Phase plane analysis and parameter estimation in R

Rob J de Boer, Theoretical Biology, Utrecht University, 2024

This tutorial describes an R-script, `grind.R`, that allows students and investigators whom are not very familiar with the R language to perform phase plane analysis and parameter fitting. Phase plane analysis is powerful graphical method to analyze the steady states of low-dimensional ODE models, and fitting ODE models to data has become a very common practice in biology. Grind is a "wrapper" around the commonly-used R-packages `deSolve`, `FME` and `rootSolve` developed by Karline Soetaert and colleagues [1–4]. Our aim with developing `grind.R` was to define five easy-to-use functions:

- `run()` integrates a model numerically and provides a time plot or a trajectory in the phase plane,
- `plane()` draws nullclines and can provide a vector field or phase portrait,
- `newton()` finds steady states and can provide the Jacobian with its eigenvalues and eigenvectors.
- `continue()` performs parameter continuation of a steady state, providing a bifurcation diagram,
- `fit()` fits a model to data by estimating its parameters, and depicts the result in a timeplot.

Grind's `run()` function calls `ode()` from the `deSolve` library, the `fit()` function calls `modFit()` from the `FME` library, and `newton()` and `continue()` call `steady()` from the `rootSolve` library. One can get help for the library functions by typing `?...`, where `...` is the name of the function (e.g., `?ode`). In RStudio all options of the Grind functions are given while typing their name, e.g., typing `run(` highlights all possible options of the `run()` function. Below we provide all options in an alphabetical order. The following sections are tutorials illustrating the usage of Grind. Instructions for installation are given in the final section.

1 Phase plane analysis

Lotka Volterra model. The ODEs of the model are defined in the simple notation defined for the `deSolve` package. The following is an example of the Lotka Volterra model, here defined by the function `model()`. This R-script is available as the file `lotka.R` on the website `http://tbb.bio.uu.nl/rdb/grindR/`:

```
model <- function(t, state, parms) {
  with(as.list(c(state,parms)), {
    dR <- r*R*(1 - R/K) - a*R*N
    dN <- c*a*R*N - delta*N
    return(list(c(dR, dN)))
  })
}
p <- c(r=1,K=1,a=1,c=1,delta=0.5) # p is a named vector of parameters
s <- c(R=1,N=0.01)                # s is the state
```

where the two lines below the function define the parameter values in the vector `p`, and the initial state of the variables in the vector `s`. Note that the function needs a current `state` and returns a list of derivatives `(dR,dN)`. **Since this is just a list of numbers, these derivatives should be given in the same order as the variables in the state**. The names `model`, `s`, and `p` are the default designations for the model, state, and parameter values in all Grind functions. This example should be self explanatory as it just defines the Lotka Volterra model $\mathrm{d}R/\mathrm{d}t = rR(1 - R/K) - aRN$, and $\mathrm{d}N/\mathrm{d}t = caRN - \delta N$, with its parameter values and initial state as R-vectors, `p <- c(r=1,K=1,a=1,c=1,delta=0.5)`, and `s <- c(R=1,N=0.01)`, respectively. A shiny interactive primer on this model is available on the website: https://grind.shinyapps.io/lotka/.

The following tutorial is an example session illustrating the usage of the first four Grind functions, by simulating and analyzing this Lotka Volterra model; see Fig. 1 for its graphical output (where the `main="text"` add a title to a panel) :

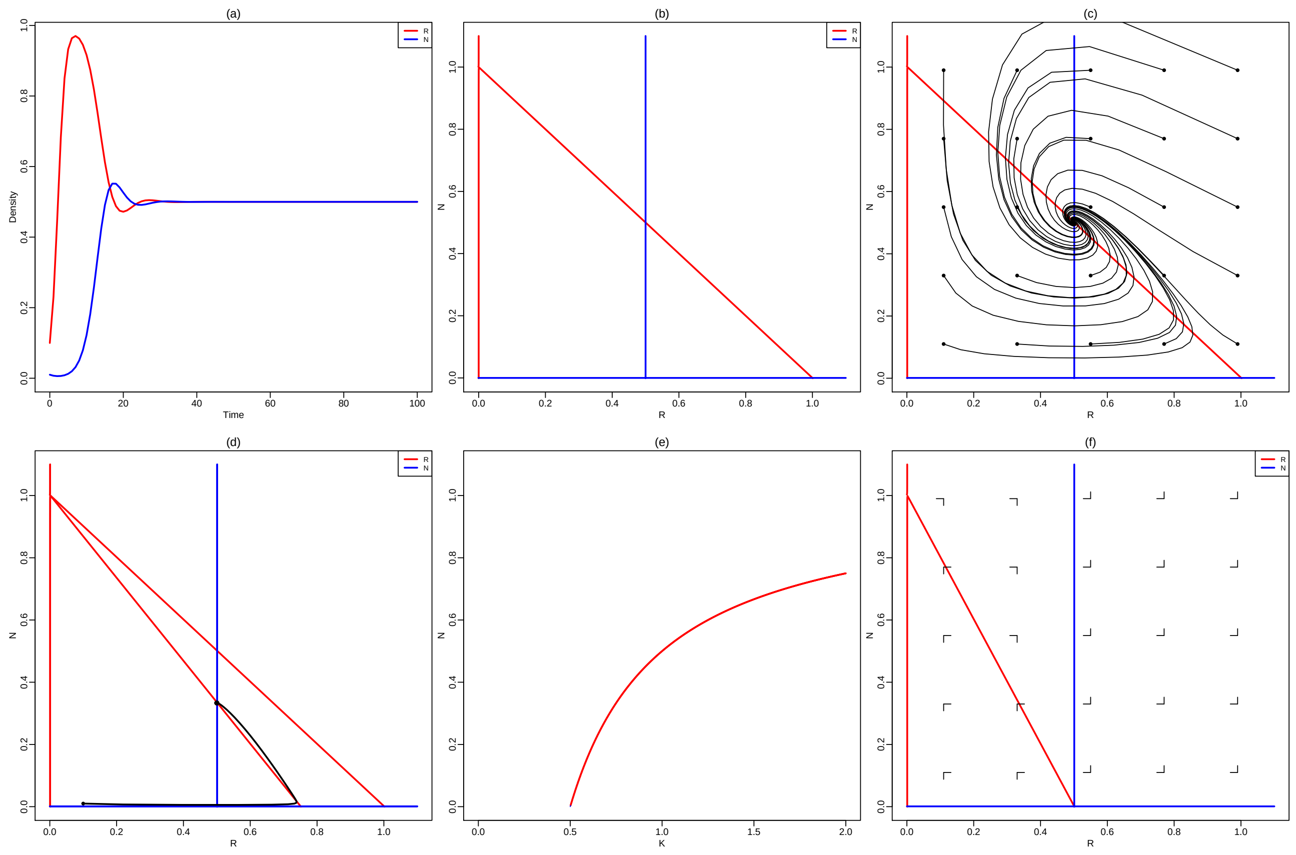

Figure 1: Numerical integration, phase plane analysis, and a bifurcation diagram of the Lotka Volterra model. The six panels collect the graphical output of the example session listed above.

