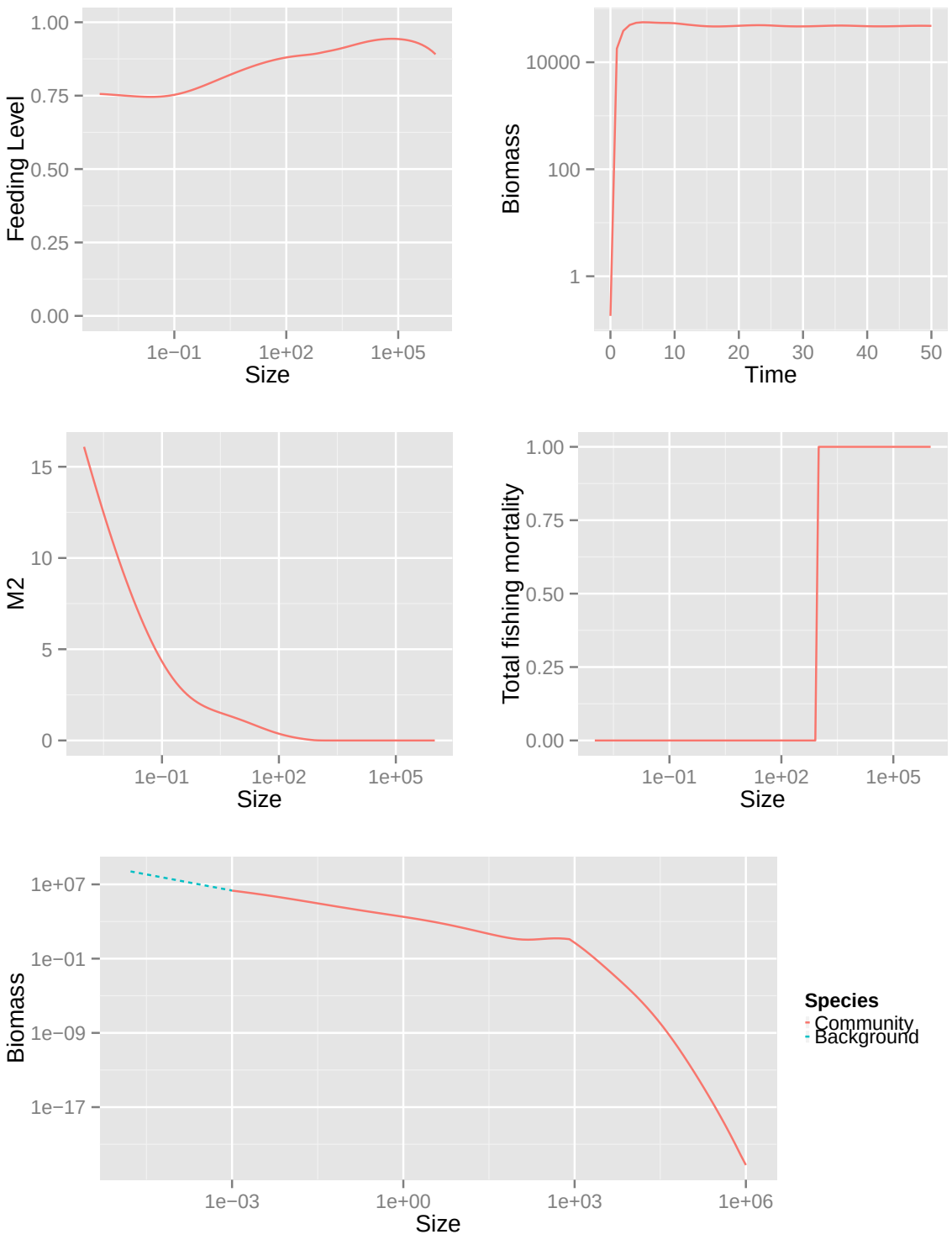

Figure 5: Summary plot for the community model when fishing with knife-edge selectivity at size = 1000 g.

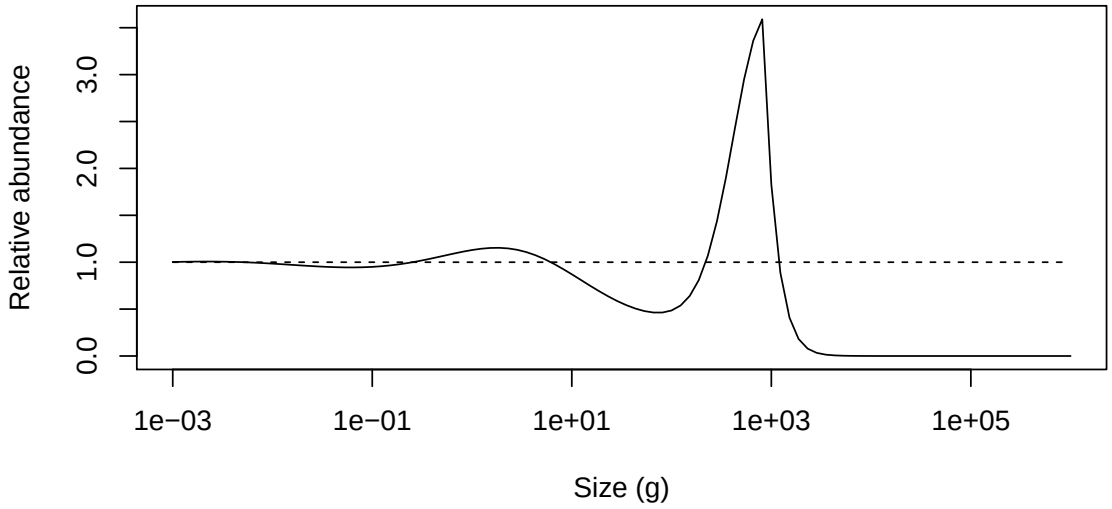

Figure 6: Relative abundances from the unfished (dashed line) and fished (solid line) community model.

the projection. As you can see in the underlying model equations in Section 3, the model runs in continuous time. Therefore, to project it forward, `project()` must solve the system of equations using numerical methods. The quality of these methods is strongly affected by dt. The default value of dt is 0.1, which will be fine for most of the projections we run in this Vignette. Here it is necessary to reduce the value to 0.01 to avoid introducing any artefacts into the projected values. Decreasing dt increases the time it takes to run a projection. Here we project the new parameters object for 50 time steps without fishing:

```
sim_sigma1 <- project(params_sigma1, effort = 0, t_max = 50, dt=0.01)
```

Let's take a look at how the abundances change through time. We can do this with the `plotBiomass()` method:

```
plotBiomass(sim_sigma1, print_it = FALSE)
```

Figure 7 shows that abundances of the community no longer converge to a stable equilibrium and the dynamics appear to be chaotic. The ecological significance of the change in dynamics, and of the ability of simple community models to show chaotic behaviour, is still being debated. It can be argued that the size of the oscillations are too large to be 'true'. Additionally, when a trait-based model (see Section 6) is implemented, the magnitude of the oscillations are much smaller.

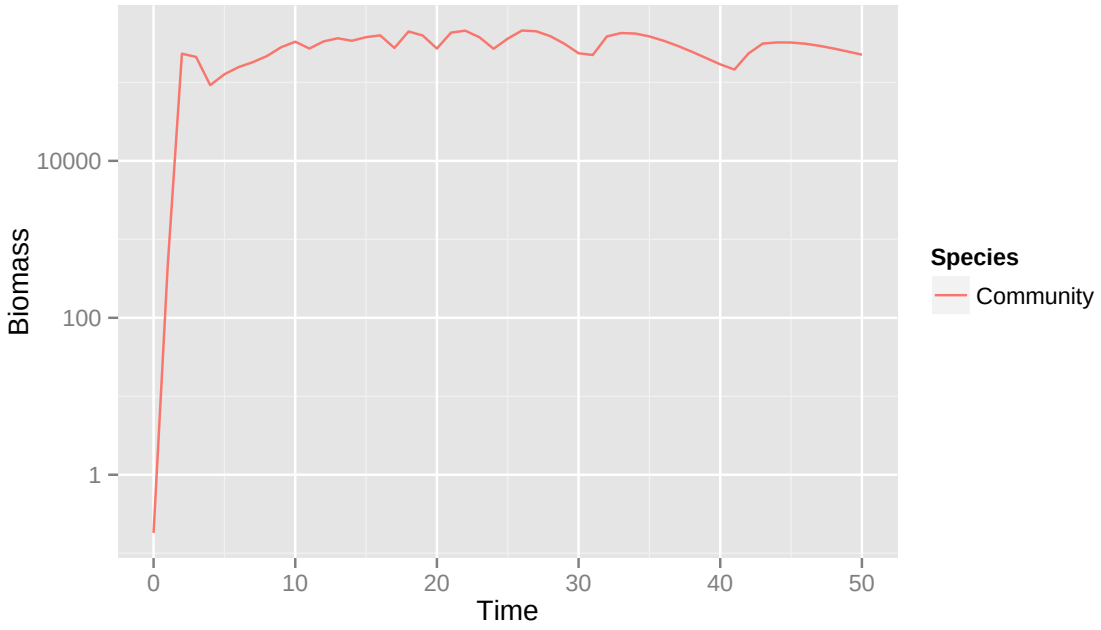

Figure 7: Biomass of the community model when σ is reduced to a value of 1.0.

6 Implementing a trait-based model

6.1 Introduction

As mentioned above,the trait-based size spectrum model can be derived as a simplification of the general model outline in Section 3. It is more complicated than a community model and the most significant difference between the two is that while the community model only resolves a single "species", the trait-based model resolves many species. In a trait-based model the asymptotic size is considered to be the most important "trait" characterizing a species. All of the species-specific parameters, such as β and σ, are the same for all species. Other model parameters are determined by the asymptotic size. For example, the weight at maturation, $w_m = \eta_m W$, where $\eta_m = 0.25$. The number of species is not important and does not affect the general dynamics of the model. The asymptotic sizes of the species are spread evenly over the size range of the community. For applications of the trait-based model see Andersen and Pedersen (2010) and Andersen and Rice (2010).

6.2 Setting up a trait-based model

To help set up a trait-based model, there is a wrapper function, `set_trait_model()`. Like the `set_community_model()` function described above, this function can take many arguments. Most of them have default values so you don't need to worry about them for the moment. See the help page for more details.

```
?set_trait_model
```

The main parameters of interest are the number of the species in the model (`no_sp`) and the minimum

and maximum asymptotic sizes (`min_w_inf` and `max_w_inf` respectively, the asymptotic sizes are spread evenly on a logarithmic scale).

One of the key differences between the community type model described above and the trait-based model is that reproduction and egg production are considered. In the community model, recruitment is constant and there is no relationship between the abundance in the community and egg production. In the trait-based model, the egg production is modeled using a "Beverton-Holt" type function (the default in `mizer`, see Section 3.6) where the recruitment flux R_i (numbers per time) approaches a maximum recruitment as the egg production increases. The maximum recruitment flux is calculated using equilibrium theory and a recruitment multiplier (see Andersen and Pedersen, 2010), κ (see Equation 3.16), which can be passed in as an argument (`k0`) to the `set_trait_model()` function. `k0` has a default value of 50.

Here we set up the model to have 10 species, with asymptotic sizes ranging from 10 g to 100 kg. All the other parameters have default values.

```
params <- set_trait_model(no_sp = 10, min_w_inf = 10, max_w_inf = 1e5)
```

This function returns an object of type *MizerParams*, which holds all the model information, including species parameters.

```
class(params)

## [1] "MizerParams"
## attr(,"package")
## [1] "mizer"
```

This object can therefore be interrogated in the same way as described in Section 5 above, either by inspecting the individual slots or by using the `summary()` function.

```
summary(params)

## An object of class "MizerParams"
## Community size spectrum:
##  minimum size: 0.001
##  maximum size: 110000
##  no. size bins: 100
## Background size spectrum:
##  minimum size: 1e-10
##  maximum size: 110000
##  no. size bins: 130
## Species details:
##    species     w_inf     w_mat beta sigma
## 1        1     10.00     2.500  100   1.3
## 2        2     27.83     6.956  100   1.3
## 3        3     77.43    19.357  100   1.3
## 4        4    215.44    53.861  100   1.3
## 5        5    599.48   149.871  100   1.3
## 6        6   1668.10   417.025  100   1.3
## 7        7   4641.59  1160.397  100   1.3
## 8        8  12915.50  3228.874  100   1.3
```

```
## 9         9  35938.14  8984.534  100   1.3
## 10       10 100000.00 25000.000  100   1.3
## Fishing gear details:
##  Gear Target species
##   knife_edge_gear   1 2 3 4 5 6 7 8 9 10
```

The summary shows us that now we have 10 species in the model, with asymptotic sizes ranging from 10 to 10^5. The size at maturity (`w_mat`) is linearly related to the asymptotic size. Each species has the same preferred predator-prey mass ratio parameter values (`beta` and `sigma`, see Equation 3.3). There are 100 size bins in the community and 130 size bins including the background resource spectrum. Ignore the summary section on fishing gear for the moment. This is explained later.

6.3 Running the trait-based model

As with the community model described above, we can project the model through time using the `project()` method. Here we project the model for 75 time steps and without any fishing (the `effort` argument is set to 0). We use the default initial population abundances given by the `get_initial_n()` function so there is no need to pass in any initial population values (see Section 9.3).

```
sim <- project(params, t_max=75, effort = 0)
```

This results in a *MizerSim* object which contains the abundances of the community and background resource spectra through time, as well as the original *MizerParams* object. As with the community model, we can get a quick overview of the results of the simulation by calling the generic `plot()` method:

```
plot(sim)
```

The summary plot has the same panels as the one generated by the community model, but here you can see that all the species in the community are plotted (Figure 8). The panels show the situation in the final time step of the simulation, apart from the biomass through time plot. As this is a trait-based model where all species fully interact with each other, the predation mortality (M2) and feeding level by size is the same for each species. The biomasses quickly settle down to equilibria. In this simulation we turned fishing off so the fishing mortality is 0. The size-spectra show the abundances at size to be evenly spaced by asymptotic size.

6.4 Example of a trophic cascade with the trait-based model

As with the community model, it is possible to use the trait-based model to simulate a trophic cascade, similar to those seen in Andersen and Pedersen (2010). Again, we perform two simulations, one with fishing and one without. We therefore need to consider how fishing gears and selectivity have been set up by the `set_trait_model()` function.

The default fishing selectivity function is a knife-edge function, which only selects individuals larger than 1000 g. There is also only one fishing gear in operation, and this selects all of the species. You can see this if you call the `summary()` method on the *params* argument we set up above. At the bottom of the summary there is a section on "Fishing gear details". You can see that there is only one gear, called "knife_edge_gear" and that it selects species 1 to 10. To control the size at which individuals are selected there is a `knife_edge_size` argument to the `set_trait_model()` function. This has a default value of 1000 g.

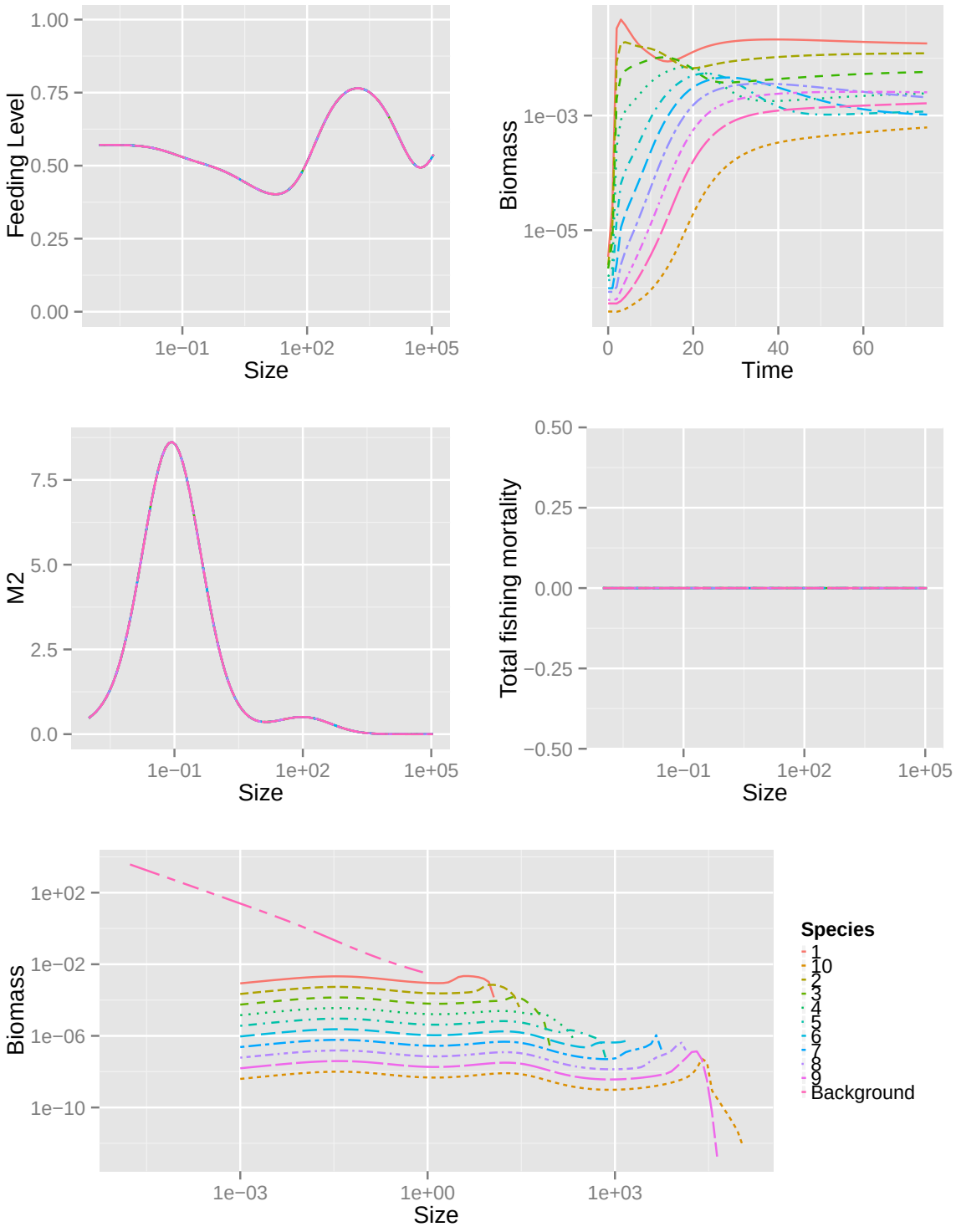

Figure 8: Example plot of the trait-based model with no fishing.

