Phase plane analysis is powerful graphical method to analyze low-dimensional ODE models. At Utrecht University we have hitherto used dedicated C-code (GRIND) for numerical integration, phase plane analysis, and stability analysis of steady states. Thanks to the R-packages deSolve and rootSolve developed by Karline Soetaert and colleagues [1, 2, 4], it was relatively easy to copy most of GRIND’s capabilities into R. People liking R may also like this simple interface to phase plane analysis. Thanks to the FME package, also developed by Karline Soetaert and colleagues [3], it was also possible to extend GRIND with non-linear parameter estimation. This resulted in an R-script grind.R defining 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 (using the Newton-Raphson method) 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.

The 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 on the grind.R functions by typing args(run), etcetera. For the library functions one can get more help by typing ?ode, etcetera. The following sections are tutorials illustrating the usage of the grind.R functions.

## 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/practicals/grindR/:

```r
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)))
  })
}
```

```r
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 returns a list of derivatives (dR,N). The names model, s, and p are the default designations for the model, state, and parameter values in all grind.R functions. This example should be self explanatory as it just defines the Lotka Volterra model \( \frac{dR}{dt} = rR(1 - R/K) - aRN \), and \( \frac{dN}{dt} = caN - \delta N \), with its parameter values and initial state as R-vectors, \( p \leftarrow c(r=1,K=1,a=1,c=1,delta=0.5) \), and \( s \leftarrow c(R=1,N=0.01) \), respectively. The following tutorial is an example session illustrating the usage of the first four grind.R functions, by simulating and analyzing this Lotka Volterra model (see Fig. 1 for its graphical output):

**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/practicals/grindR/:

```r
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)))
  })
}
```

```r
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 returns a list of derivatives (dR,N). The names model, s, and p are the default designations for the model, state, and parameter values in all grind.R functions. This example should be self explanatory as it just defines the Lotka Volterra model \( \frac{dR}{dt} = rR(1 - R/K) - aRN \), and \( \frac{dN}{dt} = caN - \delta N \), with its parameter values and initial state as R-vectors, \( p \leftarrow c(r=1,K=1,a=1,c=1,delta=0.5) \), and \( s \leftarrow c(R=1,N=0.01) \), respectively. The following tutorial is an example session illustrating the usage of the first four grind.R functions, by simulating and analyzing this Lotka Volterra model (see Fig. 1 for its graphical output):

**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/practicals/grindR/:

```r
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)))
  })
}
```

```r
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 returns a list of derivatives (dR,N). The names model, s, and p are the default designations for the model, state, and parameter values in all grind.R functions. This example should be self explanatory as it just defines the Lotka Volterra model \( \frac{dR}{dt} = rR(1 - R/K) - aRN \), and \( \frac{dN}{dt} = caN - \delta N \), with its parameter values and initial state as R-vectors, \( p \leftarrow c(r=1,K=1,a=1,c=1,delta=0.5) \), and \( s \leftarrow c(R=1,N=0.01) \), respectively. The following tutorial is an example session illustrating the usage of the first four grind.R functions, by simulating and analyzing this Lotka Volterra model (see Fig. 1 for its graphical output):

**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/practicals/grindR/:

```r
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)))
  })
}
```

```r
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 returns a list of derivatives (dR,N). The names model, s, and p are the default designations for the model, state, and parameter values in all grind.R functions. This example should be self explanatory as it just defines the Lotka Volterra model \( \frac{dR}{dt} = rR(1 - R/K) - aRN \), and \( \frac{dN}{dt} = caN - \delta N \), with its parameter values and initial state as R-vectors, \( p \leftarrow c(r=1,K=1,a=1,c=1,delta=0.5) \), and \( s \leftarrow c(R=1,N=0.01) \), respectively. The following tutorial is an example session illustrating the usage of the first four grind.R functions, by simulating and analyzing this Lotka Volterra model (see Fig. 1 for its graphical output):