`run(main="(a)")`	run the model and make a time plot (Fig 1a)
`plane()`	make a phase plane with nullclines
`plane(xmin=-0.001,ymin=-0.001,main="(b)"))`	include the full axis in the phase plane (Fig 1b)
`plane(tstep=0.5,portrait=TRUE,main="(c)")`	make a phase portrait (Fig 1c)
`plane(xmin=-0.001,ymin=-0.001,main="(d)")`	make a clean phase plain again (Fig 1d)
`p["K"] <- 0.75`	change the parameter K from 1 to 0.75
`plane(add=TRUE)`	add the new nullclines (see Fig 1d)
`s["R"] <- 0.1`	change the initial state to (R=0.1,N=0.01)
`run(traject=TRUE)`	run the model and plot a trajectory (see Fig 1d)
`newton(c(R=0.5,N=0.5),plot=TRUE)`	find a steady state around (R=0.5,N=0.5); see the bullet in Fig 1d
`f <- newton(c(R=0.5,N=0.5))`	store this steady state in f
`continue(f,x="K",xmax=2,y="N",,main="(e)")`	continue this steady state while varying K (Fig 1e)
`continue(f,x="K",xmax=2,y="N",step=0.001)`	get a better value with a smaller step size
`p["K"] <- 0.5`	set K to the value at which N goes extinct
`plane(vector=TRUE,main="(f)")`	make a phase plane for this value of K (Fig 1f)

Lac-operon model. A slightly more sophisticated example shows how one can continue steady states to make a bifurcation diagram with a saddle-node bifurcation. We use a Lac-operon model defined in the reader of our Systems Biology course at Utrecht University. The phase plane of this example is displayed in Fig. 2a, where bullets indicate stable steady states and circles depict unstable equilibria. Subsequently, we start close to each of the three steady states, call the Newton-Raphson algorithm, and store these states in the variables `low`, `mid`, and `hig`, respectively. Using parameter continuation in Fig. 2b we make a bifurcation diagram in which we follow the middle steady state as a function of the external lactose concentration, L, by a call to `continue(mid,...)`. Consider the following R-script:

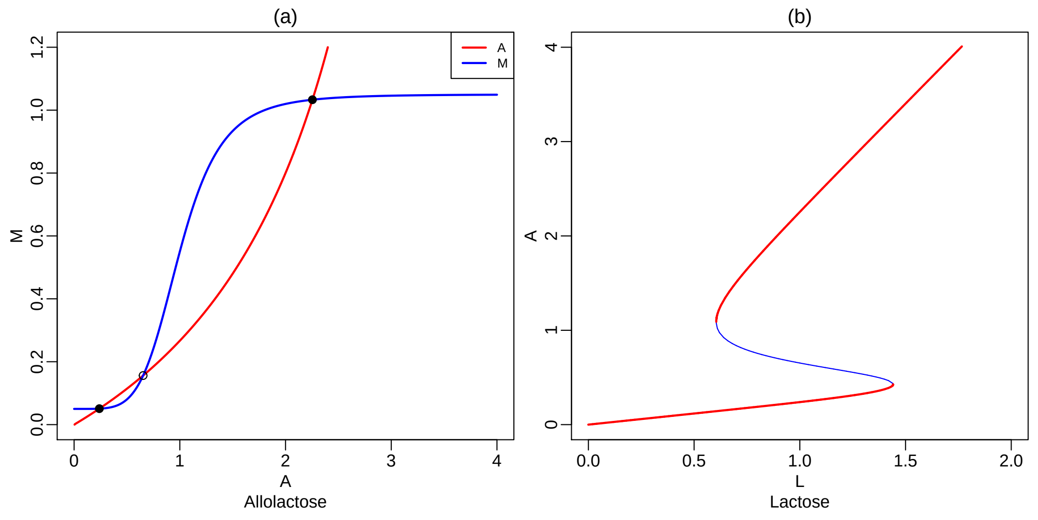

Figure 2: Nullclines and a bifurcation diagram of the Lac-operon model.

```
model <- function(t, state, parms) {
  with(as.list(c(state,parms)), {
  R = 1/(1+A^n)               # Repressor
  dA = M*L - delta*A - v*M*A  # Allolactose
  dM = c0 + c*(1-R) - d*M     # mRNA
  return(list(c(dA, dM)))
  })
}

p <- c(L=1,c=1,c0=0.05,d=1,delta=0.2,n=5,v=0.25)
s <- c(A=0,M=0)
plane(xmax=4,ymax=1.2,main="(a)",sub="Allolactose")
low <- newton(s,plot=TRUE)
mid <- newton(c(A=0.8,M=0.2),plot=TRUE)
hig <- newton(c(A=2,M=1),plot=TRUE)
continue(mid,x="L",y="A",xmax=2,ymax=4,main="(b)",sub="Lactose")
```

These two tutorials should be a sufficient introduction for standard phase plane analyses. The following sections illustrate the usage of events and noise (Section 2), delay differential equations (Section 3), vectors of equations (Section 4), and parameter estimation (Section 5). These can be read when needed. Section 6 is a reference manual that can be consulted when required. Section 7 provides simple installation instructions.

2 Combining numerical integration with events

The `deSolve` package allows one to execute discrete events that are changing the state while integrating the model numerically by using the `events` argument (see the `ode()` manual). This remains possible in Grind because `run()` passes additional options to `ode()` via the ellipsis (`...`) argument.