In `mizer` it is possible to include more than one fishing gear in the model and for different species to be caught by different gears. We will ignore this for now, but will explore it further below when we introduce an industrial fishery to the trait-based model.

To set up the trait-based model to have fishing we set up the *MizerParams* object in exactly the same way as we did before but here the `knife_edge_size` argument is explicitly passed in for clarity:

```
params_knife <- set_trait_model(no_sp = 10, min_w_inf = 10, max_w_inf = 1e5,
    knife_edge_size = 1000)
```

First we perform a simulation without fishing in the same way we did above by setting the `effort` argument to 0:

```
sim0 <- project(params_knife, effort = 0, t_max = 75)
```

Now we simulate with fishing. Here, we use an effort of 0.75. As mentioned in Section 5.4, the fishing mortality on a species is calculated as the product of effort, catchability and selectivity (see Section 8.3 for more details). Selectivity ranges between 0 (not selected) and 1 (fully selected). The default value of catchability is 1. Therefore, in this simulation the fishing mortality of a fully selected individual is simply equal to the effort. This effort is constant throughout the duration of the simulation (however, this is does not necessarily have to be the case, see Section9).

```
sim1 <- project(params_knife, effort = 0.75, t_max = 75)
```

Again, we can plot the summary of the fished community using the default `plot()` function (Figure 9). The knife-edge selectivity at 1000 g can be clearly seen in the fishing mortality panel:

```
plot(sim1)
```

The trophic cascade can be explored by comparing the total abundances of all species at size when the community is fished and unfished. As mentioned above, the abundances are stored in the `n` slot of the *MizerSim* object. The `n` slot returns a three dimensional array with dimensions time x species x size. Here we have 76 time steps (75 from the simulation plus one which stores the initial population), 10 species and 100 sizes:

```
dim(sim0@n)

## [1]  76  10 100
```

As with the community model, we are interested in the relative total abundances by size in the final time step so we want to pull out the 76th time step from the abundances and sum over the species. We can use the `apply()` function to help us:

```
total_abund0 <- apply(sim0@n[76,,],2,sum)
total_abund1 <- apply(sim1@n[76,,],2,sum)
```

We can then use these vectors to calculate the relative abundances:

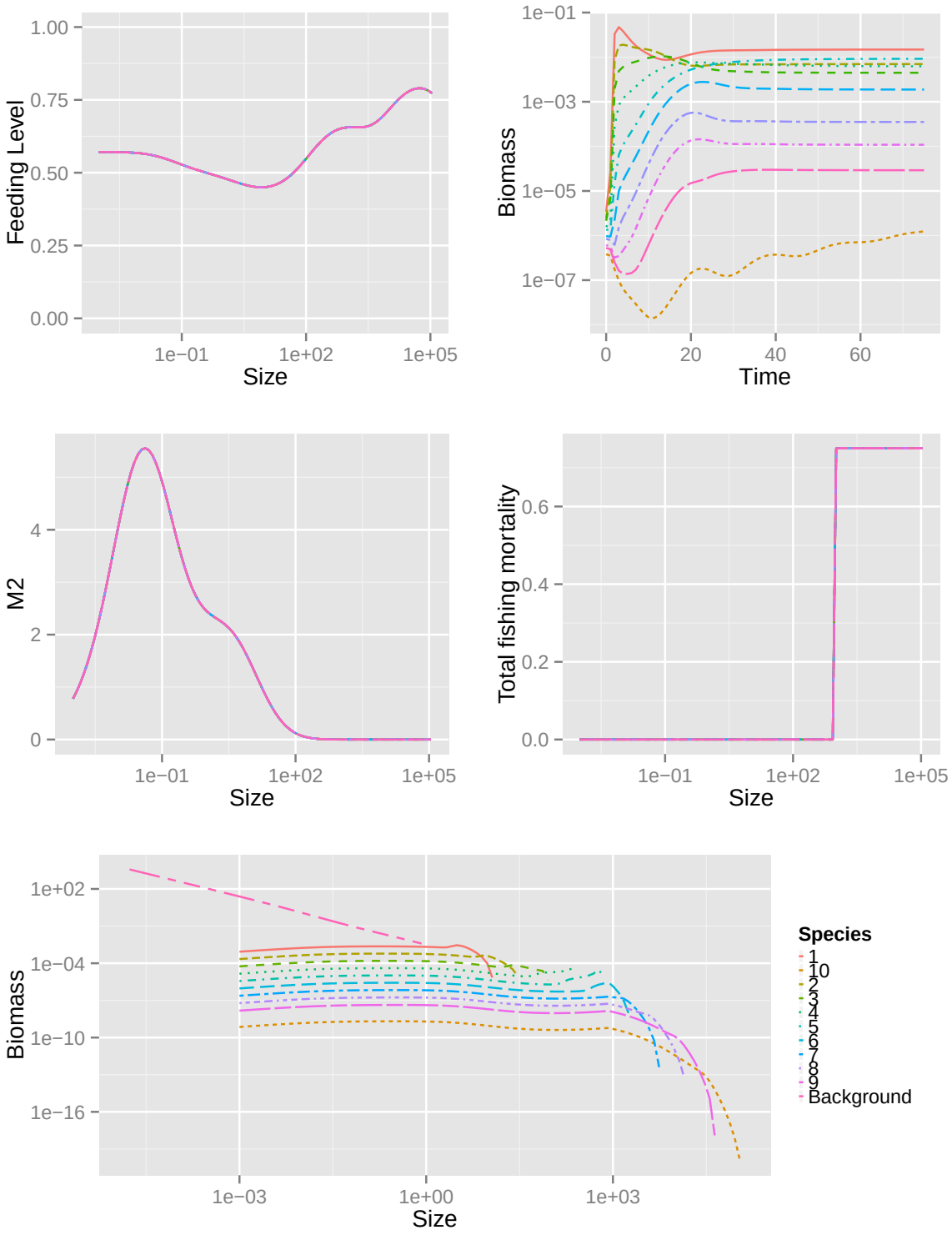

Figure 9: Summary plot for the trait-based model when fishing with knife-edge selectivity at size = 1000 g.

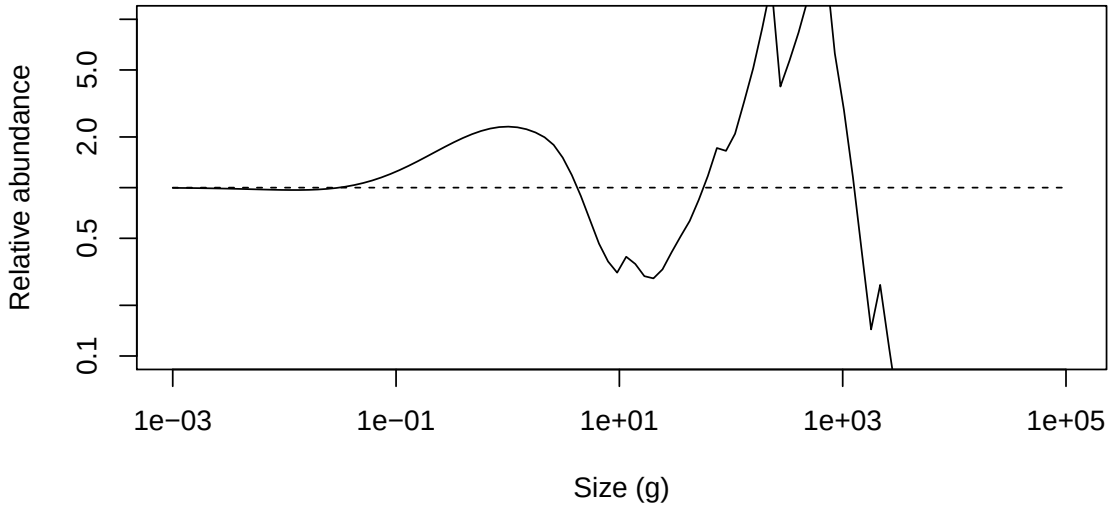

Figure 10: Relative abundances from the unfished (dashed line) and fished (solid line) trait based model.

```
relative_abundance <- total_abund1 / total_abund0
```

Which can be plotted using (Figure 10):

```
plot(x=sim0@params@w, y=relative_abundance, log="xy", type="n", xlab = "Size (g)",
    ylab="Relative abundance", ylim = c(0.1,10))
lines(x=sim0@params@w, y=relative_abundance)
lines(x=c(min(sim0@params@w),max(sim0@params@w)), y=c(1,1),lty=2)
```

The impact of fishing on species larger than 1000 g can be clearly seen. As described above and in (Andersen and Pedersen, 2010), the fishing pressure lowers the abundance of large fish ($>$ 1000 g). This then relieves the predation pressure on their smaller prey (the preferred predator-prey size ratio is given by the β parameter, which is set to 100 by default), leading to an increase in their abundance. This in turn increases the predation mortality on their smaller prey, which reduces their abundance and so on.

This impact can also be seen by looking at the predation mortality by size. The predation mortalities are retrieved using the `getM2()` method for *MizerSim* objects. This returns a three dimensional array of predation mortalities by time x species x size (see the help page for `getM2()` for more details). As mentioned above, for the trait based model the predation mortality by size is the same for each species. Therefore we only look at the predation mortality of the first species.

```
m2_no_fishing <- getM2(sim0)[76,1,]
m2_with_fishing <- getM2(sim1)[76,1,]
```

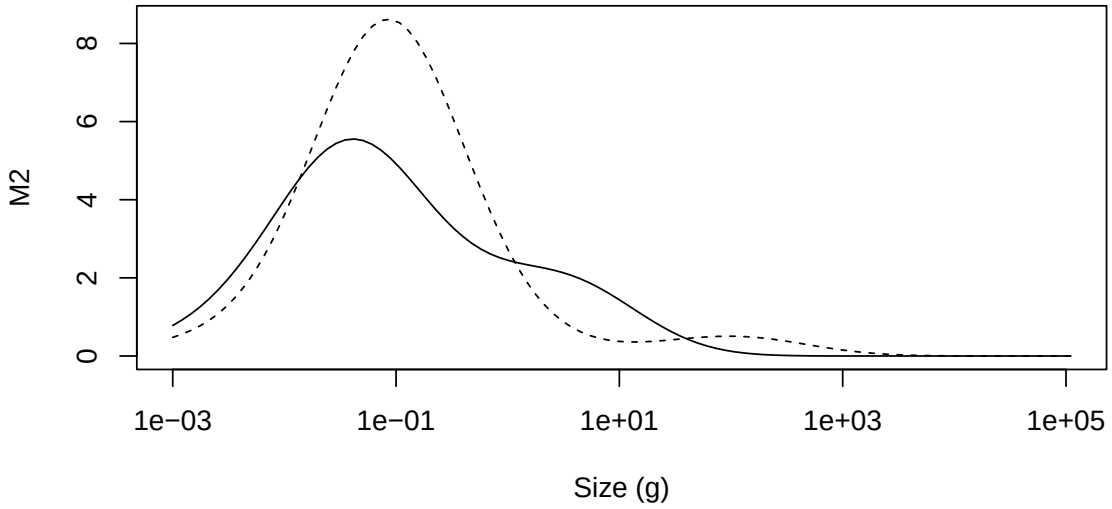

Figure 11: Predation mortalities from the unfished (dashed line) and fished (solid line) trait-based model.

The predation mortalities can then be plotted (see Figure 11).

```
plot(x = sim0@params@w, y = m2_no_fishing, log="x", type="n", xlab = "Size (g)",
    ylab = "M2")
lines(x = sim0@params@w, y = m2_no_fishing, lty=2)
lines(x = sim0@params@w, y = m2_with_fishing)
```

6.5 Setting up an industrial fishing gear

In this section we only want to operate an "industrial" fishery, like in Andersen and Pedersen (2010). Industrial fishing targets the small zooplanktivorous species that are typically used for fishmeal production.

In the previous simulations we had only one fishing gear and it targeted all the species in the community. This gear had a knife-edge selectivity that only selected species larger than 1 kg. Here we expand the model to include multiple fishing gears. This requires us to look more closely at how fishing gears are handled in `mizer`. In `mizer` it is possible for a fishing gear to catch only a subset of the species in the model. This is useful because when running a simulation with `project()` you can specify the effort per gear and so you can turn gears on or off as you want. The shape of the selectivity function of each gear will be the same for all of the species it catches but the selectivity parameter values for each species may be different. For example, we can set up an "industrial" gear to catch only a subset of species. Each species it catches will be caught with knife-edge selectivity but they may have a different knife-edge positions.

Using the trait-based model wrapper function it is only possible to have a knife-edge selectivity. Each

species in the model must be given the position of the knife edge and the name of the fishing gear. This is done using the `knife_edge_size` argument (which we have already seen) and the `gear_names` argument of the `set_trait_model()` function. In the previous examples, we passed in the `knife_edge_size` argument as a single value which was used for all the species. This effectively set up a single gear, that caught all species using the same selectivity pattern. The `gear_names` argument was not used. Now we are going to pass the `knife_edge_size` argument as a vector with the same length as the number of species in the model. The values in the vector will be the positions of the knife-edge for each species. We are also going to pass in a new argument, `gear_names` which is a vector of the names of the gears. The vector must have the same length as the number of species in the model.

We will set up the model to include two fishing gears: an "industrial" gear that only catches species with an asymptotic size less than or equal to 500g, and a second gear, "other", that catches everything else. The position of the knife-edge for both gears will occur at 0.05 x the asymptotic size i.e. the selectivity parameters will be different for each species and will depend on the asymptotic size.

To start with we need to know what the asymptotic sizes of the species in the model are so we can determine the knife-edge positions for each species. As mentioned above, the `set_trait_model()` function spaces the asymptotic sizes equally on a logarithmic scale. This means we can calculate them by hand.

```
no_sp <- 10
min_w_inf <- 10
max_w_inf <- 1e5
w_inf <- 10^seq(from=log10(min_w_inf), to = log10(max_w_inf), length=no_sp)
```

We can then use these asymptotic sizes to set a vector of knife-edges that are 0.05 times the asymptotic size:

```
knife_edges <- w_inf * 0.05
```

Now we want to assign each species to either the "industrial" or "other" gear, We want to create a vector of gear names. This vector must be the same length as the number of species in the model.

```
other_gears <- w_inf > 500
gear_names <- rep("Industrial", no_sp)
gear_names[other_gears] <- "Other"
```

Finally, we can create our *MizerParams* object by passing in the `knife_edge_sizes` and `gear_names` argument. All the other arguments are the same as before:

```
params_multi_gear <- set_trait_model(no_sp = no_sp, min_w_inf = min_w_inf,
    max_w_inf = max_w_inf, knife_edge_size = knife_edges, gear_names = gear_names)
```

To check what has just happened we can take a look inside the *MizerParams* object. There is a slot in the object called `species_params`. This is a *data.frame* that contains the life-history parameters of the species in the model (the results of running this command are not shown):

```
params_multi_gear@species_params
```

This *data.frame* is pretty interesting as it allows you to investigate the parameters for each species. For example, you can see that the predator-prey mass ratio parameters, `beta` and `sigma`, are indeed

the same for each species. The columns of the `species_params` *data.frame* that we are interested in here are `species` (the name of the species, here just numerical identifiers), `w_inf` (the asymptotic size), `sel_func` (the name of the selectivity function for that species), `knife_edge_size` (the position of the knife-edge of the selectivity) and `gear` (the name of the fishing gear). You can see that two gears have been set up, "Industrial" and "Other" and that they catch different species depending on their asymptotic size.

Having created our *MizerParams* object with multiple gears, we can now turn our attention to running a projection with multiple gears. In our previous examples of calling `project()` we have specified the fishing effort with the `effort` argument using a single value. This fixes the fishing effort for all gears in the model, for all time steps. We can do this with our multi-gear parameter object:

```
sim_multi_gear <- project(params_multi_gear, t_max = 75, effort = 0.5)
```

By plotting this you can see that the fishing mortality for each species now has a different selectivity pattern (Figure 12), and that the position of the selectivity knife-edge is given by the asymptotic size of the species.

```
plot(sim_multi_gear)
```

For the industrial fishery we said that we only wanted species with an asymptotic size of 500 g or less to be fished. There are several ways of specifying the `effort` argument for `project()` . Above we specified a single value that was used for all gears, for all time steps. It is also possible to specify a separate effort for each gear that will be used for all time steps. To do this we pass in effort as a named vector. Here we set the effort for the "Industrial" gear to 0.75, and the effort of the "Other" gear to 0 (effectively turning it off).

```
sim_multi_gear <- project(params_multi_gear, t_max = 75,
    effort = c(Industrial = 0.75, Other = 0))
```

Now you can see that the "Industrial" gear has been operating and that fishing mortality for species larger than 500 g is 0 (Figure 13).

```
plot(sim_multi_gear)
```

6.6 The impact of industrial fishing

In the previous section we set up and ran a model in which an industrial fishery was operating that only selected smaller species. We can now answer the question: what is the impact of such a fishery? We can again compare abundances of the fished (*sim_industrial*1) and unfished (*sim_industrial*0) cases:

```
sim_industrial0 <- project(params_multi_gear, t_max = 75, effort = 0)
sim_industrial1 <- project(params_multi_gear, t_max = 75,
    effort = c(Industrial = 0.75, Other = 0))
total_abund0 <- apply(sim_industrial0@n[76,,],2,sum)
total_abund1 <- apply(sim_industrial1@n[76,,],2,sum)
relative_abundance <- total_abund1 / total_abund0
```

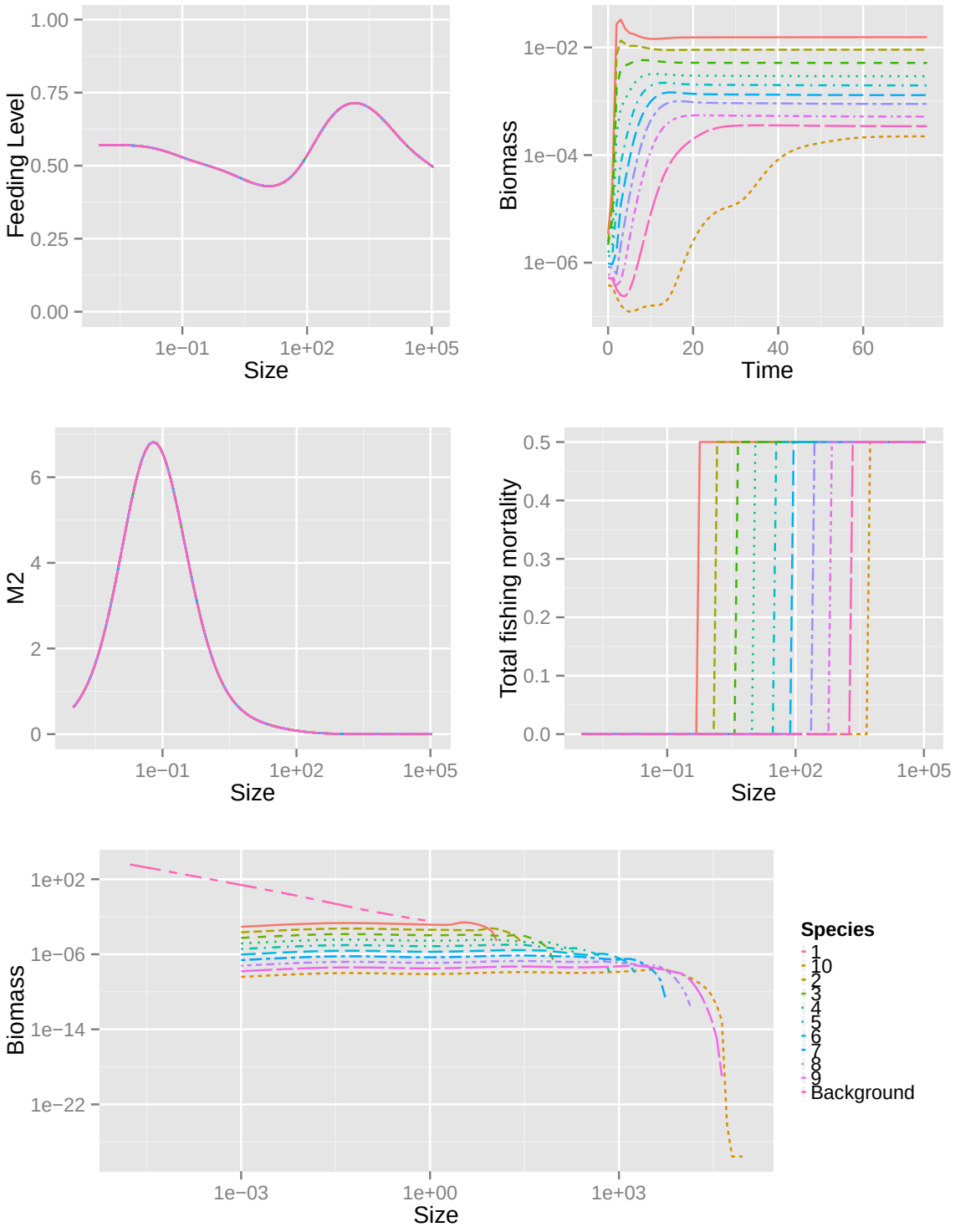

Figure 12: Summary plot for the trait-based model with multiple gears when all gears are operational.