**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/practicals/grindR/:

```r
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)))
  })
}
```

```r
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 returns a list of derivatives (dR,N). The names model, s, and p are the default designations for the model, state, and parameter values in all grind.R functions. This example should be self explanatory as it just defines the Lotka Volterra model \( \frac{dR}{dt} = rR(1 - R/K) - aRN \), and \( \frac{dN}{dt} = caN - \delta N \), with its parameter values and initial state as R-vectors, \( p \leftarrow c(r=1,K=1,a=1,c=1,delta=0.5) \), and \( s \leftarrow c(R=1,N=0.01) \), respectively. The following tutorial is an example session illustrating the usage of the first four grind.R functions, by simulating and analyzing this Lotka Volterra model (see Fig. 1 for its graphical output):
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.

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

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 graphical output of this example is displayed in Fig. 2, where bullets indicate stable steady states and circles depict unstable equilibria. We start close to the three steady states, call the Newton-Raphson algorithm, and store these states in the variables low, mid, and high, respectively. Using parameter continuation 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, ...). This R-script is available as the file operon.R:

```r
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)
low <- newton(s,plot=T)
mid <- newton(c(A=0.8,M=0.2),plot=T)
high <- newton(c(A=2,M=1),plot=T)
continue(mid,x="L",y="A",xmax=2,ymax=4)
```

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), vectors of equations (Section 3), and parameter estimation (Section 4), and can be read when needed. Section 5 is a reference manual that can be skimmed through and consulted when required. Finally, Section 6 provides simple installation instructions.

2 Combining numerical integration with events

The deSolve package allows one to execute discrete events while integrating the model numerically by using the events argument (see the ode() manual). This remains possible in grind.R because run() passes additional options to ode() via the ellipsis (…) argument in R. We have added a somewhat simpler option (after) to handle events that are executed after each time step within run(). For instance run(after="state<-ifelse(state<1e-9,0,state)"") will set small variables to zero after each time step. This new option is illustrated by the following three examples each providing an R-command as a text, after="text" in a call of run() of the Lotka Volterra model introduced above.

For example, after="parms[\"r\"]<-rnorm(1,1,0.1)" 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 the backslashes in \"\" before the quotes around the parameter name, because these quotes would otherwise mark the beginning or ending of a text. (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=1 in run()).

The second example, run(after="if(t==20)state[\"N\"]<-0"), sets the predators \( N = 0 \) when time \( t = 20 \) (see Fig. 3b). Note again that \( s \) is called state in run() (see the Manual), and the backslashes in \"\". Again, the more simple state[2]<-0 would achieve the same effect, because \( N \) is the second variable. Finally, the third example adds Gaussian noise to both variables, e.g., after="state<-state+rnorm(2,0,0.01)" (see Fig. 3c). Note that rnorm(2,0,0.01) provides two random values, that are added to the two variables, respectively. The integration starts close to the steady state to prevent problems arising from random values setting a population to a negative value.
Figure 3: The three runs with `after="text"` executed after every timestep

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

These three examples are combined in the R-script `events.R`:

```r
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)")
run(after="if(t==20)state["N"]<-0")

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

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

**A final “event” to tweak the numerical data.** To modify the numerical solution data computed within `run()` one can pass on any R-command as a text with the `tweak` option. For instance, `tweak="nsol<-cbind(nsol,nsol[,2]+nsol[,3]);names(nsol)[4]<-"T"",` adds a fourth column to the solution by summing the first and second variable, and calls this column “T” (for to-
tal). Note that the numerical solution is called nsol, and that the first column contains the time. This manipulated nsol table is subsequently passed 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.

Maps. Note that one can study maps by switching to Euler integration:

```r
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=".")
```

### 3 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 \( N_i \),

\[
\frac{dR_i}{dt} = b_i R_i \left( 1 - \sum R_i \right) - d_1 R_i - a R_i N_i \quad \text{and} \quad \frac{dN_i}{dt} = a R_i N_i - d_2 N_i .
\]

In the R-script `vector.R` we draw random prey birth rates, \( b_i \), from a normal distribution:

```r
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()
```

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

\[
\frac{dN_i}{dt} = b_i N_i \left( 1 - \sum N_i \right) - dN_i , \quad \text{for} \quad i = 1, 2, \ldots, n
\]