Grind additionally has simple option `after` to handle events, or change parameters, after each time step. For instance one can set variables to zero when they become negative by `after="state <- ifelse(state<0, 0, state)"` (to generally prevent negative variables in the model one should add `state <- ifelse(state < 0, 0, state)` above the `with`-statement in the model). Another example is `after="parms[\"r\"] <- rnorm(1,1,0.1)"`, which sets the parameter r to a random value, drawn from a normal distribution with a mean of one and a standard deviation of 0.1 (see the result in Fig. 3a). Note that `p` is called `parms` within `run()` (see the Manual), and note the backslashes in `\"r\"` before the quotes around the parameter name. Since r is the first parameter, one can also just write `"parms[1]<-rnorm(1,1,0.1)"` to achieve the same effect. This random resetting of r is done every timestep (as defined by the parameter `tstep` in `run()`). Another example adds Gaussian noise

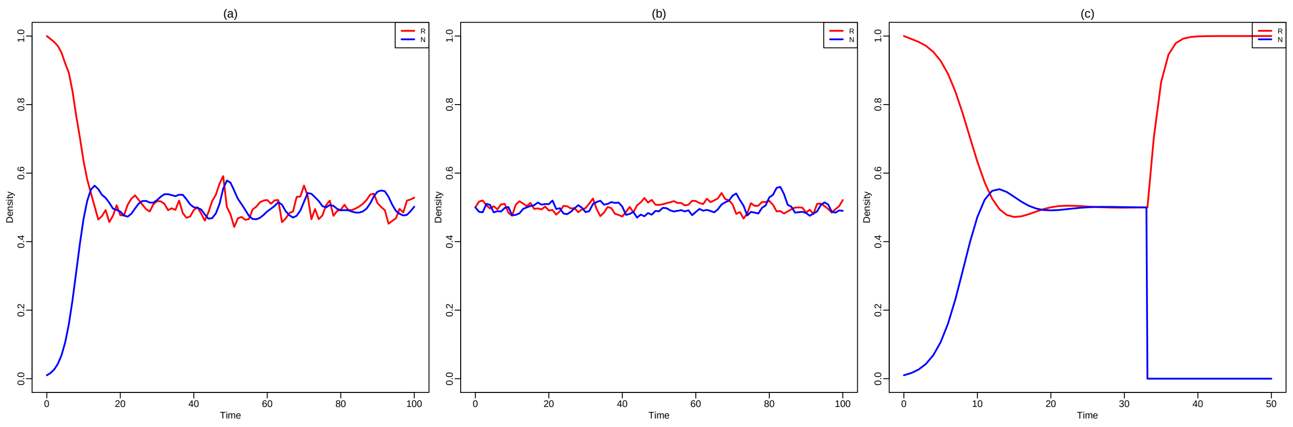

Figure 3: Three examples using `after` and/or `arrest`.

to both variables, e.g., `after="state<-state+rnorm(2,0,0.01)"`, after each time step (see Fig. 3b). The `rnorm(2,0,0.01)` provides two random values, that are added to the two variables, respectively. In Fig. 3b we start close to the steady state to prevent problems arising from random values setting a population to a negative value.

One can also handle events within time steps using the option `arrest`, which stops the integrator at the requested time points. Combining, `after` and `arrest` one can create events, or change parameters at any time point, e.g., by `run(50,arrest=33.14,after="if(t==33.14)state[\"N\"]<-0")` one can set the predators $N = 0$ at $t = 33.14$ (see Fig. 3c). Note again that `s` is called `state` in `run()`, and mind the backslashes in `\"N\"`. `arrest` should also be used if there are time-dependent `if`-statements in the model, like `k<-ifelse(t<tau,1,0)`, which sets $k = 1$ at all times before $t = \tau$ and sets $k = 0$ otherwise (here one should supply `after="tau"` to `run()` or `fit()` to make sure this switch happens at the correct point in time). `arrest` can be a vector of time points or a vector of parameter names. It works by creating dummy events using the `events`-handling of `deSolve()`.

These examples are illustrated by the following R-script making the Panels in Fig. 3:

```
model <- function(t, state, parms) {
  with(as.list(c(state,parms)), {
    dR <- r*R*(1 - R/K) - a*R*N
    dN <- c*a*R*N - delta*N
    return(list(c(dR, dN)))
  })
}

p <- c(r=1,K=1,a=1,c=1,delta=0.5)
s <- c(R=1,N=0.01)

run(after="parms[\"r\"]<-rnorm(1,mean=1,sd=0.1)")

f <- newton(c(R=0.5,N=0.5))
run(state=f,after="state<-state+rnorm(2,mean=0,sd=0.01)",ymax=1)

# Use arrest to handle events at specific time points:
run(50,arrest=33.14,after="if(t==33.14)state[\"N\"]<-0")

# Here is an example of similar event handling in deSolve:
fun<-function(t, y, parms){y["N"]<-0;return(y)}
run(events=list(func=fun,time=33.14))
```

A final "event" to tweak the numerical data. To afterwards modify the numerical solution data computed within `run()` one can pass on any R-command as a text with the `tweak` option. For instance, `run(tweak="nsol$T<-nsol[,2]+nsol[,3]")`, adds a new column to the solution by summing the first and second variable, and calls this column "T" (for total). Note that the numerical solution is called `nsol`, and that the first column contains the time. This manipulated `nsol` table is subsequently passed

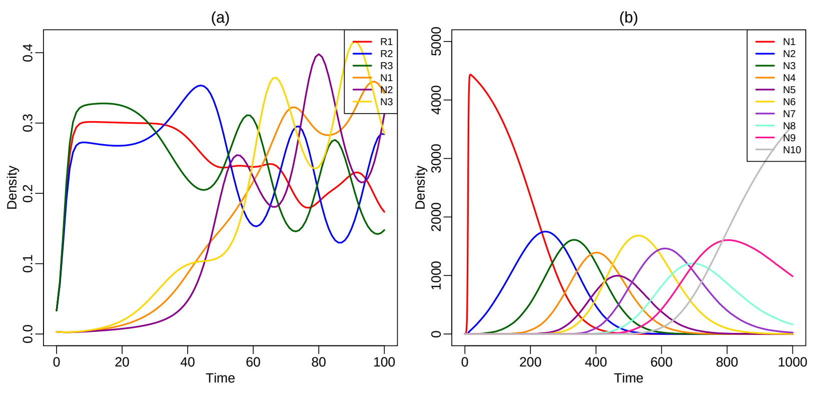

Figure 4: Left: the graphical output of the "vector of equations" example. Right: the output of the examples of mutating strains.

on to `timePlot`, or printed to screen (with the `table=TRUE` option in `run()`). This `tweak` option can be very helpful when fitting data containing columns representing (transformed) combinations of the variables of the model. The same summation can also be done by using R's `apply()` to call R's `sum()` function: `run(tweak="nsol$T<-apply(nsol[,2:3],1,sum))"`, illustrating how one can use `tweak` to sum over a subset of columns, or to call any other function.