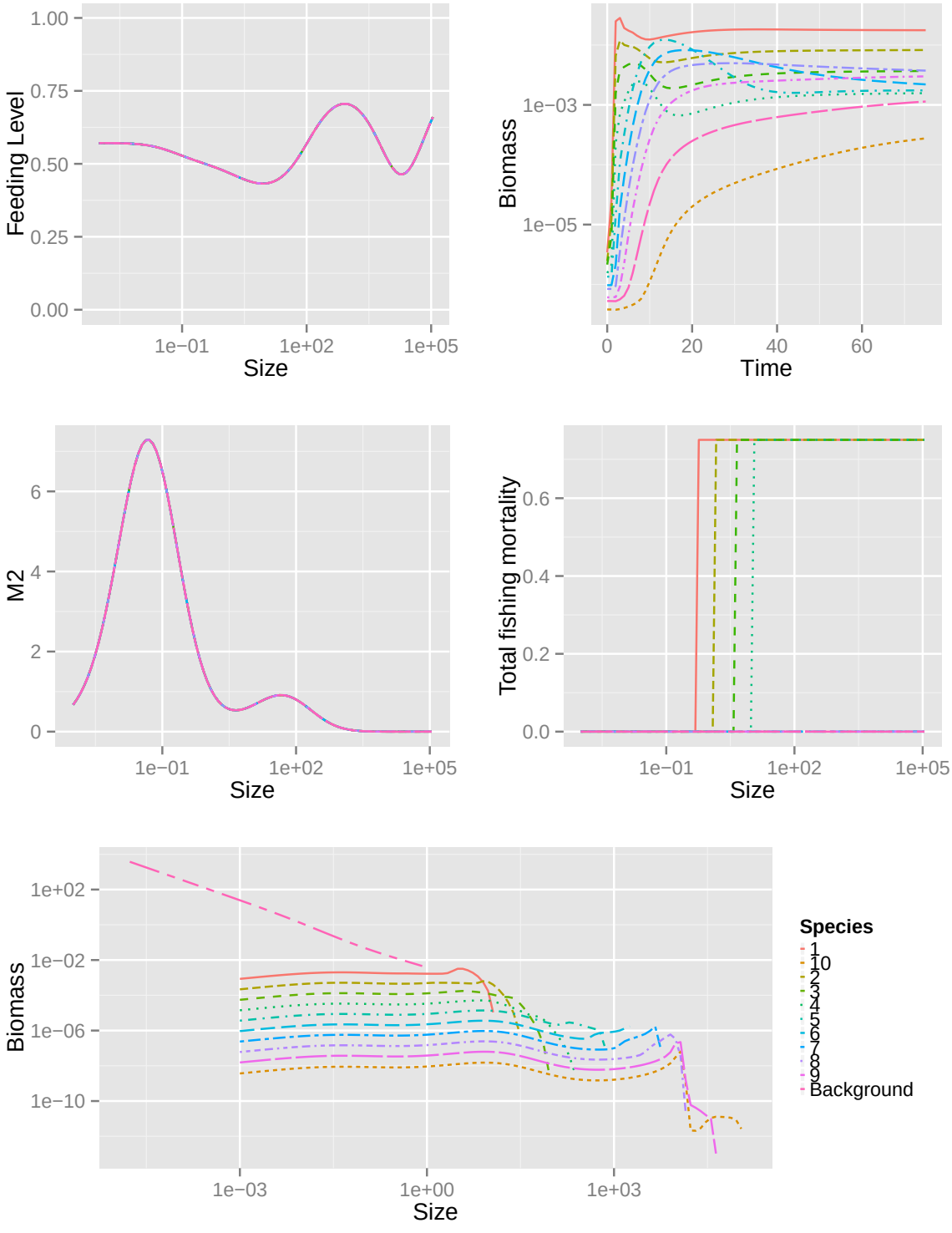

Figure 13: Summary plot for the trait-based model with multiple gears when only the industrial gear that fishes on species with asymptotic size of 500 g or less is operational.

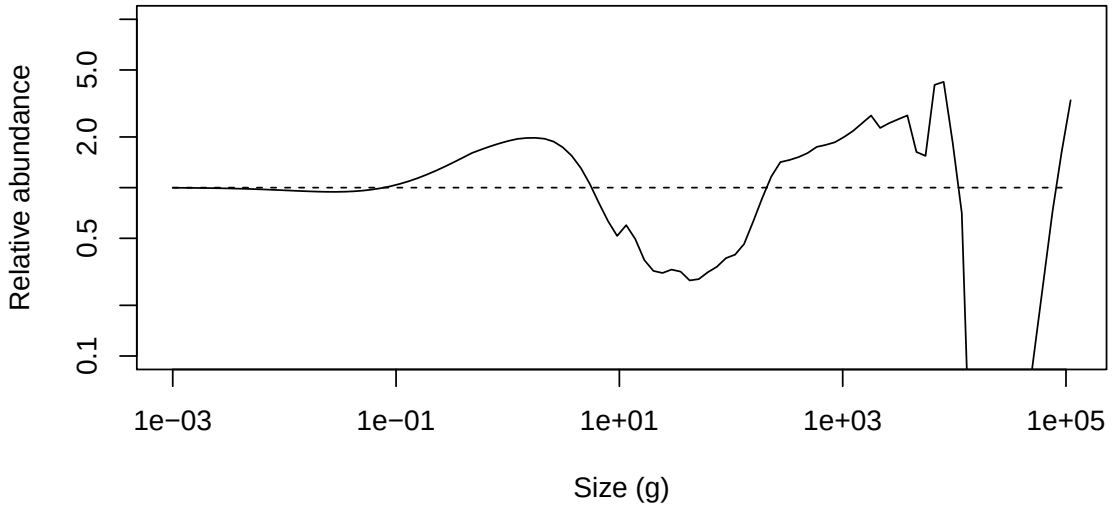

Figure 14: Relative abundances from the unfished (dashed line) and fished (solid line) trait based model with an industrial fishery that targets species with an asymptotic size of 500 g or less.

And plot the relative abundances:

```
plot(x=sim0@params@w, y=relative_abundance, log="xy", type="n", xlab = "Size (g)",
    ylab="Relative abundance", ylim = c(0.1,10))
lines(x=sim0@params@w, y=relative_abundance)
lines(x=c(min(sim0@params@w),max(sim0@params@w)), y=c(1,1),lty=2)
```

This shows another trophic cascade (Figure 14), although this time one driven by fishing the species at the midrange part of the spectrum, not the largest individuals as before. This trophic cascade acts in both directions. The cascade upwards is driven by the lack of food for predators leading to smaller realised maximum sizes. The cascade downwards has the same mechanism as fishing on large fish, a combination of predation mortality and food limitation.

7 Introducing multispecies models

The previous sections have used wrapper functions to set up *MizerParams* objects that are appropriate for community and trait-based simulations. We now turn our attention to multispecies, or species-specific, models. These are potentially more complicated than the community and trait-based models and use the full power of the `mizer` package.

In multispecies type models multiple species are resolved. However, unlike in the trait-based model which also resolves multiple species, explicit species are represented. There are several advantages to this approach. As well as investigating the community as a whole (as was done for the community and trait-based models), we are able to investigate the dynamics of individual species. This means

that species specific management rules can be tested and species specific metrics, such as yield, can be compared to reference levels.

A multispecies model can take more effort to set up. For example, each species will have different life-history parameters; there may be multiple gear types with different selectivities targeting different groups of species; the fishing effort of each gear may change with time instead of just being constant (which has been the case in the simulations we have looked at so far); the interactions between the species needs to be considered.

For the remainder of this vignette we build up a multispecies model for the North Sea. To effectively use `mizer` for a multispecies model we are going to have to take a closer look at the *MizerParams* class and the `project()` method. This will all be done in the context of examples so hopefully everything will be clear.

We also take a closer look at some of the summary plots and analyses that can be performed, for example, calculating a range of size-based indicators.

8 Setting up a multispecies model

8.1 Overview

The *MizerParams* class is used for storing model parameters. We have already met the *MizerParams* class when we looked at community and trait-based models. However, direct handling of the class was largely hidden by the use of the wrapper functions. To set up a multispecies model we need to directly create and use the *MizerParams* class. This is probably the most complicated part of using the `mizer` package so we will take it slowly. For additional help you can look at the help page for the class by entering `class ? MizerParams`.

The *MizerParams* class stores the:

- life-history parameters of the species in the community, such as W_∞;
- size-based biological parameters for the species, such as the search volume, $V(w)$;
- stock-recruitment relationship functions and parameters of each species;
- interaction matrix to describe the spatial overlap of pairs of species;
- parameters relating to the growth and dynamics of the background resource spectrum;
- fishing gear parameters: selectivity and catchability.

Note that the *MizerParams* class does not store any parameters that can vary through time, such as fishing effort or population abundance. These are stored in the *MizerSim* class which we will come to later in Section 9.

Although the *MizerParams* class seems complicated, it is relatively straightforward to set up and use. Objects of class *MizerParams* are created using the constructor method `MizerParams()`. This constructor method can take many arguments. However, creation is simplified because many of the arguments have default values.

In the rest of this section we look at the main arguments to the `MizerParams()` constructor method. To help understand how the constructor is used and how the *MizerParams* class relates to the equations given in Section 3, there is an example section where we create example parameter objects using data that comes with the `mizer` package.

8.2 The species parameters

Although many of the arguments used when creating a *MizerParams* object are optional, there is one argument that must be supplied by the user: the *species specific parameters*. These are stored in a single *data.frame* object. The *data.frame* is arranged species by parameter, so each column is a parameter and each row has the parameters for one of the species in the model. Although it is possible to create the data.frame by hand in R , it is probably easier to create the data externally as a .csv file (perhaps using a suitable open source spreadsheet such as LibreOffice) and then read the data into R .

For each species in the model community there are certain parameters that are essential and that do not have default values. The user must provide values for these parameters. There are also some essential parameters that have default values, such as the selectivity function parameters, and some that are calculated internally using default relationships if not explicitly provided. These defaults are used if the parameters are not found in the data.frame. A description of the columns of the species parameter data.frame and any default values can be seen Table 2.

The essential columns of the species parameters data.frame that have no default values are: `species`, the names of the species in the community; `w_inf`, the asymptotic mass of the species; `w_mat`, the mass at maturation; `beta` and `sigma`, the predator-prey mass ratio parameters β and σ (see Equation 3.3); stock-recruitment parameters (by default, the stock-recruitment function is a Beverton-Holt type and so, unless a different SRR function is used, a `r_max` column must be provided - see Section 8.4 for more details).

Essential columns that have default values are: `k`, the activity coefficient (default = 0); `alpha`, the assimilation efficiency (default = 0.6); `erepro`, the reproductive efficiency (default value = 1); `w_min`, the size of recruits (default value is the smallest size of the community size spectrum); `sel_func`, the name of the fishing selectivity function (default value = "knife_edge"); `gear`, the name of the fishing gear that catches the species (default value is the name of the species in the `species` column); `catchability`, the catchability of the fishing gear on that species (default value = 1). Additionally, columns that contain the selectivity function parameters are needed. As mentioned, the default selectivity function is a "knife_edge" function. This has an argument `knife_edge_size` that determines the knife-edge position and has a default value of `w_min`. If any of these columns are not included in the species parameter data.frame, the default values are used.

As mentioned above, there are some columns that are essential but if they are not provided, values for them are estimated using the values in the other columns. These columns are: `h`, the maximum food intake; `gamma`, the volumetric search rate; `ks`, the coefficient for standard metabolism; `z0`, the mortality from other sources ($\mu_{b,i}$ in Equation 3.14). These parameters can be included as columns in the species parameters *data.frame* if they are available. If they are not provided then the `MizerParams()` construction method will try to calculate them.

The `h` column is calculated as:

$$h = \frac{3k_{vb}}{\alpha f_0} W_\infty^{1/3} \tag{8.1}$$

where k_{vb} is the von Bertalanffy K parameter and f_0 is the feeding level of small individuals feeding mainly on the background resource. This mean that if an `h` column is not included in the species parameter data.frame, a column for `k_vb` is necessary. If it is not included then the `MizerParams()` method will fail with an error message. The calculation also requires a value of the feeding level of small individuals. This can be passed as an additional argument, `f0`, to the `MizerParams()` constructor and it has a default value of 0.6.

The `gamma` column is calculated using Equation 3.16. This calculation requires that the `h` column is available, either included in the species parameter data.frame, or calculated internally using the `k_vb`

column as described above. This means that if you include a `k_vb` column in the data.frame, `h` and `gamma` will be calculated from it.

The `z0` column (mortality from other sources) is calculated using Equation 3.14. `z0pre` (equivalent to μ_0) and `z0exp` (the power that W_∞ is raised to) can be passed as arguments to the `MizerParams()` constructor and have default values of 0.6 and `n`-1 (`n` has a default value of 2/3 - see below) respectively.

The `ks` column is calculated as 20% of `h`, i.e. the standard metabolism coefficient is 20% of the maximum consumption. Standard metabolism is used in calculation of the growth of individuals as $ks * w_i^p$ (see Section 3.4).

You can see in Table 2 that most of the species specific parameters relate to the life history of the species. The others relate to the gear selectivity function and the stock-recruitment relationship. These are explained further in Sections 8.3 and 8.4 respectively.

8.3 Fishing gears and selectivity

In this section we take a look at how fishing is implemented and how fishing gears are set up within `mizer`.

In `mizer`, fishing mortality is imposed on species by fishing gears. The fishing mortality F imposed by gear g on species s at size w is calculated as:

$$F_{s,g,w} = S_{s,g,w} Q_{s,g} E_g \tag{8.2}$$

where S is the selectivity by species, gear and size, Q is the catchability by species and gear and E is the fishing effort by gear. The selectivity at size has a range between 0 (not selected at that size) to 1 (fully selected at that size). Catchability is used as an additional scalar to make the link between gear selectivity, fishing effort and fishing mortality. For example, it can be set so that an effort of 1 gives a desired fishing mortality. In this way effort can then be specified relative to a 'base effort', e.g. the effort in a particular year. Fishing effort is not stored in the *MizerParams* object. Instead, effort is set when the simulation is run and can vary through time (see Section 9).

At the moment a species can only be selected by one fishing gear, although each gear can select more than one species (this is a limitation with the current package that will be developed in future releases).

The selectivity at size of each gear is given by a selectivity function. Some selectivity functions are included in the package. New functions can be defined by the user. Each gear has the same selectivity function for all the species it selects, but the parameter values for each species may be different, e.g. the lengths of species that a gear selects may be different.

The name of the selectivity function is given by the *sel_func* column in the species parameters data.frame. Each selectivity function has a range of arguments. Values for these arguments must be included as columns in the species parameters data.frame. The names of the columns must exactly match the names of the arguments. For example, the default selectivity function is `knife_edge` which has sudden change of selectivity from 0 to 1 at a certain size. The arguments for this selectivity function can be seen in the help page for this function. To see them enter:

```
?knife_edge
```

It can be seen that the `knife_edge()` function has arguments `w` and `knife_edge_size` The first argument, `w`, is size (the function calculates selectivity at size). All selectivity functions must have `w` as the first argument. The values for the other arguments must be found in the species parameters data.frame. So for the `knife_edge()` function there should be a `knife_edge_size` column (but note that because `knife_edge()` is the default selectivity function, the `knife_edge_size` argument actually

has a default value = `w_mat`). This can be seen in the example in Section 8.7. If the columns of the selectivity function arguments are not in the species parameter data.frame, an error is thrown when the *MizerParams* object is created.

Users are able to write their own size based selectivity function. The first argument to the function must be `w` and the function must return a vector of the selectivity (between 0 and 1) at size.

The name of the fishing gear is given in the `gear` column of the species parameter data.frame. If the `gear` column is not specified, the default gear name is simply the name of the species. This implies that each species is fished by a different gear. This approach can be used to explore the impacts of changing fishing mortality on individual species.