and let strain \( N_{i+1} \) evolve from strain \( N_i \) at a mutation rate \( \mu \). When the expected number of mutants, \( \mu N_i \), 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 R-script `evolve.R` birth rates \( b_i \) of the strains increase linearly from \( b_1 = 1 \) to \( b_n = 2 \):

```r
model <- function(t, state, parms){
  with(as.list(c(state,parms)),{
```
Figure 5: Fitting the Lotka Volterra model to data. Lines show the model for the best estimated parameters, and the symbols depict the data.

```r
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)")

4 Parameter estimation

One can fit the parameters of a model to data using the function `fit()`. 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 same Lotka Volterra model as above, after creating a data set using `run(table=T)`. 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. Calling `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 provide a list of data sets to the first `datas` option (see the `fit(list(dataR,dataN))` example). When fitting several data sets, some of the parameters could be the same across all data sets, whereas others could differ, and have a unique value in each data set.
(see Fig. 5B & C). The shared parameters remain to be provided by the free option, and the unique parameters are provided by the differ option (see the fit(list(data, data2), differ=c("R", "N", "K")) example below). Here differ is just a vector of parameter names. However, 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). When data sets differ in parameters that are known, these can be provided with the option fixed, which has to be a list (see the lines below fixed<-list() in the example). 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. The following script is available as the file lotka_fit.R:

```r
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

w <- c(names(s),names(p)) # w provides the names of free parameters
f <- fit(data,free=w) # Fit the data again
f <- fit(data,initial=T,free=names(p)) # Take initial condition from data

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)) # which gives the same result

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) # 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

par(mfrow=c(1,2)) # Show two panels next to each other
f <- fit(list(data,data2),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),differ=differ,main=c("A","B"))

# Provide fixed parameters as a list:
fixed <- list(R=c(1.05),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:
fit(list(data,data2),free=free,differ=differ,fixed=fixed,main=c("A","B"),bootstrap=100)$par
par(mfrow=c(1,1))
```
5 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:

```r
run <- function(tmax=100, tstep=1, state=s, parms=p, odes=model, ymin=0, ymax=NULL, log=",", x=1, y=2, xlab="Time", ylab="Density", tmin=0, draw=lines, times=NULL, show=NULL, arrest=NULL, after=NULL, tweak=NULL, timeplot=TRUE, traject=NULL, table=FALSE, add=FALSE, legend=TRUE, solution=FALSE, lwd=2, ...) run() calls the ode() function from the deSolve package. Additional arguments (...) are passed on to ode() and plot().
```

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:

```r
plane <- function(xmin=0, xmax=1.1, ymin=0, ymax=1.1, log=",", npixels=500, state=s, parms=p, odes=model, x=1, y=2, time=0, grid=5, eps=0, show=NULL, portrait=FALSE, vector=FALSE, add=FALSE, legend=TRUE, zero=TRUE, lwd=2, ...) Additional arguments (...) are passed on to run() (for the phase portrait) and to plot(). 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()`.
```

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:

```r
newton <- function(state=s, parms=p, odes=model, time=0, x=1, y=2, positive=FALSE, jacobian=FALSE, vector=FALSE, plot=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:

```r
continue <- function(state=s, parms=p, odes=model, x=1, step=0.01, xmin=0, xmax=1, y=2, ymin=0, ymax=1.1, log=",", time=0, positive=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.
```