3 Delay differential equations and maps

One can study maps by simply switching to Euler integration:

```
model <- function(t, state, parms) {
  with(as.list(c(state,parms)), {
    dN <- r*N*(1 - N) - N
    return(list(c(dN)))
  })
}
p <- c(r=3.75)
s <- c(N=0.01)
data <- run(1000,method="euler",table=TRUE)
plot(data$N[1:999],data$N[2:1000],pch=".")
```

Be careful with using the steady state functions (`newton()`, `continue()`, and `plane()`) because they are not aware of the fact that this model defines a map. Steady state values of this model will be correct because of the `-N` in the equation (i.e., `dN=0` has the same solution as $N_{t+1} = rN_t(1 - N_t)$), but the reported eigenvalues no longer define the stability of steady states.

One can study delay differential equations (DDEs) because the `deSolve` package implements a general solver (`dede()`) using the same syntax as the general ODE solver (`ode()`). For instance, the Lotka Volterra model with delayed growth of the predator,

$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = rR(t)(1 - R(t)/K) - aR(t)N(t) \ , \quad \text{and} \quad \frac{\mathrm{d}N(t)}{\mathrm{d}t} = caR(t-\Delta)N(t-\Delta) - \delta N(t) \ ,$$

would look like:

```
model <- function(t, state, parms) {
  with(as.list(c(state,parms)), {
    tlag <- t - Delta
    if (tlag < 0) lags <- c(0,0)    # no initial predation
    else lags <- lagvalue(tlag)     # returns lags of R and N
    dR <- r*R*(1 - R/K) - a*R*N
    dN <- a*lags[1]*lags[2] - d*N
    return(list(c(dR, dN)))
  })
}

p <- c(r=1,K=1,a=1,c=1,d=0.5,Delta=10)
s <- c(R=1,N=0.1)
run(delay=TRUE)
```

where the option `delay` tells Grind to use the `dede()` solver. This would correspond to the situation where a prey at carrying capacity starts to be eaten by a predator at time zero, but the predator will only start to grow `Delta` time steps later. The `deSolve` function `lagvalue()` stores previous values of R and N in a vector (that is called `lags[]` here) that can be indexed to obtain the time lagged R and N values. Read the `deSolve` manual for further documentation and/or type `?dede` for help. Please note that the steady state functions (`newton()`, `continue()`, and `plane()`) should not be called with models containing calls to `lagvalue()`. Fitting DDE models to data should work fine (but solving DDEs is notoriously tricky).

4 Vectors of equations

Here is an example of a model with $n = 3$ prey populations, R_i, that are competing with each other via a logistic term (see the left panel in Fig. 4). Each prey has its own predator N_i,

$$\frac{\mathrm{d}R_i}{\mathrm{d}t} = b_i R_i(1 - \sum R_j) - d_1 R_i - aR_iN_i \quad \text{and} \quad \frac{\mathrm{d}N_i}{\mathrm{d}t} = aR_iN_i - d_2N_i \ .$$

Here is an R-script drawing random prey birth rates, b_i, from a normal distribution:

```
model <- function(t, state, parms){
  with(as.list(c(state,parms)),{
    R <- state[1:n]
    N <- state[(n+1):(2*n)]
    S <- sum(R)
    dR <- b*R*(1-S) - d1*R - a*R*N
    dN <- a*R*N - d2*N
    return(list(c(dR,dN)))
  })
}

n <- 3                                    # number of species
b <- rnorm(n,mean=1,sd=0.1)               # b is a global parameter
p <- c(d1=0.1,d2=0.2,a=1)                 # other parameters
R <- rep(0.1/n,n)                         # initial condition of R
names(R) <- paste("R",seq(1,n),sep="")    # Name them R1, R2, ... Rn
N <- rep(0.01/n,n)                        # initial condition of N
names(N) <- paste("N",seq(1,n),sep="")    # Name them N1, N2, ... Nn
s <- c(R,N)                               # combine R and N into s
run(main="(a)")
```

The result is show in Fig. 4a.

Mutations after each time step. Combining vectors and events one can model a series of evolving bacterial strains with increasing replication rates (see Fig. 4b). Consider the following model,

$$\frac{\mathrm{d}N_i}{\mathrm{d}t} = b_i N_i(1 - \sum N_j) - dN_i \ , \quad \text{for} \quad i = 1, 2, \ldots, n$$

and let strain N_{i+1} evolve from strain N_i at a mutation rate μ. When the expected number of mutants, μN_i, is smaller than a single bacterium we calculate the probability that a single mutant appears, and add a single cell to N_{i+1} (and subtract it from N_i). Otherwise the expected number of mutants is added to strain N_{i+1} (and subtracted from N_i). This is realized by using `after` to call the function `mutate()`. In the following R-script birth rates b_i of the strains increase linearly from $b_i = 1$ to $b_n = 2$:

```
model <- function(t, state, parms){
  with(as.list(c(state,parms)),{
    N <- state[1:n]
    S <- sum(N)
    dN <- b*N*(1-S/K) - d*N
    return(list(c(dN)))
  })
}

mutate <- function(t, state, parms){
  nmut <- rep(0,n+1)
  emut <- parms["mu"]*state                                   # Expected number of mutants
  nmut[2:(n+1)] <- ifelse(emut>1,emut,ifelse(runif(n)<emut,1,0))
  state <- state + nmut[1:n] - nmut[2:(n+1)]
  return(state)
}

n <- 10                                                     # number of variants with
b <- seq(1,2,length=n)                                      # increasing birth rates
p <- c(d=0.1,K=5000,mu=0.001)                               # other parameters
s <- rep(0,n)                                               # set all variables to 0
s[1] <- 1                                                   # set first to 1
names(s) <- paste("N",seq(1,n),sep="")                      # add names
run(1000,ymax=5000,after="state<-mutate(t,state,parms)",main="(b)")
```

5 Parameter estimation

One can fit the parameters of a model to data using the function `fit()`, which minimizes the sum of the squared residuals (SSR) between the data and the model. When this function is called without any options it is assumed that there is `data.frame` called `data` with column names corresponding to the variables of the model, and `fit()` will then use the state, `s`, and all parameters, `p`, as an initial guess for fitting the data. We illustrate this using the Lotka Volterra model after creating an artificial data set using `run(table=TRUE)`. In the example we randomize the initial condition, `s`, and parameters, `p`, using a normal distribution (`rnorm()`) before we fit the model to the data (see Fig. 5A).