8.4 The stock-recruitment relationship

In size spectrum modelling recruitment refers to the flux of individuals that enter the size-spectrum at the smallest size group of that species (given by the parameter `w_min` in the species parameter data.frame). As can be seen in Section 3, calculating the recruitment flux involves calculating the "density independent" recruitment, $R_{p.i}$ (see Section 3.5). The $R_{p.i}$ is then modified by a stock-recruitment relationship (SRR) to impose some form of density-dependence. This then results in the density-dependent recruitment, R_i (see Section 3.6). Without this density dependence, the realised recruitment flux to the smallest size class is determined only by $R_{p.i}$. The default SRR is a Beverton-Holt type function (see Equation 3.11).

Similar to the fishing selectivity functions, any parameter used in the stock-recruitment function, other than $R_{p.i}$, must be in the species parameter data.frame and the column must have the same name as the function argument. For example, the default stock-recruitment function has a second argument to it, `r_max`. Therefore the species parameter data.frame must have an `r_max` column.

Users are able to write their own stock-recruitment function. The first argument to the function must be `rdi`, which is the density independent recruitment, $R_{p.i}$.

8.5 The interaction matrix

The interaction matrix describes the interaction of each pair of species in the model. This can be viewed as a proxy for spatial interaction e.g. to model predator-prey interaction that is not size based. The values in the interaction matrix are used to scale the encountered food and predation mortality (see Section 3.2). The matrix is square with every element being the interaction between a pair of species. The dimensions, nrows and ncolumns, therefore equal the number of species. The values are between 0 (species do not overlap and therefore do not interact with each other) to 1 (species overlap perfectly). If all the values in the interaction matrix are set to 1 then predator-prey interactions are determined entirely by size-preference.

The interaction matrix must be of type *array* or *matrix*. One way of creating your own is to enter the data using a spreadsheet (such as LibreOffice) and saving it as a .csv file. The data can be read into R using the command `read.csv()`. This reads in the data as a data.frame. We then need to convert this to a matrix using the `as()` function. An example of how to do this is given in Section 8.7.

It should be noted that the order of species in the interaction matrix has to be the same as the order in the species parameters data.frame. Although you can specify the dimnames of the interaction matrix, these names are overwritten by the species names from the species parameters data.frame inside the *MizerParams* constructor.

If an interaction matrix is not specified to the `MizerParams()` constructor the default iteration matrix is used. This has all values set to 1.

8.6 The other `MizerParams()` arguments

As well as the essential species parameters data.frame and the interaction matrix, there are several other arguments to the `MizerParams` constructor. These have default values. The arguments can be seen in Table 3.

Some of these parameters may be used to calculate the species specific parameters if they are not provided. For example, if there is no `gamma` column in the species parameter data.frame, then it is calculated using `kappa` and `f0`. This means that depending on which columns have been provided in the species parameters data.frame, some of the parameters in Table 3 may not be used. For example, if the column `z0` (the mortality from other sources) has been included, then the arguments `z0pre` and `z0exp` are not used.

Determining a value for the `kappa` argument can be difficult and may need to be estimated through some kind calibration process. The default value `kappa` is for the North Sea model.

8.7 Examples of making a *MizerParams* objects

As mentioned in the preceding sections, an object of *MizerParams* is created by using the `MizerParams()` constructor method. You can see the help page for the constructor:

```
help(MizerParams)
```

This shows that the constructor takes the following arguments:

object The species parameter data.frame (see Section 8.2). This is compulsory with no default value.

inter The interaction matrix (see Section 8.5). The default is a matrix of 1s.

... Other model parameters (see Section 8.6).

In the rest of this section we demonstrate how to pull these elements together to make *MizerParams* objects.

The first step is to prepare the species specific parameter data.frame. As mentioned above, one way of doing this is to use a spreadsheet and save it as a .csv file. We will use this approach here. An example .csv file has been included in the package. This contains the species parameters for a multispecies North Sea model (Blanchard et al., 2013). This file is placed in the `doc` folder of the package installation. The location of the file can be found by running:

```
system.file("doc/NS_species_params.csv",package="mizer")
```

This file can be opened with most spreadsheets or a text editor for you to inspect. This can be loaded into R using the following code (after you have told R to look in the right directory):

```
params_data <- read.csv("NS_species_params.csv")
```

This reads the .csv file into R in the form of a data.frame. You can check this with the `class`:

```
class(params_data)

## [1] "data.frame"
```

The example data.frame can be inspected by entering the name of the object.

```
params_data
```

```
##    species w_inf w_mat   beta sigma    r_max  k_vb
## 1    Sprat    33    13  51076   0.8 7.38e+11 0.681
## 2  Sandeel    36     4 398849   1.9 4.10e+11 1.000
## 3   N.pout   100    23     22   1.5 1.05e+13 0.849
## 4  Herring   334    99 280540   3.2 1.11e+12 0.606
## 5      Dab   324    21    191   1.9 1.12e+10 0.536
## 6  Whiting  1192    75     22   1.5 5.48e+11 0.323
## 7     Sole   866    78    381   1.9 3.87e+10 0.284
## 8  Gurnard   668    39    283   1.8 1.65e+12 0.266
## 9   Plaice  2976   105    113   1.6 4.08e+14 0.122
## 10 Haddock  3485   165    558   2.1 1.84e+12 0.271
## 11     Cod 40044  1606     66   1.3 8.26e+09 0.216
## 12  Saithe 16856  1076     40   1.1 1.12e+11 0.175
```

You can see that there are 12 species and 7 columns of parameters: `species`, `w_inf`,`w_mat`,`beta`,`sigma`,`r_max` and `k_vb`.

Of these parameters, `species`, `w_inf`, `w_mat`, `beta` and `sigma` are essential and have no default values (as described in Section 8.2). `r_max` is a SRR parameter. We are going to use the default Beverton-Holt type SRR which has `r_max` as an argument (see Section 8.4), making this column also essential. The final column, `k_vb`, will be used to calculate values for `h` and then `gamma`. This column is only essential here because the `h` and `gamma` are not included in the data.frame. It would also have been possible to include `h` and `gamma` columns in the data.frame and not include the `k_vb` column.

The values of the non-essential species specific parameters `alpha`, `k`, `ks`, `z0`, `w_min` and `erepro` are not included in the data.frame. This means that the default values will be automatically used when we create the *MizerParams* object.

Note that there are no columns describing the fishing selectivity. There is no `sel_func` column to determine the selectivity function. This means that the default selectivity function, `knife_edge`, will be used. As mentioned in Section 8.3, this function also needs another argument, `knife_edge_size`. This is not present in the data.frame and so it will be set to the default value of `w_mat`. Also, there is no `catchability` column so a default value for `catchability` of 1 will be used for all gears and species.

This species parameter data.frame is the minimum we need to create a *MizerParams* object as it contains only essential columns. To create the *MizerParams* object we pass the data.frame into the `MizerParams()` constructor method:

```
params <- MizerParams(params_data)
```

We have just created a *MizerParams* object:

```
class(params)
```

```
## [1] "MizerParams"
## attr(,"package")
## [1] "mizer"
```

As has been mentioned in Sections 5 and 6, a *MizerParams* object is made up of "slots" that store a wide range of model parameters. Each of these slots contains information on the parameters in the model. A description of these slots can be found by calling `help()` on the class: `help("MizerParams-class")`. The different slots can be accessed using the `@` operator.

The slot `species_params` contains the species parameters data.frame that was passed in to the constructor. We can inspect this slot with:

```
params@species_params
```

```
##    species w_inf w_mat   beta sigma    r_max  k_vb    gear k alpha erepro
## 1    Sprat    33    13  51076   0.8 7.38e+11 0.681   Sprat 0   0.6      1
## 2  Sandeel    36     4 398849   1.9 4.10e+11 1.000 Sandeel 0   0.6      1
## 3   N.pout   100    23     22   1.5 1.05e+13 0.849  N.pout 0   0.6      1
## 4  Herring   334    99 280540   3.2 1.11e+12 0.606 Herring 0   0.6      1
## 5      Dab   324    21    191   1.9 1.12e+10 0.536     Dab 0   0.6      1
## 6  Whiting  1192    75     22   1.5 5.48e+11 0.323 Whiting 0   0.6      1
## 7     Sole   866    78    381   1.9 3.87e+10 0.284    Sole 0   0.6      1
## 8  Gurnard   668    39    283   1.8 1.65e+12 0.266 Gurnard 0   0.6      1
## 9   Plaice  2976   105    113   1.6 4.08e+14 0.122  Plaice 0   0.6      1
## 10 Haddock  3485   165    558   2.1 1.84e+12 0.271 Haddock 0   0.6      1
## 11     Cod 40044  1606     66   1.3 8.26e+09 0.216     Cod 0   0.6      1
## 12  Saithe 16856  1076     40   1.1 1.12e+11 0.175  Saithe 0   0.6      1
##      sel_func knife_edge_size     h     gamma      z0     ks w_min
## 1  knife_edge             13 18.20 3.190e-11 0.18706  3.641 0.001
## 2  knife_edge              4 27.52 1.504e-11 0.18171  5.503 0.001
## 3  knife_edge             23 32.84 8.504e-11 0.12927  6.568 0.001
## 4  knife_edge             99 35.04 1.123e-11 0.08648  7.008 0.001
## 5  knife_edge             21 30.68 4.645e-11 0.08736  6.136 0.001
## 6  knife_edge             75 28.54 7.391e-11 0.05659  5.708 0.001
## 7  knife_edge             78 22.56 3.115e-11 0.06295  4.512 0.001
## 8  knife_edge             39 19.38 2.949e-11 0.06864  3.875 0.001
## 9  knife_edge            105 14.62 2.846e-11 0.04171  2.925 0.001
## 10 knife_edge            165 34.24 4.037e-11 0.03957  6.848 0.001
## 11 knife_edge           1606 61.58 1.597e-10 0.01754 12.316 0.001
## 12 knife_edge           1076 37.39 1.231e-10 0.02340  7.478 0.001
##    w_min_idx
## 1          1
## 2          1
## 3          1
## 4          1
## 5          1
## 6          1
## 7          1
## 8          1
## 9          1
## 10         1
## 11         1
## 12         1
```

We can see that this contains the original species data.frame (with `w_inf` and so on), plus any default

values that may not have been included in the original data.frame. For example, we can see that there are no columns for `alpha` and `h` and `gamma` etc.

Also note how the default fishing gears have been set up. Because we did not specify gear names in the original species parameter data.frame, each species is fished by a unique gear named after the species. This can be seen in the new `gear` column which holds the names of the fishing gears. Also, the selectivity function for each fishing gear has been set in the `sel_func` column to the default function, `knife_edge()`. A `catchability` column has been added with a default value of 1 for each of the species that the gear catches. An example of setting the catchability by hand can be seen in Section 11.

As has been shown in Sections 5 and 6, there is a `summary()` method for *MizerParams* objects which prints a useful summary of the model parameters:

```
summary(params)
```

As well as giving a summary of the species in the model and what gear is fishing what species, it gives a summary of the size structure of the community. For example there are 100 size classes in the community, ranging from 0.001 to 4.4×10^4. These values are controlled by the arguments `no_w`, `min_w` and `max_w` respectively. For example, if we wanted 200 size classes in the model we would use:

```
params200 <- MizerParams(params_data, no_w=200)
summary(params200)
```

So far we have created a *MizerParams* object by passing in only the species parameter data.frame argument. This means the interaction matrix will be set to the default value (see Section 8.5). This is a matrix of 1s, implying that all species fully interact with each other, i.e. the species are spread homogeneously across the model area. For the North Sea this is not the case and so the model would be improved by also including an interaction matrix which describes the spatial overlap between species (see Section 8.5).

An example interaction matrix for the North Sea has been included in `mizer` as a .csv file. The location of the file can be found by running:

```
system.file("doc/inter.csv",package="mizer")
```

Take a look at it in a spreadsheet if you want. As mentioned above, to read this file into R we can make use of the `read.csv()` function. However, this time we want the first column of the .csv file to be the row names. We therefore use an additional argument to the `read.csv()` function: `row.names`.

```
inter <- read.csv("inter.csv", row.names=1)
```

The `read.csv()` function reads the data into a data.frame. We want the interaction matrix to be of class *matrix* so we need to make use of the `as()` function.

```
inter <- as(inter, "matrix")
```

We now have an interaction matrix that can be passed to the *MizerParams* constructor along with the species parameter data.frame. To make the *MizerClass* object you just call the constructor method and pass in the arguments. We will use default values for the remainder of the arguments to the `MizerParams()` method (see Section 8.6 and Table 3). This means that we only need to pass in two arguments to the constructor:

```
params <- MizerParams(params_data, interaction = inter)
```

Note that the first argument must be the species parameters data.frame. The remaining arguments can be in any order but should be named. If we didn't want to use default values for the other arguments we would pass them in to the constructor by name.

We now have all we need to start running projections. Before we get to that though, we'll take a quick look at how different fishing gears can be set up.

8.8 Setting different gears

In the above example, each species is caught by a different gear (named after the species it catches). This is the default when there is no `gear` column in the species parameter data.frame.

Here, we look at an example where we do specify the fishing gears. We take the original `params_data` species parameter data.frame that was read in above and bind an additional column, `gear`, to it. This `gear` column contains the name of the gear that catches the species in that row. Here we set up four different gears: Industrial, Pelagic, Beam and Otter trawl, that catch different combinations of species.

```
params_data_gears <- params_data
params_data_gears$gear <- c("Industrial","Industrial","Industrial",
                        "Pelagic","Beam","Otter",
                        "Beam","Otter","Beam",
                        "Otter","Otter","Otter")
```

If you inspect the `params_data` object you will see a new column, `gear`, has been added to it. We then make a new *MizerParams* object as before:

```
params_gears <- MizerParams(params_data_gears, interaction = inter)
```

You can see the result by calling `summary()` on the `params_gears` object.

In this example the same gear now catches multiple stocks. For example, the "Industrial" gear catches Sprat, Sandeel and Norway Pout. Why would we want to set up the gears like this? In the next section we will see that to project the model through time you can specify the fishing effort for each gear through time. By setting the gears up in this way you can run different management scenarios of changing the efforts of the fishing gears rather than on individual species. It also means that after a simulation has been run you can examine the catches by gear.

9 Running a simulation

In the preceding section and in the sections on the community and trait-based models above, we used the `project()` method to perform simple simulations where the fishing effort was held constant through the duration of the simulation. In the trait-based model example, we also looked at how the effort for different gears could be specified. In this section we take a detailed look at how the `project()` method works and the different ways in which effort and time can be set up.

In `mizer` simulations are performed using the `project()` method. This method takes a *MizerParams* object and projects it forward through time, starting from an initial population abundance and with a pre-determined fishing effort pattern.

Running a projection with `project()` requires various arguments:

A *MizerParams* object The model parameters (see previous section);

Fishing effort The fishing effort of each gear through time;

Initial population The initial abundances of the stocks and the background spectrum;

Time arguments Arguments to control the time of the simulation, including the simulation time step, the length of the simulation and how frequently the output is stored.

The help page for `project()` describes the arguments in more detail.

The *MizerParams* class was explored in the previous section. In this section we will look at the other arguments and use examples to perform some simple projections.

The object returned from calling `project()` is of class *MizerSim*. This contains the abundances of the species in the model by time and size. It is described fully in Section 9.4.

9.1 The time arguments

There are three arguments that control time in the `project()` method: `dt`, `t_max` and `t_save`. All of them have default values.

`t_max` determines the maximum time of the simulation, i.e. how long the projection is run for. The default value for `t_max` is 100 respectively.

`dt` is the time step used by the numerical solver in `project()`. This is the time step on which the model operates. The smaller the value, the longer the model will take to run. However, sometimes it is necessary to use a small value to avoid numerical instabilities. The default value is 0.1.

The final argument is `t_save`. This sets how frequently `project()` stores the state of the model in the resulting *MizerSim* object. For example, if `t_save` = 2, the state of the model is stored at t = 0, 2, 4... etc. `t_save` must be a multiple of `dt`. The default value of `t_save` is 1.

9.2 Setting the fishing effort

The fishing effort argument describes the effort of the fishing gears in the model through time. We have already seen that information on the fishing gears and their selectivities and catchabilities is stored in the `MizerParams` argument.

There are three ways of setting the fishing effort. Examples of all three can be seen in Section 9.5.

The simplest way is by passing the `effort` argument as a single number. This value is then used as the fishing effort by all of the gears at each time step of the projection, i.e. fishing effort is constant throughout the simulation and is the same for all gears. We have seen this method in the community and trait-based model sections above. The length of the simulation is determined by the `t_max` argument (see Section 9.1).

The second method for setting the fishing effort is to use a numeric vector that has the same length as the number of gears. The values in the vector are used as the fishing effort of each gear at each time step, i.e. again, the fishing effort is constant through time but now each gear can have a different constant effort. The effort vector must be named and the names must be the same as the gears in the `MizerParams` object. Again, the length of the simulation is determined by the `t_max` argument.

Finally, the most sophisticated way of setting the fishing effort is to use a two-dimensional array or matrix of values, set up as time step by gear. Each row of the array has the effort values of each fishing gear by time. The array must have dimension names. The names of the first dimension (the row names) are the times. The steps between the times can be greater than the `dt` argument but must

be contiguous. The names of the second dimension (the column names) must match the names of the gears in the *MizerParams* object used in the projection.

It is not necessary to supply a `t_max` argument when the effort is specified as an array because the maximum time of the simulation is taken from the dimension names. If a value for `t_max` is also supplied it is ignored.

9.3 Setting the initial population abundance

When running a simulation with the `project()` method, the initial populations of the species and the background spectrum need to be specified. These are passed to `project()` as the arguments `initial_n` and `initial_n_pp` respectively. `initial_n` is a matrix (with dimensions species x size) that contains the initial abundances of each species at size (the sizes must match those in the species size spectrum). `initial_n_pp` is a vector of the same length as the the length of the full spectrum (which can be seen as slot `w_full` in the *MizerParams* object).

By default, the `initial_n` argument has values automatically calculated by the function `get_initial_n()`. The default value for `initial_n_pp` is the carrying capacity of the background spectrum, stored in the `cc_pp` slot of the *MizerParams* parameters object.

9.4 What do you get from running `project()`?

Running `project()` returns an object of type *MizerSim* that stores the results of the projection. Like the *MizerParams* class this is also made up of various "slots", which can be accessed using the `@` operator. An object of *MizerSim* has four slots, details of which can be seen in the help page (`help("MizerSim-class")`). The `params` slot holds the *MizerParams* object that was passed in to `project()`. The `effort` slot holds the fishing effort of each gear through time. Note that the `effort` slot may not be exactly the same as the `effort` argument that was passed in to `project()`. This is because only the saved effort is stored (the frequency of saving is determined by the argument `t_save`). The `n` and `n_pp` slots hold the saved abundances of the species and the background population at size respectively. Note that The `n` and `n_pp` slots have one more row than the `effort` slot. This is to store the initial populations.