The function `fit()` fits a model to data by non-linear parameter estimation. The output is an object (class of `modFit`) 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:

```r
fit <- function(datas=data, state=s, parms=p, odes=model, free=NULL, differ=NULL, fixed=NULL, tmin=0, tmax=NULL, ymin=0, ymax=NULL, log=",", xlab="Time", ylab="Density", bootstrap=0, show=NULL, fun=NULL, costfun=cost, 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:
These functions have many arguments, and fortunately most of them have good default values, and can hence typically be omitted. The arguments can be used to define various options and adjustments:

- **state=s, parms=p, odes=model** define the names of the state vector, parameter vector, and the model.
- **tmax=100, tstep=1** set the integration time and the reporting interval. **tmin** allows one to start a specific time point (which can be convenient when a run is continued). One can also provide a vector of time points where the run() should provide output with the times option (see the ode() manual).
- **x=1, y=2** define the variables on the axes of phase planes and bifurcation diagrams. One can also use the names of the variables to define the axes, e.g., x="R", y="N".
- **xmin=0, xmax=1.1, ymin=0, ymax=1.1, log=""** define the scaling of the horizontal and vertical axes of phase planes and bifurcation diagrams (log="y" makes the vertical axis logarithmic).
- **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 x 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.
- **xlab="Time", ylab="Density"** allow one to redefine the labels of the axes of a time plot.
- **show=NULL** defines the variables appearing in a time plot, fit, or phase plane. By default all are shown. By explicitly providing a list of variables, one can omit some of the variables. show is typically a list of names (show=c("P","Q")).
- **after=NULL** defines a command to be executed after each time step, e.g., after="state <- ifelse( state<1e-9, 0, state)" sets small variables to zero.
- **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 when fitting a piece-wise model for which the discontinuous time points have to be estimated (e.g., arrest=c("T1","T2").
- **tweak=NULL** allows one to modify the data delivered by run(). For instance one can add columns that can be fitted to data: (tweak<-"nsol<-cbind(nsol,nsol[,2]+nsol[,3]);names(nsol)[4]<-\"T\")", or transform the simulation data before they are fitted to data that is already transformed.
- **timeplot=TRUE, traject=FALSE, table=FALSE** determine the output of run() in the form of a time plot (default), trajectory in a phase plane, and/or a table with all data.
- **draw=lines** draws the timeplot as continuous lines. The alternative is draw=points.
- **lwd=2** sets the line width of the graphs, colMap=NULL and pchMap=NULL can be used to re-map the colors or symbols, e.g., pchMap=c(3,2,1) reverts the order of the first three R-symbols (see pch in points()). Note that grind.R defines its own color table (of dark colors that print well). If you don’t like this, uncomment the second colors <- line in the grind.R script.
- **main=NULL, sub=NULL** allow one to put a title at the top and/or a subtitle at the bottom of the graph (these are passed on to the R-function plot()). Note that these titles are set in a plain font (to set back to bold face, change the two font lines in the grind.R file).
- **add=FALSE, legend=TRUE** define whether or not the new plot should be added to the current one, and a legend should be placed.
- **solution=FALSE** tells grind.R whether or not the model provides time derivatives (default), or a full solution. This is particularly useful when fitting data to functions, and should obviously not be used in combination with phase plane analysis, nor with searching steady states (newton(), continue()). Models returning a solution obey the same format as the ODE models required by deSolve, except for the fact that they should return a value, or a vector of values (and not a list).
- **npxels=500** defines the resolution of the phase space in plane()
- **time=0** defines the time point for which nullclines are computed and steady states are computed (for non-autonomous ODEs).
- **grid=5** defines the number of grid points for which the vector field or phase portrait is drawn.
• \( \text{eps} = 0 \) is a shortcut in `plane()` to include or exclude the axis in the nullclines: \( \text{eps} \) is added to both \( \text{xmin} \) and \( \text{ymin} \).

• `portrait=FALSE, vector=FALSE` define whether or not `plane()` should include a phase portrait or vector field.

• `zero=TRUE` draws the phase plane for all variables other than \( x \) and \( y \) set to zero (also important when drawing nullclines of variables not appearing on the axes).

• `positive=FALSE`, setting `positive=TRUE` will restricts the search of `newton()` and `continue()` to positive steady states only.

• `jacobian=FALSE, vectors=FALSE, plot=FALSE` define whether or not `newton()` should print the Jacobian and eigenvectors, and indicate the steady state by a symbol in the phase plane.

• `datas=data` in `fit()` defines the name of the data frame containing the data, or defines a list of data frames.

• `free=NULL` defines the names of the parameters to be fitted. By default `free` equals `c(names(state), names(parms))`.

• `differ=NULL` defines the names of the 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).

• `fixed=NULL` defines a named list of the 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).

• `initial=FALSE` allows one to read the initial condition from the data (and not estimate it).

• `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`).

• `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.

• ... can be used to define parameters that are passed on to other functions

6 Installation and startup

The very first time one may need to install the Soetaert libraries into the R-environment, e.g., `install.packages(c("deSolve", "rootSolve", "FME")` in R or Jupyter, 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 on the webpage: [http://tbb.bio.uu.nl/rdb/practicals/grindR/](http://tbb.bio.uu.nl/rdb/practicals/grindR/).

For example, download the R-codes `grind.R` and `lotka.R` in a local directory, and open them in RStudio (if RStudio is the default R-environment one can just double click the links). 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.R` 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.R` functions.

Once you have a picture that you like, you may copy the lines creating that figure into the `lotka.R` window for later usage. (Again, use “Run” or “Control Enter” to execute lines from the `lotka.R` panel into the console). Finally, one can reduce the amount of white space in the margins, or plot panels
next to each other, by changing R’s default graphics settings \([\texttt{par}()]\); see the examples in \texttt{bbox.R}.

July 27, 2017, Rob J. de Boer

References