`fit()` internally calls the function `modFit` from the `FME` package (use `?modFit` to see all options), and delivers an object providing the estimated parameters, confidence ranges, and correlations between the parameters. Typing `fit()$par` just returns the estimated parameters (see below). Use the option `free` to explicitly define which parameters are "free" and should be estimated (`free` is a vector of names). The boolean option `initial` can be used to read the initial condition from the data (instead of estimating it).

One can fit simultaneously several data sets by providing a list of data sets to the first `datas` option (see the `fit(list(dataR,dataN))` example). When fitting several data sets, we need to better distinguish between free and non-free parameters, because they could differ between the data sets or be the same. Grind defines free parameters that are shared between all data sets as `free`, and free parameters that differ between the data sets as `differ`. Both are vectors of parameter names (and Grind makes sure they do not overlap). Grind defines non-free parameters that differ between the data sets as `fixed`. Hence all parameters that are not in the vectors `free`, `differ` or `fixed`, are not fitted and are shared between the data sets. Note that fitting several data sets together actually requires Maximum Likelihood procedures that are not implemented in Grind, because different data sets may have different amplitudes and variances. Using Grind's Least squares procedures for fitting data sets together therefore assumes that they all have the same "nature", such that the goodness of a fit can be described as the sum of their individual SSRs.

The usage of `free` and `differ` is illustrated in the `f <- fit(list(data,data2), free=free, differ =c("R","N","K"))` example given below. In this example `differ` is just a vector of parameter names. In case one needs to supply an initial guess for the different parameters in each data set, one should define `differ` as a named list, containing the various guesses (see the `differ<-list()` example below). The usage of `fixed` is illustrated in the lines below `fixed<-list()`.

Finally, we show how one can bootstrap the data by sampling (with recruitment) from every individual data set, and re-fit the samples using the best parameters as an initial guess. Another important option is `logpar` which allows one to fit the logarithm of the parameters (rather than their true values), which can help finding the true optimum when parameters have very different values. If worried about finding the true optimum, one can pass the option `method="Pseudo"` to `modFit` (see the `modFit` manual).

Consider the following script:

```
model <- function(t, state, parms) {
  with(as.list(c(state,parms)), {
    dR <- r*R*(1 - R/K) - a*R*N
    dN <- c*a*R*N - delta*N
    return(list(c(dR, dN)))
  })
}

p <- c(r=1,K=1,a=1,c=1,delta=0.5)
s <- c(R=1,N=0.01)
data <- run(20,table=T)        # Make a data set

s <- s*abs(rnorm(2,1,0.1));s   # Random guess for initial condition
p <- p*abs(rnorm(5,1,0.1));p   # Random guess for parameters
f <- fit()                     # Fit data with all 7 parameters free
summary(f)                     # Check confidence ranges, etcetera
p <- f$par[3:7];p              # Store estimates in p

p <- p*abs(rnorm(5,1,0.1));p   # Another random guess for the parameters
free <- c(names(s),names(p))   # Provides the names of free parameters
f <- fit(data,free=free)       # Fit the data again

dataR <- data; dataR$N <- NULL        # Make two data sets one with R,
dataN <- data; dataN$R <- NULL        # and the other with N,
f <- fit(list(dataR,dataN),free=free) # which gives the same result

p <- c(r=1,K=1,a=1,c=1,delta=0.5)  # Start again with same parameters
p["K"] <- 0.75                     # Change K,
s <- c(R=0.5,N=0.05)               # and the initial condition,
data2 <- run(25,table=T)           # and make a new data set
s <- c(R=0.75,N=0.02)              # An "average" guess for the 2 initial conditions
p <- p*abs(rnorm(5,1,0.1));p       # A random guess for the parameters
free <- names(p)[-2]               # Remove K from the free parameters
par(mfrow=c(1,2))                  # Show two panels next to each other
f <- fit(list(data,data2),free=free,differ=c("R","N","K"),main=c("A","B"))
f$par                              # Show parameters only

# Provide individual initial guesses as a list:
differ <- list(R=c(0.9,0.55),N=c(0.02,0.04),K=c(1.1,0.7))
f <- fit(list(data,data2),free=free,differ=differ,main=c("A","B"))

# Provide fixed parameters as a list:
fixed <- list(R=c(1,0.5),N=c(0.01,0.05))
differ <- "K"                               # one unknown parameter (K)
free <- names(p)[-2];free                   # and four shared unknown parameters
f <- fit(list(data,data2),free=free,differ=differ,fixed=fixed,main=c("A","B"))

# The latter is identical to taking the initial condition from the data:
f <- fit(list(data,data2),free=free,differ=differ,initial=T,main=c("A","B"))

# Finally perform a 100 bootstrap simulations:
f <- fit(list(data,data2),free=free,differ=differ,fixed=fixed,main=c("A","B"),bootstrap=100)
```

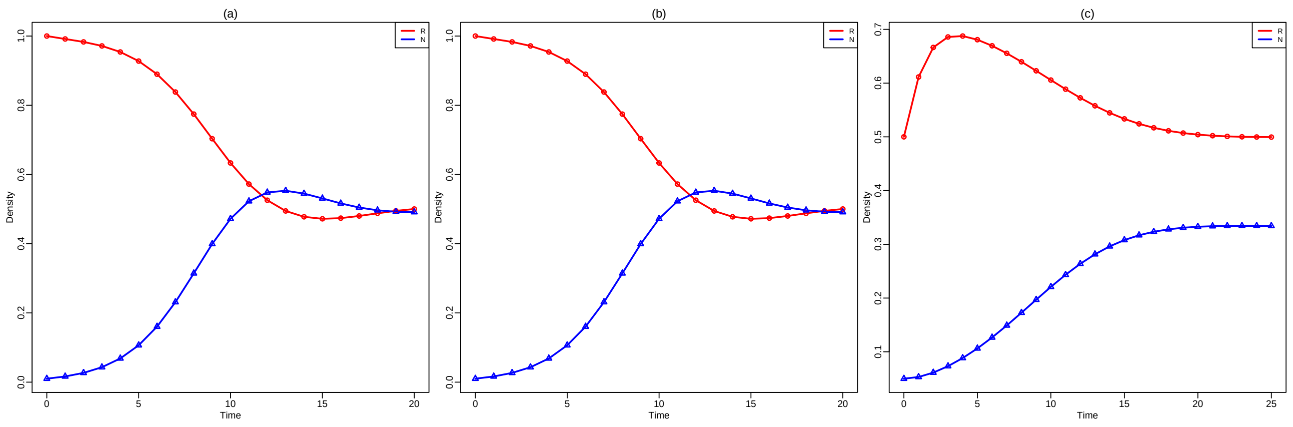

Figure 5: Fitting the Lotka Volterra model to data. Lines show the model for the best estimated parameters, and the symbols depict the data.