9.5 Projection examples

In this section we'll look at how to run simulations with the `project()` method. The examples will focus on how fishing effort can be specified in different ways. The results of the simulations will not be explored in detail. We will leave that for Section 10.2.

Remember that the fishing mortality by size on a species is the product of the selectivity, the catchability and the effort of the gear that caught it (see Equation 8.2). We have not specified any catchability values in the species parameter data.frame so the default value of 1 is used. The selectivity ranges between 0 and 1. This means that in these examples the fishing mortality of a fully selected species is given by the effort of the gear that catches it.

9.5.1 Projections with single, simple constant effort

When we use a single value for the `effort` argument, the value is used as a constant effort for all the gears. This method can be particularly useful for quickly projecting forward without fishing (you just set the `effort` argument to 0).

We will use the `params` object that was created in Section 8.7 in which each species is caught by a separate gear. Here we make the object again:

```
params <- MizerParams(params_data, interaction = inter)
```

As described above, effort is associated with fishing gears. Because we haven't specified any gears in the `params_data` species parameter data.frame, each species is caught by a separate gear, named after the species.

As well as thinking about the `effort` argument we also need to consider the time parameters. We will project the populations forward until time equals 10 (`t_max` = 10), with a time step of 0.1 (`dt` = 0.1), saving the output every time step (`t_save` = 1). We use a constant effort value of 1.0.

```
sim <- project(params, effort = 1, t_max = 10, dt = 0.1, t_save = 1)
```

The resulting `sim` object is of class *MizerSim*. At this point we won't explore how the results can be investigated in detail. However, we will use the basic summary plot that you have seen before:

```
plot(sim)
```

Without fishing the community settles down to equilibria at about t = 50 (Figure 15). The big difference between this multispecies model and the trait-based model can be seen in the range of M2 and feeding level values. With the trait-based model all the "species" had the same M2 and feeding level patterns (see Figure 8). Here the species all have different patterns, driven by their differing life history characteristics and the heterogeneous interaction matrix.

You can also see in Figure 15 that each species has different fishing selectivity (see the Total fishing mortality panel). Remember that the default setting for the fishing gears is a knife-edge gear where the knife-edge is positioned at the species `w_mat` parameter.

The effort through time can be inspected by looking at the `effort` slot (we use the `head()` function to just show the first few lines). The `effort` slot shows the effort by time and gear. You can see here that each species is caught by a separate gear, and the gear is named after the species. In this example, we specified the `effort` argument as a single numeric of value 1. As you can see this results in the same effort being used for all gears for all time steps:

```
head(sim@effort)
```

```
##     gear
## time Sprat Sandeel N.pout Herring Dab Whiting Sole Gurnard Plaice Haddock
##    1     1       1      1       1   1       1    1       1      1       1
##    2     1       1      1       1   1       1    1       1      1       1
##    3     1       1      1       1   1       1    1       1      1       1
##    4     1       1      1       1   1       1    1       1      1       1
##    5     1       1      1       1   1       1    1       1      1       1
##    6     1       1      1       1   1       1    1       1      1       1
##     gear
## time Cod Saithe
##    1   1      1
##    2   1      1
##    3   1      1
```

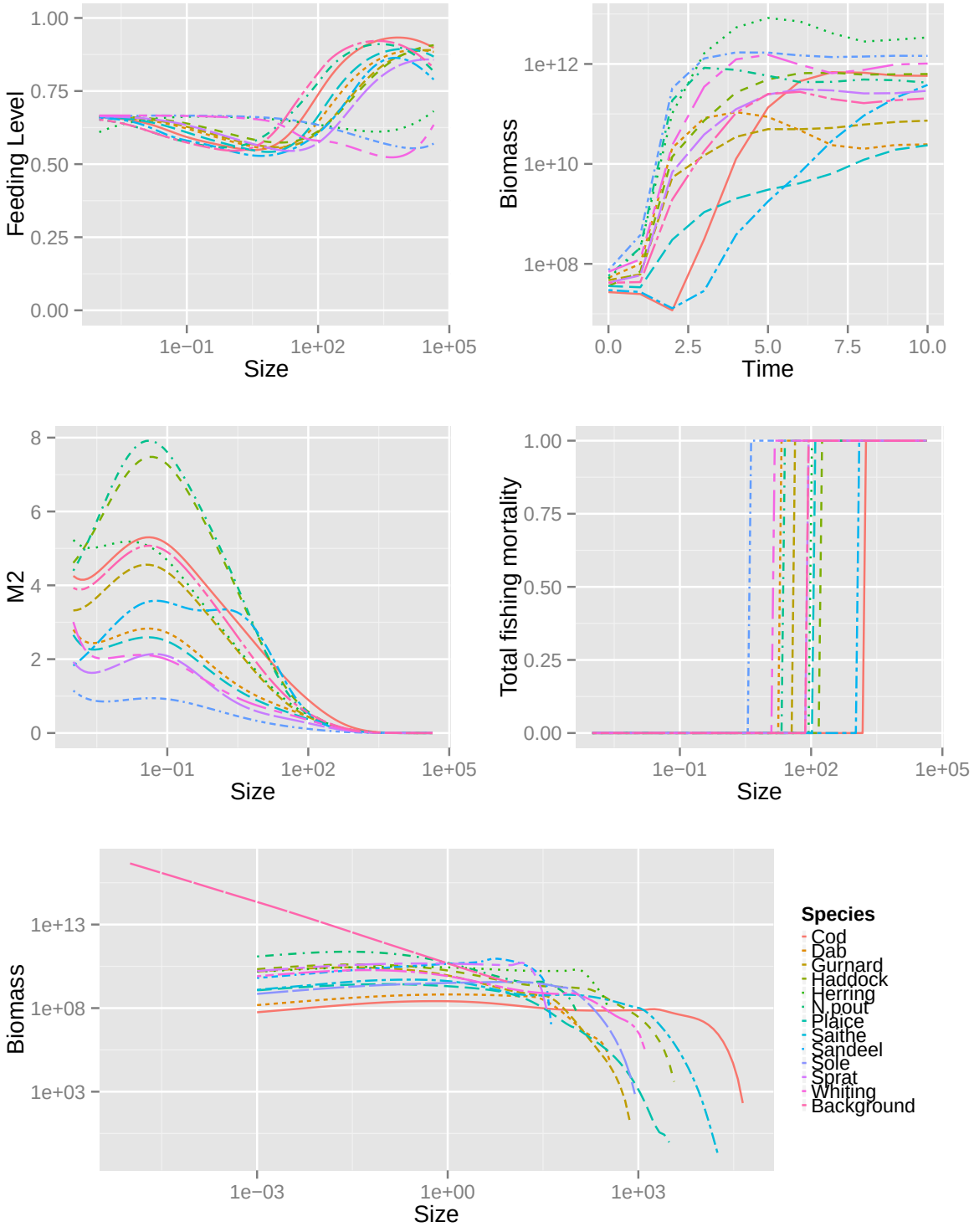

Figure 15: Plot of the North Sea multispecies model with no default fishing gears and constant effort of 1.

```
##    4   1      1
##    5   1      1
##    6   1      1
```

A `summary()` method is also available for objects of type *MizerSim*. This is essentially the same as the summary for *MizerParams* objects, but includes information on the simulation time parameters.

```
summary(sim)
```

If we decrease `t_save` but keep `t_max` the same then we can see that the time dimension of the `effort` slot changes accordingly. This will also be true of the `n` and `n_pp` slots. Here we reduce `t_save` to 0.5, meaning that the effort and abundance information is stored at t = 1.0, 1.5, 2.0 etc.

```
sim <- project(params_gears, effort = 1, t_max = 10, dt = 0.1, t_save = 0.5)
head(sim@effort)

##      gear
## time  Industrial Pelagic Beam Otter
##   1            1       1    1     1
##   1.5          1       1    1     1
##   2            1       1    1     1
##   2.5          1       1    1     1
##   3            1       1    1     1
##   3.5          1       1    1     1
```

9.5.2 Setting constant effort for different gears

As mentioned above, we can also set the effort values for each gear separately using a vector of effort values. This still keeps the efforts constant through time but it means that each gear can have a different constant effort.

Here we will use the *MizerParams* object with four gears, `params_gears`, that we created in Section 8.8. The names of the gears are Industrial, Pelagic, Beam, Otter. We need to create a named vector of effort, where the names match the gears. For example, here we want to switch off the industrial gear (i.e. effort = 0), keep the pelagic gear effort at 1, set the effort of the beam trawl gears to 0.3 and the effort of the otter trawl gear to 0.7. We set the `effort` like this:

```
effort <- c(Industrial = 0, Pelagic = 1, Beam = 0.3, Otter = 0.7)
```

We then call `project()` with this effort and inspect the resulting `effort` slot (again we use the `head()` function to just show the first few lines):

```
sim <- project(params_gears, effort = effort, t_max = 10, dt = 1, t_save = 1)
head(sim@effort)

##     gear
## time Industrial Pelagic Beam Otter
##    1          0       1  0.3   0.7
##    2          0       1  0.3   0.7
```

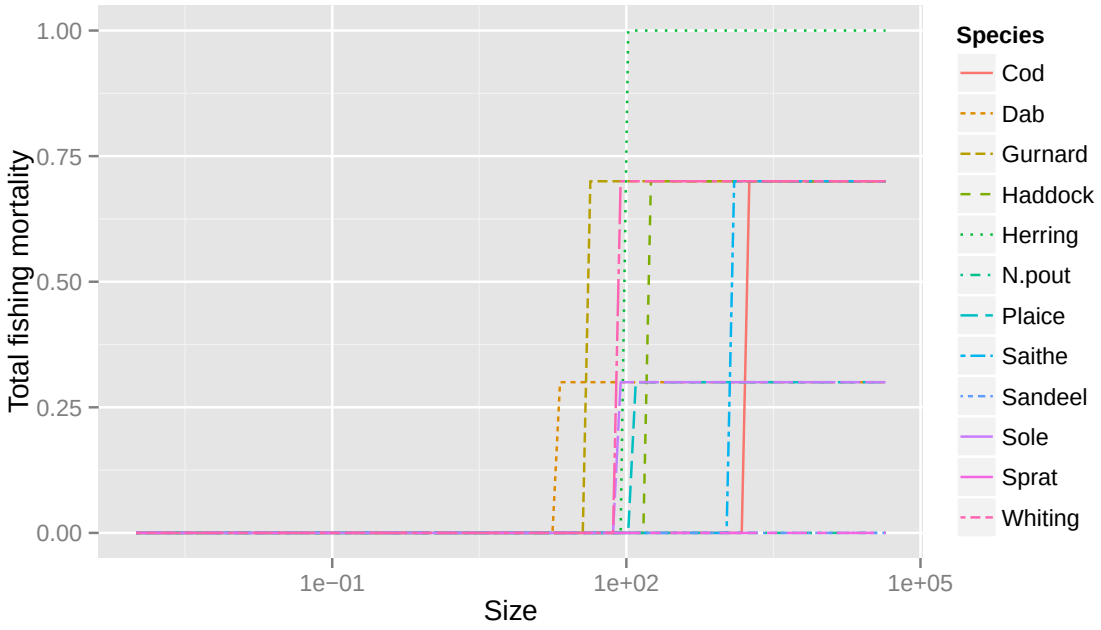

Figure 16: An example of using the `plotFMort()` method to show how different efforts for different gears can be specified.

```
##    3          0       1  0.3   0.7
##    4          0       1  0.3   0.7
##    5          0       1  0.3   0.7
##    6          0       1  0.3   0.7
```

You can see that the effort for each gear is constant but each gear has the effort that was specified in the `effort` argument.

This impact of this can be seen plotting the fishing mortality. There is a dedicated plot, `plotFMort()`, that shows the fishing mortality at size for each species at a particular time step (the default is the final time step). The fishing mortality on each of the species is determined by the effort of the gear that caught it (Figure 16).

```
plotFMort(sim)
```

9.5.3 An example of changing effort through time

In this example we set up a more complicated fishing effort structure that allows the fishing effort of each gear to change through time. As mentioned above, to do this, effort must be supplied as a two dimensional array or matrix. The first dimension is time and the second dimension is gear. The dimensions must be named. The gear names must match the gears in the *MizerParams* object. Also, as mentioned above, if effort is passed in as an array then the length of the simulation is determined by the time dimension names and the argument `t_max` is not used.

Here, we will use the `params_gears` object we created in Section 8.8 which has four gears. The names of the gears are Industrial, Pelagic, Beam, Otter.

In this example, we will project forward for 10 time steps. The effort of the industrial gear is held constant at 0.5, the effort of the pelagic gear is increased linearly from 1 to 2, the effort of the beam trawl decreases linearly from 1 to 0, whilst the effort of the otter trawl decreases linearly from 1 to 0.5.

First we create the empty `effort` array:

```
gear_names <- c("Industrial","Pelagic","Beam","Otter")
times <- seq(from = 1, to = 10, by = 1)
effort_array <- array(NA, dim = c(length(times), length(gear_names)),
    dimnames = list(time = times, gear = gear_names))
```

Then we fill it up, one gear at a time, making heavy use of the `seq()` function to create a sequence:

```
effort_array[,"Industrial"] <- 0.5
effort_array[,"Pelagic"] <- seq(from = 1, to = 2, length = length(times))
effort_array[,"Beam"] <- seq(from = 1, to = 0, length = length(times))
effort_array[,"Otter"] <- seq(from = 1, to = 0.5, length = length(times))
```

The first few rows of the effort array are shown as an illustration:

```
head(effort_array)
```

```
##     gear
## time Industrial Pelagic   Beam  Otter
##    1        0.5   1.000 1.0000 1.0000
##    2        0.5   1.111 0.8889 0.9444
##    3        0.5   1.222 0.7778 0.8889
##    4        0.5   1.333 0.6667 0.8333
##    5        0.5   1.444 0.5556 0.7778
##    6        0.5   1.556 0.4444 0.7222
```

Now we can use this effort array in the projection:

```
sim <- project(params_gears, effort=effort_array, dt=0.1, t_save = 1)
head(sim@effort)
```

```
##     gear
## time Industrial Pelagic   Beam  Otter
##    1        0.5   1.000 1.0000 1.0000
##    2        0.5   1.111 0.8889 0.9444
##    3        0.5   1.222 0.7778 0.8889
##    4        0.5   1.333 0.6667 0.8333
##    5        0.5   1.444 0.5556 0.7778
##    6        0.5   1.556 0.4444 0.7222
```

As you can see, it can be quite fiddly to set up a complicated effort array so it may be easier to prepare it in advance as a .csv file and read it in, similar to how we read in the interaction matrix in Section 8.7. We give an example of this in Section 11.

Note that in this example we set up the effort array so that the effort was set every whole time step (e.g. time = 1, 2, etc). This does not have to be the case and it is possible to set the effort more frequently than that, e.g. at time = 1.0, 1.5, 2.0, 2.5 etc. The only restriction is that the difference between time dimension names must be at least as big as the `dt` argument.

10 Exploring the simulation results

In the previous sections we saw how to set up a model and project it forward through time under our desired fishing scenario. The result of running a projection is an object of class *MizerSim*. What do we then do? How can we explore the results of the simulation? In this section we introduce a range of summaries, plots and indicators that can be easily produced using methods included in `mizer`.

We will use the same *MizerSim* object for these examples:

```
sim <- project(params_gears, effort=effort_array, dt=0.1, t_save = 1)
```

10.1 Directly accessing the slots of *MizerSim* objects

We have seen in previous sections that a *MizerSim* object is made up of various "slots" that contain the original parameters (`species_params`), the population abundances (`n`), the abundance of the background resource spectrum (`n_pp`) and the fishing effort (`effort`).

The projected species abundances at size through time can be seen in the `n` slot. This is a three-dimensional array (time x species x size). Consequently, this array can get very big so inspecting it can be difficult. In the example we have just run the time dimension of the `n` slot has 11 rows (one for the initial population and then one for each of the saved time steps). There are also 12 species each with 100 sizes. We can check this by running the `dim()` function and looking at the dimensions of the `n` array:

```
dim(sim@n)

## [1]  11  12 100
```

To pull out the abundances of a particular species through time at size you can subset the array. For example to look at Cod through time you can use:

```
sim@n[,"Cod",]
```

This returns a two-dimensional array: time x size, containing the cod abundances. The time dimension depends on the value of the argument `t_save` when `project()` was run. You can see that even though we specified `dt` to be 0.1 when we called `project()`, the `t_save` argument had means that the output is only saved every time step.

The `n_pp` slot can be accessed in the same way.

10.2 Summary methods for *MizerSim* objects

As well as the `summary()` methods that are available for both *MizerParams* and *MizerSim* objects, there are some useful summary methods to pull information out of a *MizerSim* object (see Table 4). All

of these methods have help files to explain how they are used. (It is also possible to use most of these methods with a *MizerParams* object if you also supply the population abundance as an argument. This can be useful for exploring how changes in parameter value or abundance can affect summary statistics and indicators. We won't explore this here but you can see their help files for more details.)

The methods `getBiomass()` and `getN()` have additional arguments that allow the user to set the size range over which to calculate the summary statistic. This is done by passing in a combination of the arguments `min_l`, `min_w`, `max_l` and `max_w` for the minimum and maximum length or weight.

If `min_l` is specified there is no need to specify `min_w` and so on. However, if a length is specified (minimum or maximum) then it is necessary for the species parameter data.frame (see Section 8.2) to include the parameters `a` and `b` for length-weight conversion. It is possible to mix length and weight constraints, e.g. by supplying a minimum weight and a maximum length. The default values are the minimum and maximum weights of the spectrum, i.e. the full range of the size spectrum is used.