6 Manual

A model can be solved numerically from the initial state by calling `run()`, and the output will be a timeplot, trajectory or table. Next to the graphics output, `run()` returns the final state attained by the simulation (or all data when `table=TRUE`). The former can be helpful if one wants to continue from the previous state (e.g., `f<-run(); f<-run(state=f)`). The full definition of `run()` is:

```
run <- function(tmax=100, tstep=1, state=s, parms=p, odes=model, ymin=0, ymax=NULL,
log="", xlab="Time", ylab="Density", tmin=0, draw=lines, times=NULL, show=NULL,
arrest=NULL, events=NULL, after=NULL, tweak=NULL, timeplot=TRUE, traject=FALSE,
table=FALSE, add=FALSE, legend=TRUE, solution=FALSE, delay=FALSE, lwd=2,
col="black", pch=20, ...)
```

`run()` calls the `ode()` function from the `deSolve` package. Additional arguments, like `main` and `sub`, are passed on to `ode()` and `plot()` via the (...).

The phase plane function `plane()` sets up a space with the first variable on the horizontal axis, and the second on the vertical axis. The full definition of `plane()` is:

```
plane <- function(xmin=0, xmax=1.1, ymin=0, ymax=1.1, xlab="", ylab="", log="",
npixels=500, state=s, parms=p, odes=model, x=1, y=2, time=0, grid=5, eps=0,
show=NULL, addone=FALSE, portrait=FALSE, vector=FALSE, add=FALSE, legend=TRUE,
zero=TRUE, lwd=2, col="black", pch=20, ...)
```

Additional arguments (...) are passed on to `run()` (for the phase portrait) and to `plot()` (e.g., for `main` and `sub`). Note that `plane()` calls the “vectorized” R-function `outer()`, which implies that if one calls functions in the ODEs they should also be vectorized, e.g., one should use `pmax()` instead of `max()`. There is an extension, `cube.R`, for 3-dimensional nullclines.

The function `newton()` finds a steady state from a nearby initial state, and can report the Jacobi matrix with its eigenvalues and eigenvectors. The full definition of `newton()` is:

```
newton <- function(state=s, parms=p, odes=model, time=0, positive=FALSE,
jacobian=FALSE, vector=FALSE, plot=FALSE, silent=FALSE, addone=FALSE, ...)
```

`newton()` calls the function `steady()` from the `rootSolve` package (which calls `stode()`). Additional arguments (...) are passed on to both of them. `newton()` needs an initial state close to an equilibrium point.

The function `continue()` continues a steady state by changing a “bifurcation” parameter defined by the horizontal axis of the bifurcation diagram. The full definition of `continue()` is:

```
continue <- function(state=s, parms=p, odes=model, step=0.01, x=1, y=2,
time=0, xmin=0, xmax=1,ymin=0, ymax=1.1, xlab="", ylab="", log="",
col=c("red","black","blue"), lwd= c(2,1,1), addone=FALSE, positive=FALSE,
nvar=FALSE, add=FALSE, ...)
```

`continue()` calls the function `steady()`from the `rootSolve` package (additional arguments (...) are passed on), and needs an initial state close to an equilibrium point. Note that there is much more proper software for bifurcation analysis like XPPAUT or MatCont, which reports the type of bifurcations encountered, and automatically continues all branches of branch points. Additionally, `deBif` is a Grind-like R-package allowing for interactive bifurcation analysis using a shiny-based interface.

The function `fit()` fits a model to data by non-linear parameter estimation. The output is an object containing the estimated parameters, the summed squared residuals, confidence ranges, and correlations (see the `modFit()` manual). The data and the model behavior for its best fit parameters are depicted in a timeplot. Its full definition is:

```
fit <- function(datas=data, state=s, parms=p, odes=model, free=NULL, differ=NULL,
fixed=NULL, tmin=0, tmax=NULL, ymin=NULL, ymax=NULL, log="", xlab="Time",
ylab="Density", bootstrap=0, show=NULL, fun=NULL, costfun=cost, logpar=FALSE,
lower=-Inf, upper=Inf, initial=FALSE, add=FALSE, timeplot=TRUE, legend=TRUE,
main=NULL, sub=NULL, pchMap=NULL, ...))
```

`fit()` calls the function `modFit()` from the `FME` package (which calls `modCost()`). Additional arguments (...) are passed on to both of them, and to `run()` and `ode()`.

Finally the internal function `timePlot()` can be used to plot data and is defined as:

```
timePlot <- function(data, tmin=0, tmax=NULL, ymin=0, ymax=NULL, log="",
xlab="Time", ylab="Density", show=NULL, legend=TRUE, draw=lines, lwd=2, add=FALSE,
main=NULL, sub=NULL, colMap=NULL, pchMap=NULL, ...)
```

These functions have many options (arguments). Fortunately most of them have good default values, and can typically be omitted. Since many options are the same in the five Grind functions, we list them alphabetically, giving their default values, and a short explanation:

- `add=FALSE` defines whether a new plot should be created when drawing new graphics. When `add=TRUE` Grind assumes the definition of the axes have not been changed. Used by `continue()`, `run()`, and `plane()`.
- `addone=FALSE` allows for a "zero" on logarithmic axis (like the `log1p()` function in R). Setting `addone=TRUE` and `xmin` or `ymin` to one (or 0.99) allows one to include steady states in which one of the variables is zero in nullclines and parameter continuations. Used by `plane()`, `continue()`, and `newton()`.
- `after=NULL` defines commands to be executed after each time step of a numerical integration, e.g., `after="parms[1]<-rnorm(1,1,0.1)"`, draws a new random value of the first parameter after each time step (as defined by `tstep`). See Section 2; used by `run()` and `fit()`.
- `arrest=NULL` defines a vector of values, or parameter names, defining time points where the integrator should stop, and report the current state (i.e., these time points are added to the `times` vector of `ode()`). This can be helpful for running (and fitting) models with `if` statements, or other discontinuous time points, e.g., `arrest=c("T1","T2")`). Used by `run()` and `fit()`.
- `bootstrap=0` defines the number of samples to be taken randomly from the data (with replacement). This prints a summary and adds an element `bootstrap` to the `modFit` list, containing a matrix with all parameter estimates. Use `pairs(f$bootstrap)` to see the correlations between the estimates. Used by `fit()`.
- `col="black"` in `run()` and `plane()` defines the color of trajectories.
- `col=c("red","black","blue")` in `continue()` defines the color of stable, neutral, and unstable steady states, respectively. Alternatively, when `nvar=TRUE` one can use this option to color a steady state by its number of non-zero variables (e.g., `col=colors` would do if you like Grind's default colors).

- `colMap=NULL` is vector of integers used to re-map the colors, e.g., `colMap=c(3,2,1)` reverts the order of the first three colors. Note that `grind.R` defines its own color table (of dark colors that print well). Used by `continue()`, `plane()` and `run()`.
- `costfun=cost` allows one to redefine the cost-function measuring the distance between the model and the data. This can be useful when different data sets need different models. The default `cost`-function loops over the various data sets, and calls `run()` for each of them. That call can easily be adapted for each data set (the index of the loop is called `iset`). Used by `fit()`.
- `datas=data` in `fit()` defines the name of the data frame containing the data, or defines a list of data frames.
- `delay=FALSE` tells Grind whether or not to call the DDE solver from the `deSolve` package. The time delay(s) in the model are to be defined by calling the `lagvalue()` function from the `deSolve` package. This can only be used in `run()` and `fit()` and should also not be used in combination with phase plane analysis, nor with searching steady states (`newton(), continue()`). See Section 3; used by `run()` and `fit()`.
- `differ=NULL` defines the names of the free parameters that differ between the data sets and need to be fitted separately. `differ` can also be a named list containing the individual guesses for each data set. (One can use `makelist(differ,state,parms,nsets)` to set up such a list). See Section 5; used by `fit()`.
- `draw=lines` draws time plots as continuous lines. The alternative is `draw=points`. This tells `timePlot()` to use the R-functions `lines()` or `points()` to plot the columns of the data frame. Used by `run()` and `fit()`.
- `eps=0` is a dirty little shortcut in `plane()` to include or exclude a zero-axis in the drawing of nullclines. The contour line algorithm used by Grind has difficulties drawing nullclines at the boundaries of the space. `eps` can be used to override the defaults of both `xmin=0` and `ymin=0` to a small value (if the axis is linear). Used by `plane()`.
- `events` is an option that can be passed on to `deSolve` to handle events during numerical integration. Type `events?` for help. Used by `run()`.
- `fixed=NULL` defines a named list of the non-free parameters that differ between the data sets and have known fixed values. (One can use `makelist(fixed,state,parms,nsets)` to set up such a list). See Section 5; used by `fit()`.
- `free=NULL` defines the names of the 'free' parameters to be fitted. By default `free` equals `c(names(state), names(parms))`. See Section 5. Used by `fit()`.
- `fun=NULL` defines a function to transform the data and (numerical) solution before fitting. Used by `fit()`.
- `grid=5` defines the number of grid points for which the vector field or phase portrait is drawn. Used by `plane()`.
- `initial=FALSE` allows one to read the initial condition from the data (and not estimate it). See Section 5. Used by `fit()`.
- `jacobian=FALSE` defines whether or not `newton()` should print the Jacobian.
- `log=""` can be used to define logarithmic axis (this is passed on to the R-function `plot()`). `log="xy"` makes both axes logarithmic. Used by `continue()`, `plane()` and `run()`.
- `logpar=FALSE` can be used to fit the logarithm of the parameters, rather than their true values. This can be very useful when the parameters have very different values. Grind will report the true values that are estimated, but `modFit()` returns the logarithmic values. Used by `fit()`.
- `lower=-Inf` defines the lower limit of the free parameters. One can set `lower=0` to keep all free parameters positive. Otherwise, the order of values in lower should be the same as the names of the parameters in `free`. This becomes complicated when there are shared and different free parameters (as defined by `free` and `differ`). The order in `lower` should then first be the parameters in `free` and then those in `differ`. The latter are automatically repeated for every additional data set if the length of `lower` equals the sum of the lengths of `free` and `differ`. Thus, the length of `lower` is either 1, `length(free)+length(differ)` or `length(free)+n*length(differ)`, where `n` is the number of data sets.
- `lwd=2` sets the line-width of graphs. Used by `continue()`, `plane()` and `run()`.

- `main=NULL` allows one to put a title at the top of the graph (this is passed on to the R-function `plot()` by all Grind functions). In `fit()` main can be a vector for the titles of all plots when multiple data sets are being fitted. Note that titles are set in a plain font (to set that back to R's default bold face, change the two `font` lines in the `grind.R` file). Used by `continue()`, `fit()`, `plane()` and `run()`.
- `npixels=500` defines the resolution of the phase space in `plane()`. Setting it to a lower value speeds up the drawing of nullclines.
- `nvar=FALSE` asks `continue()` to color steady state by their stability. By default stable steady state are depicted as heavy red lines and unstable steady states by thin blue lines (which can be changed with the `col` and `lwd` options of `continue()`). If `nvar=TRUE` the states are colored by their number of non-zero variables (use `col` to define these colors), and their stability remains indicated by the line width.
- `odes=model` defines the name the model. Used by all Grind's functions.
- `parms=p` define the name of the parameter vector. Used by all Grind's functions.
- `pch=20` defines the symbol to be plotted at the start of trajectories. Used by `plane()` and `run()`.
- `pchMap=NULL` can be used to re-map the symbols, e.g., `pchMap=c(3,2,1)` reverts the order of the first three R-symbols (see `pch` in the R-function `points()`). Used by `run()`.
- `plot=FALSE` defines whether or not `newton()` depict the steady state by a symbol in the phase plane.
- `portrait=FALSE` defines whether or not `plane()` should include a phase portrait.
- `positive=FALSE`, setting `positive=TRUE` restricts the search of `newton()` and `continue()` to positive steady states only.
- `silent=FALSE`. By default `newton()` returns a steady state, while printing the eigenvalues. After setting `silent=TRUE`, `newton()` does not print anything, and returns a list containing the state, the Jacobian, the eigenvalues and the eigenvectors. Use `max(Re(newton(s,silent=TRUE)$values))` to retrieve the largest eigenvalue.
- `show=NULL` defines the variables appearing in a time plot, fit, or phase plane. By default all variables are shown. By explicitly providing a list of names one can define subsets to be plotted, e.g., `show=c("P","Q")`. Used by `run()`and `plane()`.
- `solution=FALSE` defines whether or not the model provides time derivatives (default), or a full solution. This can only be used in `run()` and `fit()` and should obviously not be used in combination with phase plane analysis, nor with searching steady states. Models returning a solution obey the same format as the ODE models required by `deSolve` (but can return a vector of values and not necessarily a list). Used by `run()` and `fit()`.
- `state=s` define the name of the state vector. Used by all Grind's functions.
- `step=0.01` defines the maximum change of the bifurcation parameter in a bifurcation diagram. When the axis is linear the parameter is increased, or decreased, in steps not exceeding `step` × `xmax`. When the axis is logarithmic the parameters is maximally multiplied by 1+`step`. `continue()` will decrease the step size to maximally `step`/100 when it looses the steady state. Used by `continue()`.
- `sub=NULL` allows one to put a subtitle at the bottom of the graph (this is passed on to the R-function `plot()`). Note that titles are set in a plain font (to set that back to bold face, change the two `font` lines in the `grind.R` file). Used by `continue()`, `plane()` and `run()`.
- `table=FALSE` asks `run()` to return a table with the values reported by the integrator. Used by `run()`.
- `time=0` defines the time point for which nullclines are computed and steady states are computed (for non-autonomous ODEs). Used by `continue()`, `newton()`and `plane()`.
- `times=NULL` can be used to define the time points at which the numerical integrator generates output (even though this is typically done by setting `tstep`). Used by `run()`.
- `timeplot=TRUE` asks `run()` to make a time plot. Used by `run()`and `fit()`.
- `tmax=100` set the integration time. Used by `run()`and `fit()`.
- `tmin=0` allows one to start a specific time point (which can be convenient when a run is continued). Used by `run()`.
- `traject=FALSE` asks `run()` to draw trajectory in an existing phase plane. Used by `run()`.