10.2.1 Examples of using the summary methods

Here we show a simple demonstration of using a summary method using the `sim` object we created earlier. Here, we use `getSSB()` to calculate the SSB of each species through time (note the use of the `head()` function to only display the first few rows).

```
ssb <- getSSB(sim)
dim(ssb)

## [1] 11 12

head(ssb)

##     sp
## time     Sprat   Sandeel    N.pout   Herring       Dab   Whiting      Sole
##    0 3.603e+07 5.473e+07 3.557e+07 2.990e+07 3.848e+07 2.993e+07 2.919e+07
##    1 3.693e+07 6.001e+07 3.584e+07 2.855e+07 3.568e+07 2.724e+07 2.681e+07
##    2 9.445e+08 9.499e+10 8.540e+09 1.489e+09 1.264e+09 1.614e+07 1.435e+07
##    3 4.147e+10 8.648e+11 3.260e+11 1.139e+11 1.752e+10 1.146e+09 2.851e+08
##    4 3.203e+11 1.569e+12 4.126e+11 9.342e+11 4.468e+10 8.528e+09 5.133e+09
##    5 7.695e+11 1.917e+12 3.610e+11 2.067e+12 5.693e+10 5.452e+10 2.283e+10
##     sp
## time   Gurnard   Plaice   Haddock       Cod   Saithe
##    0 3.448e+07 25984507 2.596e+07 2.006e+07 21521377
##    1 3.156e+07 23615150 2.378e+07 1.821e+07 19530302
##    2 1.840e+07  9306041 2.397e+07 6.968e+06  7454960
##    3 4.338e+08  4407878 2.239e+09 4.283e+06  3541163
##    4 2.333e+09  7259033 2.249e+10 2.602e+08  3503658
##    5 3.305e+09 72834627 5.387e+10 1.435e+10  7996987
```

As mentioned above, we can specify the size range for the `getBiomass()` and `getN()` methods. For example, here we calculate the total biomass of each species but only include individuals that are larger than 10 g and smaller than 1000 g.

```
biomass <- getBiomass(sim, min_w = 10, max_w = 1000)
head(biomass)
```

```
##     sp
## time     Sprat   Sandeel    N.pout   Herring       Dab   Whiting      Sole
##    0 4.904e+07 5.654e+07 5.158e+07 4.932e+07 5.102e+07 3.469e+07 4.153e+07
##    1 5.130e+07 5.620e+07 5.419e+07 5.314e+07 4.805e+07 3.214e+07 3.868e+07
##    2 2.603e+09 2.404e+10 4.634e+10 6.106e+10 8.994e+09 8.141e+08 1.570e+09
##    3 9.085e+10 6.223e+11 6.337e+11 1.245e+12 6.629e+10 8.850e+09 2.627e+10
##    4 6.253e+11 1.508e+12 5.782e+11 4.270e+12 1.083e+11 4.801e+10 9.182e+10
##    5 1.267e+12 1.956e+12 4.857e+11 6.357e+12 1.106e+11 2.033e+11 2.130e+11
##     sp
## time   Gurnard    Plaice   Haddock       Cod    Saithe
##    0 4.648e+07 1.249e+07 1.047e+07 6.854e+05 1.801e+06
##    1 4.289e+07 1.158e+07 1.042e+07 7.402e+05 1.866e+06
##    2 6.669e+08 1.222e+07 9.051e+09 1.985e+06 3.041e+06
##    3 9.140e+09 2.736e+08 5.915e+10 1.882e+08 6.419e+06
##    4 1.308e+10 1.256e+09 1.535e+11 9.281e+09 1.521e+08
##    5 1.536e+10 2.417e+09 3.407e+11 7.024e+10 1.218e+09
```

10.3 Methods for calculating indicators

Methods are available to calculate a range of indicators from a *MizerSim* object after a projection. These can be seen in Table 5. You can read the help pages for each of the methods for full instructions on how to use them, along with examples.

With all of the methods in the table it is possible to specify the size range of the community to be used in the calculation (e.g. to exclude very small or very large individuals) so that the calculated metrics can be compared to empirical data. This is used in the same way that we saw with the method `getBiomass()` in Section 10.2. It is also possible to specify which species to include in the calculation. See the help files for more details.

10.3.1 Examples of calculating indicators

For these examples we use the `sim` object we created earlier.

The slope of the community can be calculated using the `getCommunitySlope()` method. Initially we include all species and all sizes in the calculation (only the first five rows are shown):

```
slope <- getCommunitySlope(sim)
head(slope)
```

```
##     slope intercept     r2
## 0 -0.4970     14.32 0.7198
## 1 -0.9769     17.57 0.9519
## 2 -1.3303     21.62 0.8063
## 3 -1.4268     22.96 0.7519
## 4 -1.2670     23.67 0.7049
## 5 -1.0714     24.12 0.6597
```

This gives the slope, intercept and R^2 value through time (see the help file for `getCommunitySlope` for more details).

We can include only the species we want with the `species` argument. Here we only include demersal species. We also restrict the size range of the community that is used in the calculation to between 10 g and 5 kg. The `species` is a character vector of the names of the species that we want to include in the calculation.

```
dem_species <- c("Dab","Whiting","Sole","Gurnard","Plaice","Haddock",
    "Cod","Saithe")
slope <- getCommunitySlope(sim, min_w = 10, max_w = 5000,
    species = dem_species)
head(slope)

##     slope intercept     r2
## 0 -0.2885     12.25 0.5562
## 1 -0.8501     16.10 0.9538
## 2 -0.9711     18.47 0.7135
## 3 -1.2231     20.55 0.7683
## 4 -1.1576     21.97 0.7403
## 5 -0.9692     22.56 0.6754
```

10.4 Plotting the results

R is very powerful when it comes to exploring data through plots. A useful package for plotting is `ggplot2`. `ggplot2` uses data.frames for input data. Many of the summary methods and slots of the `mizer` classes are arrays or matrices. Fortunately it is straightforward to turn arrays and matrices into data.frames using the command `melt` which is in the `reshape2` package. Although `mizer` does include some dedicated plots, it is definitely worth your time getting to grips with these `ggplot2` and other plotting packages. This will make it possible for you to make your own plots.

Included in `mizer` are several dedicated plots that use *MizerSim* objects as inputs (see Table 6). As well as displaying the plots, these methods all return objects of type *ggplot* from the `ggplot2` package meaning that they can be further modified by the user (e.g. by changing the plotting theme). See the help page of the individual plot methods for more details. The generic `plot()` method has also been overloaded for *MizerSim* objects. This produces several plots in the same window to provide a snapshot of the results of the simulation.

Some of the plots plot values by size (for example `plotFeedingLevel()` and `plotSpectra()`). For these plots, the default is to use the data at the final time step of the projection. With these plotting methods, it is also possible to specify a different time step or a time range to average the values over before plotting.

10.4.1 Plotting examples

Using the plotting methods is straightforward. For example, to plot the total biomass of each species against time you use the `plotBiomass()` method:

```
plotBiomass(sim)
```

As mentioned above, some of the plot methods plot values against size at a point in time (or averaged over a time period). For these plots it is possible to specify the time step to plot, or the time period to average the values over. The default is to use the final time step. Here we plot the abundance spectra (biomass), averaged over time = 5 to 10 (see Figure 18):

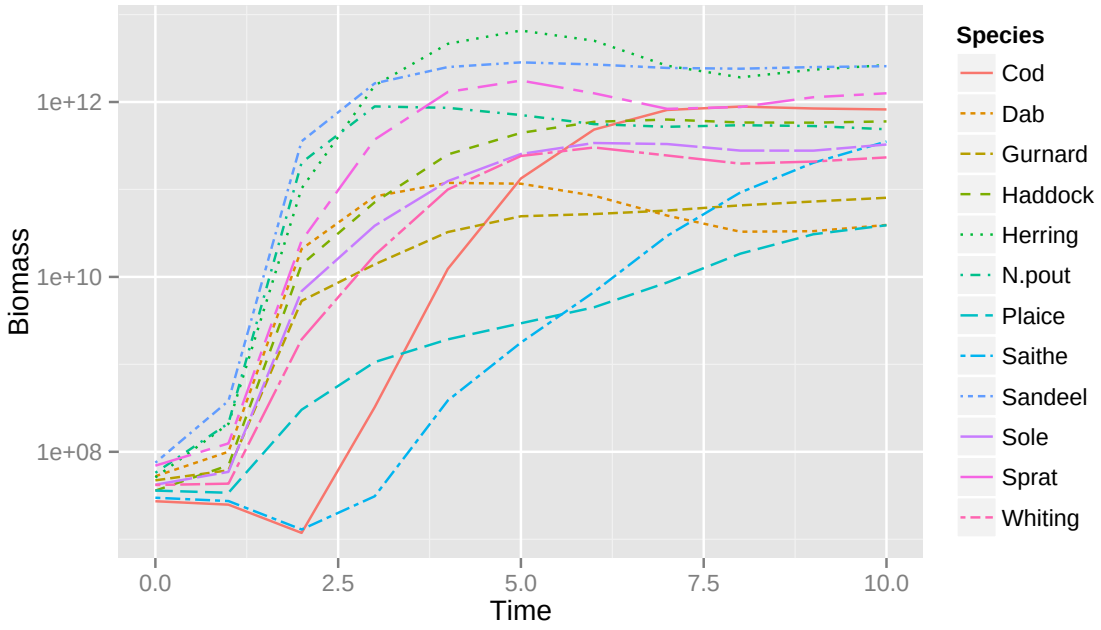

Figure 17: An example of using the `plotBiomass()` method.

```
plotSpectra(sim, time_range = 5:10, biomass=TRUE)
```

As mentioned above, and as we have seen several times in this vignette, the generic `plot()` method has also been overloaded. This produces 5 plots in the same window (`plotFeedingLevel()`, `plotBiomass()`, `plotM2()`, `plotFMort()` and `plotSpectra()`, see Figure 19). It is possible to pass in the same arguments that these individual plots use, e.g. arguments to change the time period over which the data is averaged.

```
plot(sim)
```

11 A multispecies model of the North Sea

In this section we try to pull everything together with an extended example of a multispecies model for the North Sea. First we will set up the model, project it through time using historical levels of fishing effort, and then examine the results. We then run two different future projection scenarios. This example is based on the multispecies model in Blanchard et al. (2013).

11.1 Setting up the North Sea model

The first job is to set up the *MizerParams* object for the North Sea model. In the previous multispecies examples we have already been using the life-history parameters and the interaction matrix for the North Sea model used in Blanchard et al. (2013). We will use them again here but will make some changes. In particular we set up the fishing gears differently.

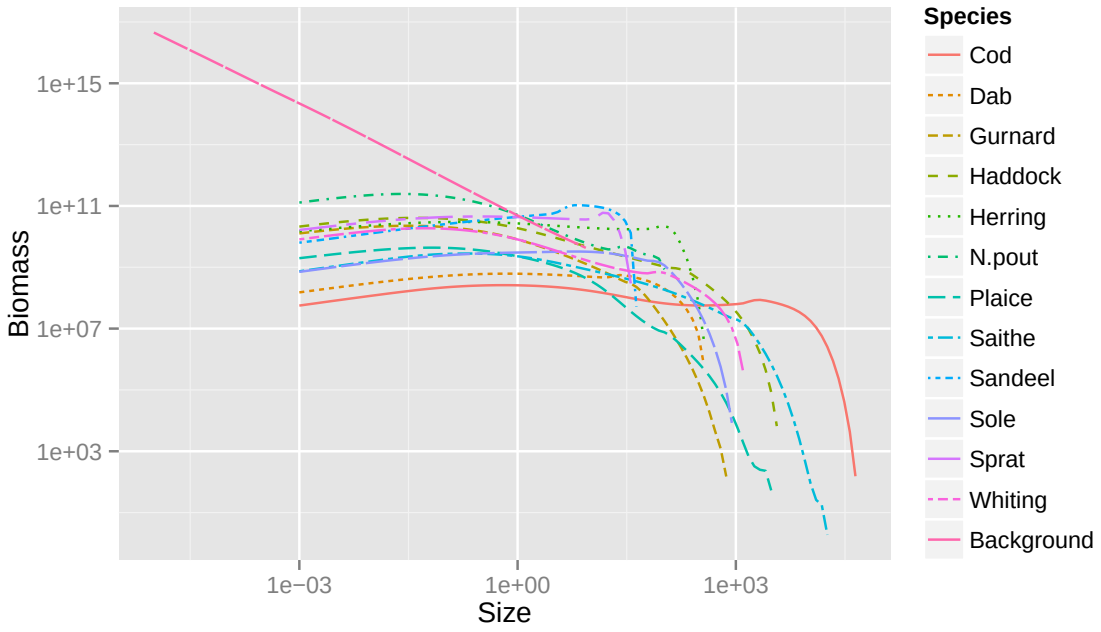

Figure 18: An example of using the `plotSpectra()` method, plotting values averaged over the period $t = 5$ to 10.

The parameters and the interaction matrix are stored as *.csv files that need to be read in. We can use the `system.file()` function to tell us the location of the files. Remember that we need to convert the interaction file into a matrix, hence the use of the `as()` function.

```
params_location <- system.file("doc/NS_species_params.csv",package="mizer")
params_data <- read.csv(params_location)
inter_location <- system.file("doc/inter.csv",package="mizer")
inter <- as(read.csv(inter_location, row.names=1),"matrix")
```

The species in the model are Sprat, Sandeel, N.pout, Herring, Dab, Whiting, Sole, Gurnard, Plaice, Haddock, Cod, Saithe which account for about 90% of the total biomass of all species sampled by research trawl surveys in the North Sea. The `params_data` object is a data.frame with columns for `species`, `w_inf`, `w_mat`, `beta`, `sigma`, `r_max` and `k_vb`. We have seen before that only having these columns in the species data.frame is sufficient to make a *MizerParams* object. Any missing columns will be added by the default values and relationships in the *MizerParams* constructor. For example, the data.frame does not include columns for `h` or `gamma`. This means that they will be estimated using the `k_vb` column (see Section 8.2).

We will use the default stock-recruitment relationship, which is the Beverton-Holt shape. As we saw in Section 8.4, this requires a column `r_max` in the species data.frame which contains the maximum recruitment flux for each species. This column is already in the `params_data` data.frame. The values for `r_max` are taken from Blanchard et al. (2013). The values were found through a calibration process which is not covered here but will be added to a later version of this manual.

At the moment, the species data.frame does not contain any information on the selectivity of the species. By default, the selectivity function is a knife-edge which only takes a single argument, `knife_edge_size`. In this model we want the selectivity pattern to be a sigmoid shape which more

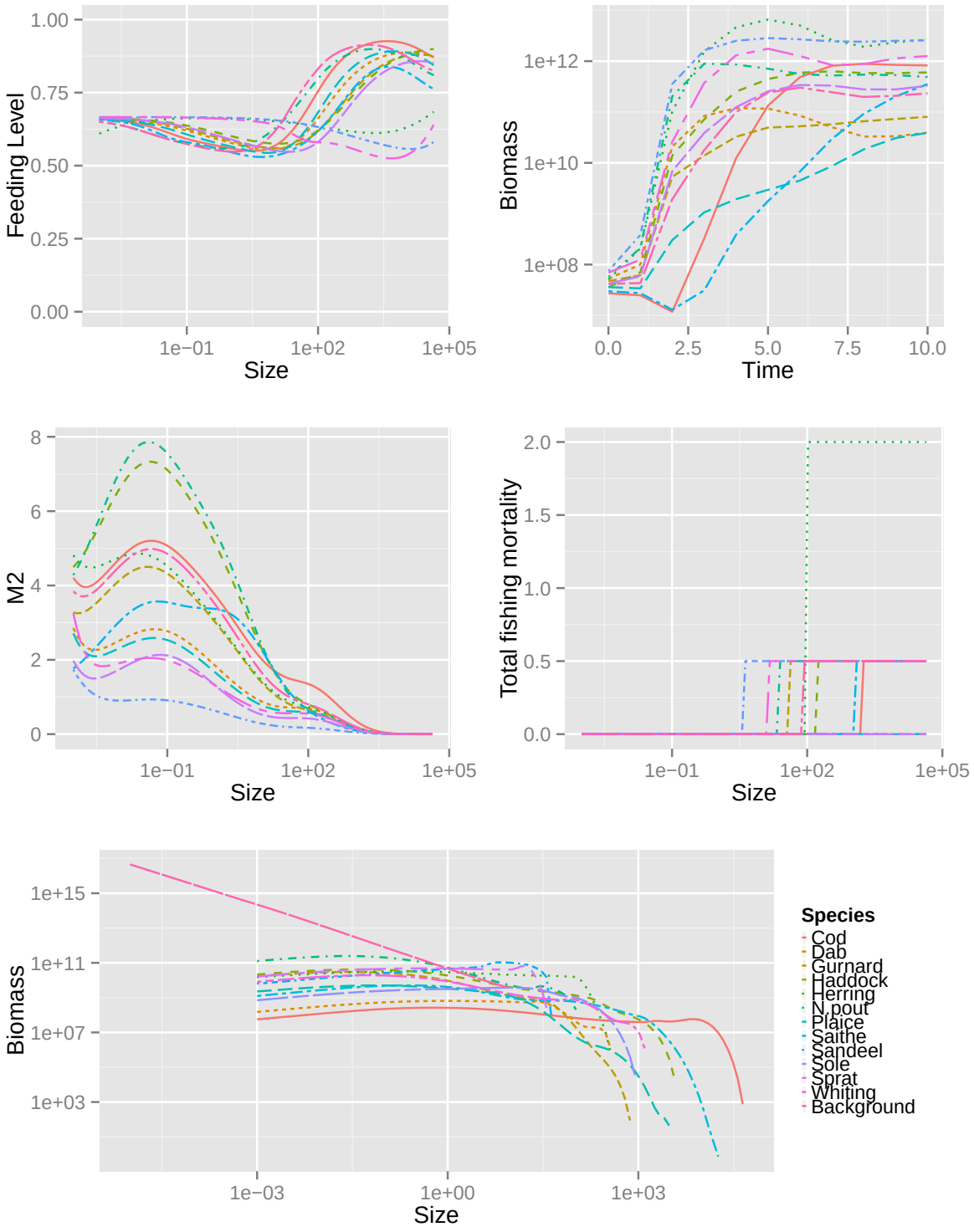

Figure 19: Example output from using the summary `plot()` method.

accurately reflects the selectivity pattern of trawlers in the North Sea. The sigmoid selectivity function is expressed in terms of length rather than weight and uses the parameters `l25` and `l50`, which are the lengths at which 25% and 50% of the stock is selected. The length based sigmoid selectivity looks like:

$$S_l = \frac{1}{1 + \exp(S1 - S2\ l)} \tag{11.1}$$

where l is the length of an individual, S_l is the selectivity at length, $S2 = \log(3)/(l50 - l25)$ and $S1 = l50 \cdot S2$.