- `tstep=1` set the reporting interval of numerical integrations. One can also provide a vector of time points where the intergrator should provide output with the `times` option (see the `ode()` manual). Used by `run()`and `fit()`.
- `tweak=NULL` allows one to modify the data delivered by `run()`. For instance one can add columns that can be fitted to data, e.g., `tweak<-"nsol$T<-nsol$R+nsol$N"` to sum R and N into a new column T. One can also transform predictions from the model before they are fitted to data that is already transformed. Used by `run()` and `fit()`.
- `upper=Inf` defines the upper limit of the free parameters. One can set `upper=10` to give all free parameters a maximum value of 10. Otherwise, the order of values in upper should be the same as the names of the parameters in `free`. This becomes complicated when there are shared and different free parameters (as defined by `free` and `differ`). The order in `upper` should then first be the parameters in `free` and then those in `differ`. The latter are automatically repeated for every additional data set if the length of `upper` equals the sum of the lengths of `free` and `differ`. Thus, the length of `upper` is either 1, `length(free)+length(differ)` or `length(free)+n*length(differ)`, where `n` is the number of data sets.
- `x=1` define the variable on the horizontal axis of phase planes and bifurcation diagrams. One can also use the names of the variables to define the axes, e.g., `x="R"`. Used by `continue()`and `plane()`.
- `xlab="Time"` or `xlab=variable` allows one to redefine the label of a horizontal axes. Used by `continue()`, `plane()` and `run()`.
- `xmax=1.1` define the maximum of vertical axes of time plots phase planes and bifurcation diagrams. Used by `continue()`and `plane()`.
- `xmin=0` define the origin of the horizontal axis of phase planes and bifurcation diagrams. Used by `continue()`and `plane()`.
- `y=2` define the variable on the vertical axis of phase planes and bifurcation diagrams. One can also use the names of the variables to define the axes, e.g., `y="N"`. Used by `continue()`and `plane()`.
- `ylab="Density"` or `ylab=variable` allows one to redefine the label of a vertical axes. Used by `continue()`, `plane()` and `run()`.
- `ymax=1.1` define the maximum of the vertical axis of time plots, phase planes and bifurcation diagrams. Used by `continue()`, `plane()` and `run()`.
- `ymin=0` define the origin of the vertical axis of phase planes and bifurcation diagrams. Used by `continue()`and `plane()`.
- `vector=FALSE` defines whether or not `plane()` should include a vector field.
- `vectors=FALSE` defines whether or not `newton()` should print the eigenvectors.
- `zero=TRUE` draws the phase plane for all variables other than `x` and `y` set to zero (important when drawing nullclines of variables not appearing on the axes). Used by `plane()`.
- `...` can be used to define parameters that are passed on to other functions

7 Installation and startup

The very first time Grind is used one needs to install the Soetaert libraries into the R-environment, e.g., `install.packages(c("deSolve", "rootSolve", "FME")` in R, or `Install Packages` in the `Tools` menu of RStudio. After downloading `grind.R` one can include the grind.R environment by typing `source("grind.R")`. In RStudio one can also open `grind.R` in one tab (and "source" it) and open the model in another tab. The R-scripts can be found

For example, download the R-codes `grind.R` and `lotka.R` from the webpage: `https://tbb.bio.uu.nl/rdb/grindR/`, store them in a local directory, and open them in RStudio. It may be convenient to set the working directory to the folder where the R-codes were stored (`Set working directory` in the `Session` menu of RStudio). First "source" the `grind.R` file (button in right hand top corner) to define the Grind functions. Then click the other tab, and select the function `model()` by highlighting everything up to the closing curly bracket, and execute this by clicking the `Run` button (or typing `Control Enter`). Subsequently proceed through that file by running it line-by-line (using `Control`

`Enter`), to see what is happening, and become familiar with the behavior of the Grind functions.

March 7, 2024, Rob J. de Boer

References

[1] **Soetaert, K.**, 2009. rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations. R package 1.6.

[2] **Soetaert, K. and Herman, P. M.**, 2009. A Practical Guide to Ecological Modelling. Using R as a Simulation Platform. Springer. ISBN 978-1-4020-8623-6.

[3] **Soetaert, K. and Petzoldt, T.**, 2010. Inverse modelling, sensitivity and Monte Carlo analysis in R using package FME. Journal of Statistical Software **33**:1–28.

[4] **Soetaert, K., Petzoldt, T., and Setzer, R. W.**, 2010. Solving differential equations in R: Package deSolve. Journal of Statistical Software **33**:1–25.

A Appendix

A.1 Grind extensions

There is currently only one extension (`cube.R`) which uses the `plot3D` R-library to define a function for plotting 3-dimensional nullclines (`cube()`) and a function for plotting 3-dimensional trajectories (`run3d()`). The syntax of these functions is similar to that of `plane()` and `run()`. See the `cube.R` script for some help on the 3D projection.