This selectivity function is included in `mizer` as `sigmoid_length()`. You can see the help page for more details. As well as the arguments `l25` and `l50`, this function has the arguments `a` and `b` to convert between length and weight: $w = al^b$. This is because the sigmoid selectivity function is defined in terms of length, and the size spectrum model works in terms of weight. As explained in Section 8.3, all arguments of the selectivity function need to be in the species parameter data.frame. Therefore, columns for `l25`, `l50`, `a` and `b` need to be added to `params_data`. We also add a column specifying the name of the selectivity function we wish to use. The data is taken from Blanchard et al. (2013). Note it would probably be easier to add this data directly to the *.csv file and then read it in rather than type it in by hand like we do here:

```
params_data$sel_func <- "sigmoid_length"
params_data$l25 <- c(7.6, 9.8, 8.7, 10.1, 11.5, 19.8, 16.4, 19.8, 11.5,
    19.1, 13.2, 35.3)
params_data$l50 <- c(8.1, 11.8, 12.2, 20.8, 17.0, 29.0, 25.8, 29.0, 17.0,
    24.3, 22.9, 43.6)
params_data$a <- c(0.007, 0.001, 0.009, 0.002, 0.010, 0.006, 0.008, 0.004,
0.007, 0.005, 0.005, 0.007)
params_data$b <- c(3.014, 3.320, 2.941, 3.429, 2.986, 3.080, 3.019, 3.198,
    3.101, 3.160, 3.173, 3.075)
```

Note that we do not set up a `gear` column to give specific gear names. This means that each species will be caught by a separate gear named after the species.

In this model we are interested in projecting forward using historical fishing mortalities. The historical fishing mortality from 1967 to 2010 for each species is stored in the csv file `NS_f_history.csv` included in the package. As before, we can use `read.csv()` to read in the data. This reads the data in as a *data.frame*. We want this to be a *matrix* so we use the `as()` function:

```
f_location <- system.file("doc/NS_f_history.csv",package="mizer")
f_history <- as(read.csv(f_location, row.names=1), "matrix")
```

We can take a look at the first years of the data:

```
head(f_history)
```

```
##      Sprat Sandeel N.pout Herring     Dab Whiting   Sole Gurnard Plaice
## 1967     0       0      0  1.0360 0.09418  0.8295 0.6502       0 0.4709
## 1968     0       0      0  1.7345 0.07376  0.8009 0.7831       0 0.3688
## 1969     0       0      0  1.4345 0.07574  1.3168 0.8744       0 0.3787
## 1970     0       0      0  1.4342 0.10537  1.3474 0.6390       0 0.5269
```

```
## 1971     0       0      0  1.8235 0.08386  0.9742 0.8168       0 0.4193
## 1972     0       0      0  0.9034 0.09044  1.3149 0.7383       0 0.4522
##      Haddock    Cod Saithe
## 1967  0.7429 0.6677 0.4725
## 1968  0.7085 0.6994 0.4270
## 1969  1.3303 0.6918 0.3845
## 1970  1.3671 0.7071 0.5987
## 1971  0.9173 0.7738 0.4828
## 1972  1.3279 0.8393 0.5796
```

As mentioned in Section 8.3, fishing mortality is calculated as the product of selectivity, catchability and fishing effort (see Equation 8.2). The values in `f_history` are absolute levels of fishing mortality. We have seen that the fishing mortality in the `mizer` simulations is driven by the fishing effort argument passed to the `project()` function. Therefore if we want to project forward with historical fishing levels, we need to provide `project()` with effort values that will result in these historical fishing mortality levels.

One of the model parameters that we have not really considered so far is `catchability`. Catchability is a scalar parameter used to modify the fishing mortality at size given the selectivity at size and effort of the fishing gear. By default catchability has a value of 1, meaning that an effort of 1 results in a fishing mortality of 1 for a fully selected species. When considering the historical fishing mortality, one option is therefore to leave catchability at 1 for each species and then use the `f_history` matrix as the fishing effort. However, an alternative method is to use the effort relative to a chosen reference year. This can make the effort levels used in the model more meaningful. Here we use the year 1990 as the reference year. If we set the catchability of each species to be the same as the fishing mortality in 1990 then an effort of 1 in 1990 will result in the fishing mortality being what it was in 1990. The effort in the other years will be relative to the effort in 1990.

The catchability can be set by including a `catchability` column in the species parameters data.frame. Doing this overwrites the default values when the *MizerParams* constructor is called.

```
params_data$catchability <- as.numeric(f_history["1990",])
```

Considering the other model parameters, we will use default values for all of the other parameters apart from `kappa`, the carrying capacity of the resource spectrum (see Section 3.9). This was estimated along with the values `r_max` as part of the calibration process described in Blanchard et al. (2013).

We now have all the information we need to create the `MizerParams` object using the species parameters data.frame.

```
params <- MizerParams(params_data, inter, kappa = 9.27e10)
```

11.2 Setting up and running the simulation

As we set our catchability to be the level of fishing mortality in 1990, before we can run the projection we need to rescale the effort matrix to get a matrix of efforts relative to 1990. To do this we want to rescale the `f_history` object to 1990 so that the relative fishing effort in 1990 = 1. This is done using R function `sweep()`. We then check a few rows of the effort matrix to check this has happened:

```
relative_effort <- sweep(f_history,2,f_history["1990",],"/")
relative_effort[as.character(1988:1992),]

##       Sprat Sandeel N.pout Herring   Dab Whiting   Sole Gurnard Plaice
## 1988 0.8954  1.2633 0.8954   1.215 1.177  0.9973 1.2787  0.0000  1.177
## 1989 1.1046  1.2931 1.1046   1.233 1.074  0.8798 0.9910  0.0000  1.074
## 1990 1.0000  1.0000 1.0000   1.000 1.000  1.0000 1.0000  1.0000  1.000
## 1991 1.1902  0.8814 1.1902   1.108 1.143  0.8097 1.0045  0.8097  1.143
## 1992 1.2500  0.8501 1.2500   1.317 1.113  0.7719 0.9506  0.7719  1.113
##      Haddock    Cod Saithe
## 1988  0.9946 1.0460  1.033
## 1989  0.8546 1.0605  1.122
## 1990  1.0000 1.0000  1.000
## 1991  0.7971 1.0011  0.962
## 1992  0.8797 0.9709  1.053
```

We could just project forward with these relative efforts. However, the population dynamics in the early years will be strongly determined by the initial population abundances (known as the transient behaviour - essentially the initial behaviour before the long term dynamics are reached). As this is ecology, we don't know what the initial abundance are. One way around this is to project forward at a constant fishing mortality equal to the mortality in the first historical year until equilibrium is reached. We then can carry on projecting forward using the remaining years of effort. This approach reduces the impact of transient dynamics.

Here we make an initial effort matrix of 100 years at the first effort level. We need to include dimension names for the time dimension. We then stick it on top of the original matrix of historical relative effort using `rbind()`.

```
initial_effort <- matrix(relative_effort[1,],byrow=TRUE, nrow=100,
    ncol=ncol(relative_effort), dimnames = list(1867:1966))
relative_effort <- rbind(initial_effort,relative_effort)
```

We now have our parameter object and out matrix of efforts relative to 1990. This includes an initial 100 years of constant relative effort at the 1957 level, followed by the relative effort from 1957 to 2010. We use this effort matrix as the `effort` argument to the `project()` method. We use `dt` $= 0.5$ (the simulation will run faster than with the default value of 0.1, but tests show that the results are still stable) and save the results every year.

```
sim <- project(params, effort=relative_effort, dt = 0.5, t_save = 1)
```

Plotting the results, we can see how the biomasses of the stocks change over time (Figure 20). You can see the 100 year period of transients with constant fishing fishing mortality before the historical relative mortality is used from 1967.

```
plotBiomass(sim)
```

To explore the state of the community it is useful to calculate indicators of the unexploited community. Therefore we also project forward for 100 years with 0 fishing effort.

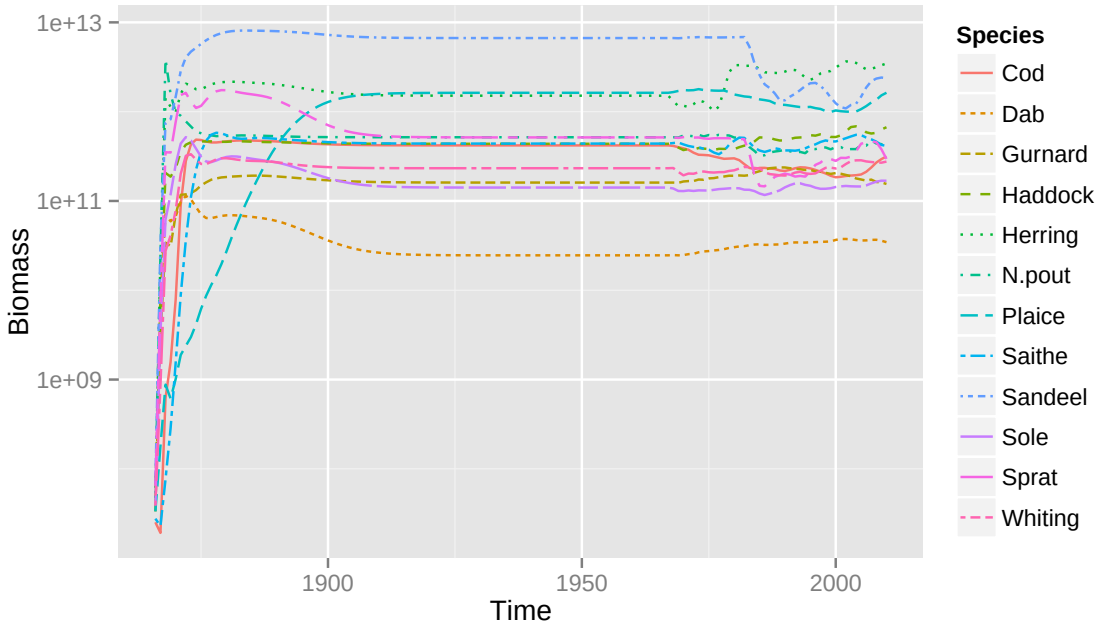

Figure 20: Simulated biomasses of stocks in the North Sea with 100 years of transients.

```
sim0 <- project(params, effort=0, dt = 0.5, t_save = 1, t_max = 100)
```

11.3 Exploring the model outputs

Here we look at some of the ways the results of the simulation can be explored. We calculate the community indicators "mean maximum weight", "mean individual weight", "community slope" and the "large fish indicator" (LFI) over the simulation period, and compare them to the unexploited values. We also compare the simulated values of the LFI to a community target based on achieving a high proportion of the unexploited value of the LFI if $0.8LFI_{F=0}$ (Rochet et al., 2011).

The indicators are calculated using the methods described in see Table 5 and Section 10.2. Here we calculate the LFI and the other community indicators for the unexploited community. As in Blanchard et al. (2013), when calculating these indicators we only include demersal species and individuals in the size range 10 g to 100 kg, and the LFI is based on species larger than 40 cm. Each of these methods returns a time series. We are interested only in the equilibrium unexploited values so we just select the final time step (year = 100).

```
demersal_species <- c("Dab","Whiting","Sole","Gurnard","Plaice","Haddock",
    "Cod","Saithe")
lfi0 <- getProportionOfLargeFish(sim0, species = demersal_species,
    min_w = 10, max_w = 100e3, threshold_l = 40)["100"]
mw0 <- getMeanWeight(sim0, species = demersal_species,
    min_w = 10,max_w = 100e3)["100"]
mmw0 <- getMeanMaxWeight(sim0, species = demersal_species,
    min_w = 10, max_w = 100e3)["100","mmw_biomass"]
```

```
slope0 <- getCommunitySlope(sim0, species = demersal_species,
    min_w = 10, max_w = 100e3)["100","slope"]
```

We also calculate the time series of these indicators for the exploited community (we are only interested in the fishing history years, 1967 to 2010, ignoring the transients):

```
years <- 1967:2010
lfi <- getProportionOfLargeFish(sim, species = demersal_species,
    min_w = 10, max_w = 100e3, threshold_l = 40)[as.character(years)]
mw <- getMeanWeight(sim, species = demersal_species,
    min_w = 10, max_w = 100e3)[as.character(years)]
mmw <- getMeanMaxWeight(sim, species = demersal_species,
    min_w = 10, max_w = 100e3)[as.character(years),"mmw_biomass"]
slope <- getCommunitySlope(sim, species = demersal_species,
    min_w = 10, max_w = 100e3)[as.character(years),"slope"]
```

We can plot the exploited and unexploited indicators, along LFI reference level. Here we do it using `ggplot2` which uses data.frames We make three data.frames (one for the time series, one for the unexploited levels and one for the reference level): Each data.frame is a data.frame of each of the measures, stacked on top of each other.

```
library(ggplot2)
# Simulated data
community_plot_data <- rbind(
    data.frame(year = years, measure = "LFI", data = lfi),
    data.frame(year = years, measure = "Mean Weight", data = mw),
    data.frame(year = years, measure = "Mean Max Weight", data = mmw),
    data.frame(year = years, measure = "Slope", data = slope))
# Unexploited data
community_unfished_data <- rbind(
    data.frame(year = years, measure = "LFI", data = lfi0[[1]]),
    data.frame(year = years, measure = "Mean Weight", data = mw0[[1]]),
    data.frame(year = years, measure = "Mean Max Weight", data = mmw0[[1]]),
    data.frame(year = years, measure = "Slope", data = slope0[[1]]))
# Reference level
community_reference_level <-
    data.frame(year=years, measure = "LFI", data = lfi0[[1]] * 0.8)
# Build up the plot
p <- ggplot(community_plot_data) + geom_line(aes(x=year, y = data)) +
    facet_wrap(~measure, scales="free")
p <- p + geom_line(aes(x=year,y=data), linetype="dashed",
    data = community_unfished_data)
p + geom_line(aes(x=year,y=data), linetype="dotted",
    data = community_reference_level)
```

According to our simulations, historically the LFI in the North Sea has been below the reference level.

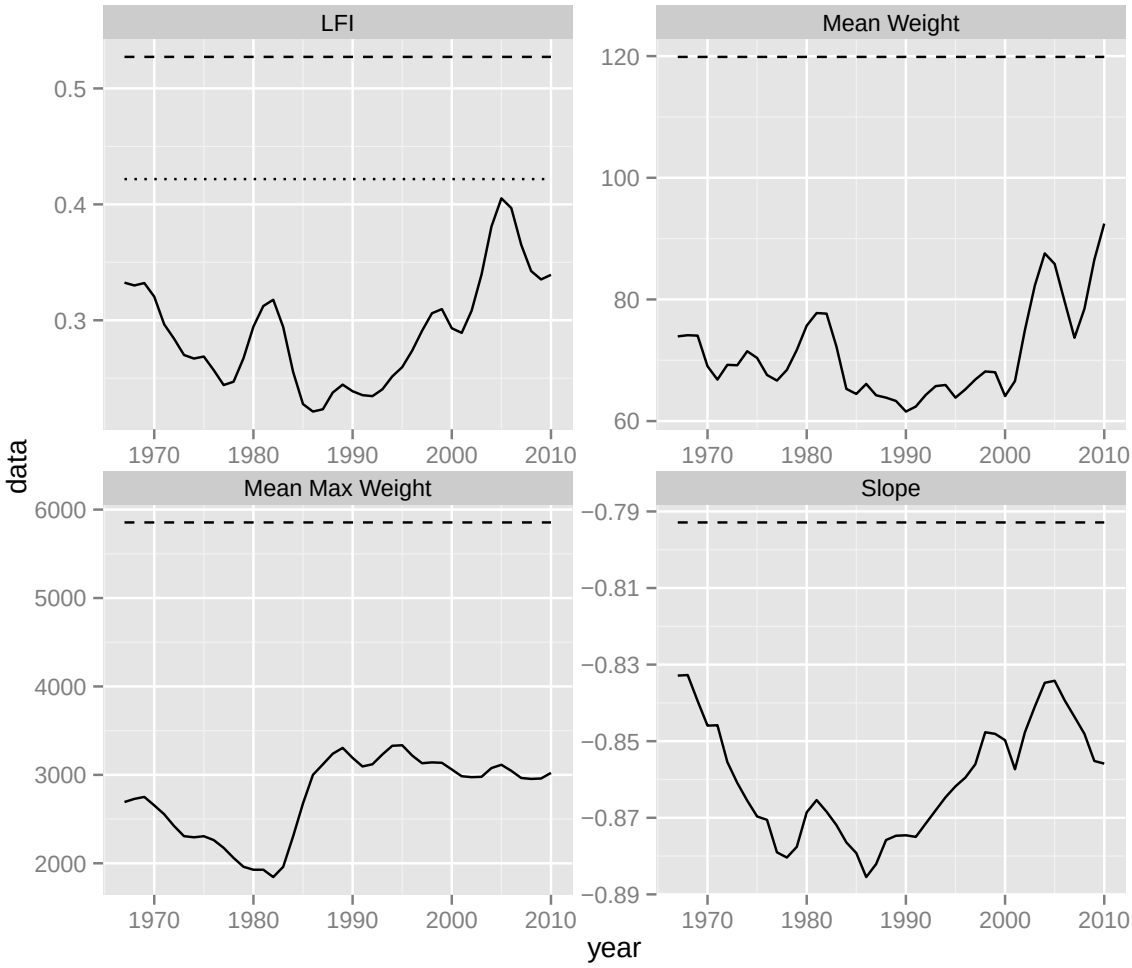

Figure 21: Historical (solid) and unexploited (dashed) and reference (dotted) community indicators for the North Sea multispecies model.

11.4 Future projections

As well as investigating the historical simulations, we can run projections into the future. Here we run two projections to 2050 with different fishing scenarios.

1. Continue fishing at 2010 levels (the status quo scenario).
2. From 2010 to 2015 linearly change the fishing mortality to approach F_{MSY} and then continue at F_{MSY} until 2050.

Rather than looking at community indicators here, we will calculate the SSB of each species in the model and compare the projected levels to a biodiversity target based on the reference point $0.1SSB_{F=0}$ (Rochet et al., 2011). The F_{MSY} values are also taken from Rochet et al. (2011).

Before we can run the simulations, we need to set up arrays of future effort. We will continue to use effort relative to the level in 1990. Here we build on our existing array of relative effort to make an array for the first scenario. Note the use of the `t()` command to transpose the array. This is needed because R recycles by rows, so we need to build the array with the dimensions rotated to start with. We make an array of the future effort, and then bind it underneath the `relative_effort` array used in the previous section.

```
scenario1 <- t(array(relative_effort["2010",], dim=c(12,40),
    dimnames=list(NULL,year = 2011:2050)))
scenario1 <- rbind(relative_effort, scenario1)
```

The relative effort array for the second scenario is more complicated to make and requires a little bit of R gymnastics (it might be easier for you to prepare this in a spreadsheet and read it in). For this one we need values of F_{MSY}.

```
fmsy <- c(Sprat = 0.2, Sandeel = 0.2, N.pout = 0.2, Herring = 0.25, Dab = 0.2,
    Whiting = 0.2, Sole = 0.22, Gurnard = 0.2, Plaice = 0.25, Haddock = 0.3,
    Cod = 0.19, Saithe = 0.3)
scenario2 <- t(array(fmsy, dim=c(12,40), dimnames=list(NULL,year = 2011:2050)))
scenario2 <- rbind(relative_effort, scenario2)
for (sp in dimnames(scenario2)[[2]]){
    scenario2[as.character(2011:2015),sp] <- scenario2["2010",sp] +
        (((scenario2["2015",sp] - scenario2["2010",sp]) / 5) * 1:5)
}
```

Both of our new effort scenarios still include 100 years at the 1967 level to reduce the impact of the transient behaviour. We are now ready to project the two scenarios.

```
sim1 <- project(params, effort = scenario1, dt = 0.5, t_save = 1)
sim2 <- project(params, effort = scenario2, dt = 0.5, t_save = 1)
```

We can now compare the projected SSB values in both scenarios to the biodiversity reference points. First we calculate the biodiversity reference points (from the final time step in the unexploited `sim0` simulation):

```
ssb0 <- getSSB(sim0)["100",]
```

Now we build a data.frame of the projected SSB for each species, ignoring the transients. We make use of the `melt()` function in the very useful `reshape2` package (REF).

```
library(reshape2)
years <- 1967:2050
ssb1 <- getSSB(sim1)[as.character(years),]
ssb2 <- getSSB(sim2)[as.character(years),]
ssb1_df <- melt(ssb1)
ssb2_df <- melt(ssb2)
ssb_df <- rbind(
    cbind(ssb1_df, scenario = "Scenario 1"),
    cbind(ssb2_df, scenario = "Scenario 2"))
ssb_unexploited_df <- cbind(expand.grid(
    sp = names(ssb0),
    time = 1967:2050),
    value = as.numeric(ssb0),
    scenario = "Unexploited")
ssb_reference_df <- cbind(expand.grid(
    sp = names(ssb0),
    time = 1967:2050),
    value = as.numeric(ssb0*0.1),
    scenario = "Reference")
ssb_all_df <- rbind(
    ssb_df,
    ssb_unexploited_df,
    ssb_reference_df)
p <- ggplot(ssb_all_df) +
    geom_line(aes(x = time, y = value, colour = scenario)) +
    facet_wrap(~sp, scales = "free", nrow = 4)
p + theme(legend.position = "none")
```

12 Acknowledgements

Finlay Scott would like to thank the Cefas Seedcorn Project DP266 and the Defra project MF1225 for supporting this work.

References

K. H. Andersen and J. E. Beyer. Asymptotic size determines species abundance in the marine size spectrum. *The American Naturalist*, 168(1):54–61, July 2006. ISSN 0003-0147, 1537-5323. doi: 10.1086/504849.

K. H. Andersen and M. Pedersen. Damped trophic cascades driven by fishing in model marine ecosystems. *Proceedings of the Royal Society B-Biological Sciences*, 277(1682):795–802, March 2010. ISSN 0962-8452. doi: 10.1098/rspb.2009.1512. WOS:000273882800018.

Figure 22: Historical and projected SSB under two fishing scenarios. Status quo (red), Fmsy (yellow). Unexploited (blue) and reference levels (purple) are also shown.

K. H. Andersen and Jake C. Rice. Direct and indirect community effects of rebuilding plans. *ICES Journal of Marine Science: Journal du Conseil*, 67(9):1980–1988, January 2010. ISSN 1054-3139, 1095-9289. doi: 10.1093/icesjms/fsq035.

K. H. Andersen, J. E. Beyer, and P. Lundberg. Trophic and individual efficiencies of size-structured communities. *Proceedings of the Royal Society B: Biological Sciences*, 276(1654):109–114, July 2009a. ISSN 0962-8452, 1471-2954. doi: 10.1098/rspb.2008.0951.

K. H. Andersen, K. D. Farnsworth, M. Pedersen, H. Gislason, and J. E. Beyer. How community ecology links natural mortality, growth, and production of fish populations RID b-5546-2008 RID c-1357-2008. *Ices Journal of Marine Science*, 66(9):1978–1984, October 2009b. ISSN 1054-3139. doi: 10.1093/icesjms/fsp161. WOS:000269611300018.

K.H. Andersen, J.E. Beyer, M. Pedersen, N.G. Andersen, and H. Gislason. Life-history constraints on the success of the many small eggs reproductive strategy. *Theoretical Population Biology*, 73(4): 490–497, June 2008. ISSN 0040-5809. doi: 10.1016/j.tpb.2008.02.001.

Knud Peter Andersen and Erik Ursin. A multispecies extension to the Beverton and Holt theory of fishing, with accounts of phosphorus circulation and primary production. *Meddelelser fra Danmarks Fiskeri- og Havundersøgelser*, 1977.

Eric Benoît and Marie-Joëlle Rochet. A continuous model of biomass size spectra governed by predation and the effects of fishing on them. *Journal of Theoretical Biology*, 226(1):9–21, January 2004. ISSN 0022-5193. doi: 10.1016/S0022-5193(03)00290-X.

Julia L. Blanchard, Simon Jennings, Richard Law, Matthew D. Castle, Paul McCloghrie, Marie-Jolle Rochet, and Eric Benot. How does abundance scale with body size in coupled size-structured food webs? *Journal of Animal Ecology*, 78(1):270280, 2009. ISSN 1365-2656. doi: 10.1111/j.1365-2656.2008.01466.x.

Julia L. Blanchard, Ken H. Andersen, Finlay Scott, Niels T. Hintzen, Gerjan Piet, and Simon Jennings. Evaluating targets and trade-offs among fisheries and conservation objectives using a multispecies size spectrum model. *Journal of Applied Ecology*, 2013.

Samik Datta, Gustav W. Delius, and Richard Law. A jump-growth model for predator-prey dynamics: derivation and application to marine ecosystems. *Bulletin of Mathematical Biology*, 72(6):1361–1382, January 2010. ISSN 0092-8240, 1522-9602. doi: 10.1007/s11538-009-9496-5.

André M. De Roos and Lennart Persson. Physiologically structured models from versatile technique to ecological theory. *Oikos*, 94(1):5171, 2001. ISSN 1600-0706. doi: 10.1034/j.1600-0706.2001.11313.x.

Tom Fenchel. Intrinsic rate of natural increase: The relationship with body size. *Oecologia*, 14(4): 317–326, December 1974. ISSN 0029-8549, 1432-1939. doi: 10.1007/BF00384576.

Martin Hartvig. *Food web ecology*. Ph.D., Lund University, 2011.

Martin Hartvig, K. H. Andersen, and Jan E. Beyer. Food web framework for size-structured populations RID c-4303-2011. *Journal of Theoretical Biology*, 272(1):113–122, March 2011. ISSN 0022-5193. doi: 10.1016/j.jtbi.2010.12.006. WOS:000287227700012.

Shaun S. Killen, Isabel Costa, Joseph A. Brown, and A. Kurt Gamperl. Little left in the tank: metabolic scaling in marine teleosts and its implications for aerobic scope. *Proceedings of the Royal Society B: Biological Sciences*, 274(1608):431–438, July 2007. ISSN 0962-8452, 1471-2954. doi: 10.1098/rspb.2006.3741. PMID: 17164208.

Richard Law, Michael J. Plank, Alex James, and Julia L. Blanchard. Size-spectra dynamics from stochastic predation and growth of individuals. *Ecology*, 90(3):802–811, February 2009. ISSN 0012-9658. doi: 10.1890/07-1900.1.

Olivier Maury, Blaise Faugeras, Yunne-Jai Shin, Jean-Christophe Poggiale, Tamara Ben Ari, and Francis Marsac. Modeling environmental effects on the size-structured energy flow through marine ecosystems. part 1: The model. *Progress in Oceanography*, 74(4):479–499, September 2007. ISSN 0079-6611. doi: 10.1016/j.pocean.2007.05.002.

Johan A. J. Metz and O. Diekmann. *The dynamics of physiologically structured populations*. Springer-Verlag, 1986. ISBN 9780387167862.

Ransom A. Myers and Noel G. Cadigan. Density-dependent juvenile mortality in marine demersal fish. *Canadian Journal of Fisheries and Aquatic Sciences*, 50(8):1576–1590, August 1993. ISSN 0706-652X, 1205-7533. doi: 10.1139/f93-179.

Robert Henry Peters. *The Ecological Implications of Body Size*. Cambridge University Press, March 1986. ISBN 9780521288866.

John G. Pope, Jake C. Rice, Niels Daan, Simon Jennings, and Henrik Gislason. Modelling an exploited marine fish community with 15 parameters - results from a simple size-based model. *Ices Journal of Marine Science*, 63(6):1029–1044, July 2006. ISSN 1054-3139. doi: 10.1016/j.icesjms.2006.04.015. WOS:000239691000007.

W. E. Ricker. Stock and recruitment. *Journal of the Fisheries Research Board of Canada*, 11(5): 559–623, May 1954. ISSN 0015-296X. doi: 10.1139/f54-039.

Marie-Joëlle Rochet, Jeremy S. Collie, Simon Jennings, and Stephen J. Hall. Does selective fishing conserve community biodiversity? predictions from a length-based multispecies model. *Canadian Journal of Fisheries and Aquatic Sciences*, 68(3):469–486, March 2011. ISSN 0706-652X. doi: 10.1139/F10-159. WOS:000287978500008.

Van M. Savage, James F. Gillooly, James H. Brown, Geoffrey B. West, and Eric L. Charnov. Effects of body size and temperature on population growth. *The American Naturalist*, 163(3):429–441, March 2004. ISSN 0003-0147, 1537-5323. doi: 10.1086/381872.

Erik Ursin. On the prey size preferences of cod and dab. *Meddelelser fra Danmarks Fiskeri-og Havundersgelser*, 7:8598, 1973.

G. G. Winberg. Rate of metabolism and food requirements of fishes. *Fish. Res. Bd. Can. Trans. Ser.*, 194:1–202, 1956.

Lai Zhang, Uffe Høgsbro Thygesen, Kim Knudsen, and K. H. Andersen. Trait diversity promotes stability of community dynamics. *Theoretical Ecology*, 6(1):57–69, February 2013. ISSN 1874-1738, 1874-1746. doi: 10.1007/s12080-012-0160-6.

Table 1: Parameters in the model with dimensions and "default" values. For a detailed explanation of the determination of the values see (Hartvig et al., 2011, App. E).

Resource spectrum			
κ_R	$5 \cdot 10^{-3}$	$g^{\lambda-1}/m^3$	Magnitude of the resource spectrum
λ	2.05	-	Exponent of resource spectrum ($= 2 - n + q$)
r_0	4	g^{1-p}/yr	Constant for regeneration rate of resources
w_{cut}	0.5	g	Upper weight limit of the resource spectrum
Individual growth			
f_0	0.6	-	Initial feeding level
α	0.6	-	Assimilation efficiency
h	40†	g^{1-n}/yr	Constant for max. food intake
n	0.75	-	Exponent for max. food intake
k_s	4.8†	g^{1-p}/yr	Constant for std. metabolism and activity
p	0.75	-	Exponent of standard metabolism*
β	100	-	Preferred predator-prey mass ratio
σ	1.3¶	-	Width of size selection function
γ	Eq. (3.16)	g^{-q}/yr	Constant for volumetric search rate
q	0.8§	-	Exponent for volumetric search rate
Mortality			
ξ	0.1	-	Fraction of body weight containing reserves
μ_0	3†	g^{1-n}/yr	Constant for background mortality
Reproduction and recruitment			
w_0	0.5	mg	Offspring weight
ϵ	0.1	-	Efficiency of offspring production
κ	50‡	-	Factor for maximum recruitment.

*Laboratory experiments on fish indicate that the exponent of standard metabolism should be higher, around $p = 0.86$ (Killen et al., 2007; Winberg, 1956). The practical implication of choosing $p > n$ is that a maximum weight for individuals at which all energy, even if $f = 1$, is used for standard metabolism at $W_+ = [(\alpha h)/k_s]^{1/(p-n)}$ (Andersen et al., 2008, Eq. 8). Here a value of $p = n$ is used to make the analysis of the model output easier.

†Adjusted to a different value than in (Hartvig et al., 2011) to give growth rates similar to growth rates of species in the North Sea.

¶The width of the selection function is chosen to be larger in the trait-based model than in the species-based model (Hartvig et al., 2011) to emulate the diversity in prey-preferences of the species within a trait-class. The practical implication of enlarging σ is that the model is more stable (fewer oscillations) (Datta et al., 2010).

‡This value should be a little higher but it has been lowered to give a stable output which will facilitate analysis of model output.

§If the value of the search exponent is set to n a lot of the formulas involving exponential factors and power-laws of β are simplified significantly as these factor then just becomes 1. Therefore using $q = n$ is preferred for analytical work.

Column name	Description	Default value
	Life history parameters	
species	Name of the species	Compulsory (no default)
w_inf	The asymptotic mass of the species	Compulsory (no default)
w_mat	Maturation mass. Used to calculate values for ψ. WHAT IS WMAT IN TERMS OF BIOLOGY? Is it the mass at first maturity?	Compulsory (no default)
beta	Preferred predator prey mass ratio	Compulsory (no default)
sigma	Width of prey size preference	Compulsory (no default)
h	Maximum food intake rate. If this is not provided, it is calculated using the `k_vb` column. Therefore, either `h` or `k_vb` must be provided.	Optional (no default)
k_vb	The von Bertalanffy K parameter. Only used to calculate `h` if that column is not provided	Optional (no default)
gamma	Volumetric search rate. If this is not provided, it is calculated using the `h` column and other parameters.	Optional (no default)
ks	Standard metabolism coefficient	`h` * 0.2
z0	Background mortality (constant for all sizes). If this is not provided then `z0` is calculated as $z0pre * w_{\infty}^{z0exp}$. `z0pre` and `z0exp` have default values of 0.6 and -1/3 respectively.	Optional (no default)
k	Activity coefficient	0.0
alpha	Assimilation efficiency	0.6
erepro	Reproductive efficiency	1
w_min	The size class that recruits are placed in.	smallest size class of the species size spectrum
	Fishing gear parameters (see Section 8.3 for more details).	
sel_func	The name of the selectivity function to be used.	"knife_edge".
gear	The name of the fishing gear that selects the species. At the moment a species can only be selected by one gear.	Name of the species
catchability	The catchability of the fishing gear.	1
other columns	Other parameters used by the selectivity function. For example, if the default "knife_edge" function is used then the parameters "knife_edge_size" must also be specified as columns (see Section 8.3).	
	Stock recruitment parameters (see Section 8.4 for more details).	
other columns	Any arguments that appear in the stock-recruitment function must also have a column of values (see Section 8.4)	

Table 2: Columns of the species parameters data.frame

Argument	Description	Default value
min_w	The smallest size of the species community size spectrum. Note that this a different w_min to the one in the species parameter data.frame.	0.001
max_w	The largest size of the species size spectrum.	The largest w_inf in the species parameters data.frame * 1.1
no_w	The number of size bins in the species size spectrum.	100
min_w_pp	The smallest size of the background size spectrum.	1e-10
no_w_pp	The number of size bins in the background size spectrum.	round(no_w) * 0.3
n	The scaling of intake.	2/3
p	The scaling of standard metabolism.	0.7
q	The search volume exponent.	0.8
r_pp	The growth rate of the primary productivity (the background spectrum).	10
kappa	The carrying capacity of the background spectrum.	1e11
lambda	The exponent of the background spectrum.	2+q-n
w_pp_cutoff	The cut off size of the background spectrum.	10
f0	The feeding level of small individuals feeding mainly on the background resource. Used to calculate `h` and `gamma` if they are not provided in the species parameter data.frame.	0.6
z0pre	If `z0`, the mortality from other sources, is not a column in the species data.frame, it is calculated as $z0pre * w_\infty^{z0exp}$.	0.6
z0exp	See z0pre	n - 1

Table 3: Other parameters to the `MizerParams()` constructor

Method	Returns	Description
getSSB()	Two dimensional array (time x species)	Total Spawning Stock Biomass (SSB) of each species through time where SSB is calculated as the sum of weight of all mature individuals.
getBiomass()	Two dimensional array (time x species)	Total biomass of each species through time.
getN()	Two dimensional array (time x species)	Total abundance of each species through time.
getFeedingLevel()	Three dimensional array (time x species x size)	Feeding level of each species by size through time.
getM2()	Three dimensional array (time x species x size)	The predation mortality imposed on each species by size through time.
getFMort()	Three dimensional array (time x species x size)	Total fishing mortality on each species by size through time.
getFMortGear()	Four dimensional array (time x gear x species x size)	Fishing mortality on each species by each gear at size through time.
getYieldGear()	Three dimensional array (time x gear x species)	Total yield by gear and species through time.
getYield()	Two dimensional array (time x species)	Total yield of each species across all gears through time.

Table 4: Summary methods for *MizerSim* objects.

Method	Returns	Description
getProportionOfLargeFish()	A vector with values at each time step.	Calculates the proportion of large fish through time. The threshold value can be specified. It is possible to calculation the proportion of large fish based on either length or weight.
getMeanWeight()	A vector with values at each saved time step.	The mean weight of the community through time. This is calculated as the total biomass of the community divided by the total abundance.
getMeanMaxWeight()	Depends on the `measure` argument. If `measure` = "both" then you get a matrix with two columns, one with values by numbers, the other with values by biomass at each saved time step. If `measure` = "numbers" or "biomass" you get a vector of the respective values at each saved time step.	The mean maximum weight of the community through time. This can be calculated by numbers or by biomass. See the help file for more details.
getCommunitySlope()	A data.frame with four columns: time step, slope, intercept and R^2 value.	Calculates the slope of the community abundance spectrum through time by performing a linear regression on the logged total numerical abundance and logged body size.

Table 5: Indicator methods for *MizerSim* objects.

Plot	Description
plotBiomass()	Plots the total biomass of each species through time. A time range to be plotted can be specified. The size range of the community can be specified in the same way as the method `getBiomass()`.
plotSpectra()	Plots the abundance (biomass or numbers) spectra of each species and the background community. It is possible to specify a minimum size which is useful for truncating the plot.
plotFeedingLevel()	Plots the feeding level of each species against size.
plotM2()	Plots the predation mortality of each species against size.
plotFMort()	Plots the total fishing mortality of each species against size.
plotYield()	Plots the total yield of each species across all fishing gears against time.
plotYieldGear()	Plots the total yield of each species by gear against time.
plot()	Produces 5 plots (`plotFeedingLevel()`, `plotBiomass()`, `plotM2()`, `plotFMort()` and `plotSpectra()`) in the same window as a summary.

Table 6: Plot methods for *MizerSim* objects.