growthTools vignette

Colin T. Kremer

2021-10-20

1 Overview

The growthTools package has two primary functions: (1) estimating exponential growth rates of microbial populations based on time series of abundances (or proxies of abundance), and (2) estimating parametric relationships between measures of microbial performance (e.g., exponential growth, metabolic rates) and abiotic factors including temperature, nutrients, and light, as a way of characterizing species' functional traits. In particular, the tools developed here are intended to help automate these estimation processes across large data sets containing information on many species/genotypes under many environmental conditions.

The growthTools package relies heavily on data manipulation tools from the dplyr package to facilitate this automation, as well as maximum likelihood techniques to estimate relationships, using the companion mleTools and bbmle packages.

2 Calculate exponential population growth rates

Goal: estimate exponential growth rates from large numbers of population time series (multiple measures of abundance or fluorescence over time), while adjusting for the possible presence of intervals of time where populations are not growing exponentially (e.g., lags in growth, saturation in abundance, both, or other complexities).

2.1 Getting Started

Load essential packages, including `growthTools`.

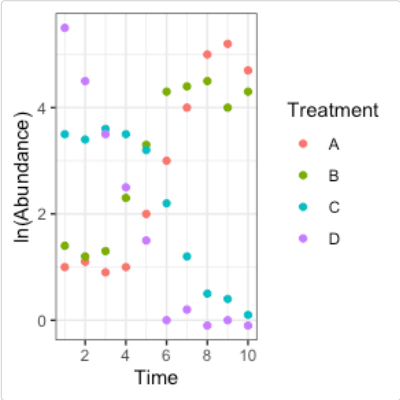
```
# To direct vignette code to use the growthTools package during development, calling
        devtools::load_all() is suggested. When ready for release, use the library command.
# http://stackoverflow.com/questions/35727645/devtools-build-vignette-cant-find-functions
# when switching, also remember to remove %\VignetteDepends{devtools} from header
# see https://community.rstudio.com/t/using-packages-that-are-not-imported-as-part-of-examples-in-
        a-vignette/2317

#devtools::load_all()
library(growthTools)

library(ggplot2)
library(dplyr)
library(tidyr)
library(reshape2)
```

```
# Construct example data set:
sdat<-data.frame(trt=c(rep('A',10),rep('B',10),rep('C',10),rep('D',10)),
                 dtime=rep(seq(1,10),4),
                 ln.fluor=c(c(1,1.1,0.9,1,2,3,4,5,5.2,4.7),
                            c(1.1,0.9,1,2,3,4,4.1,4.2,3.7,4)+0.3,
                            c(3.5,3.4,3.6,3.5,3.2,2.2,1.2,0.5,0.4,0.1),
                            c(5.5,4.5,3.5,2.5,1.5,0,0.2,-0.1,0,-0.1)))
```

2.2 Working with a single time series

We can take a quick look at the log-abundance time series created in this mock data set:

```
ggplot(sdat,aes(x=dtime,y=ln.fluor))+
  geom_point(aes(colour=trt))+theme_bw()+
  scale_x_continuous(name = 'Time',breaks = seq(0,10,2))+
  scale_y_continuous(name = 'ln(Abundance)')+
  scale_color_discrete('Treatment')
```



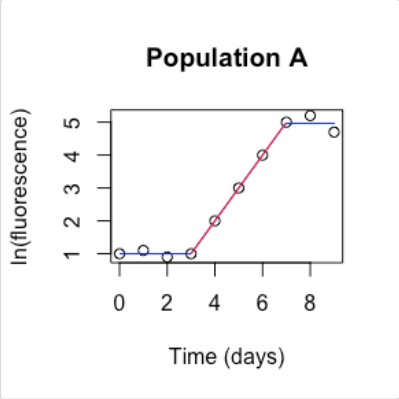
First, let's focus on applying this technique to a single time series, to get a feel for the methodology. We'll focus on log-abundance as our dependent variable, such that exponential relationships appear linear. The default approach attempts to fit a series of 5 different possible models are fit to the supplied time series: (1) a linear model, (2) a lagged growth model, (3) a saturating growth model, (4) a model of exponential decline that hits a floor, characteristic of a detection threshold, and (5) a model where growth experiences both an initial lag and achieves saturation.

Note that by default: * before fitting occurs, the x-axis (time) is shifted to start at time=0, if necessary. This can improve the stability of the regression fitting. To avoid this, you can specify `zero.time=FALSE` when calling `get.growth.rate()`) * any regression where the inferred phase of exponential growth (increasing or decreasing linear portion) contains fewer than three observations is automatically excluded from consideration, and will not appear in the output. This can be adjusted using the `min.exp.obs` option, but values less than 3 are not recommended. * the remaining models are then competed against each other based on Akaike Information Criteria adjusted for small sample sizes (AICc). Users may optionally select AIC or BIC.

The best model identified during the model comparison step is used to provide the reported estimate of exponential growth.

Here is an example from a single time series:

```
# subset the data, focusing on population A:
sdat2<-sdat[sdat$trt=='A',]

# calculate growth rate using all available methods:
res<-get.growth.rate(sdat2$dtime,sdat2$ln.fluor,plot.best.Q = T,id = 'Population A')
```



The result (shown in the above plot) indicates that both lagged and saturating portions are present in the time series, as the lagsat model outperforms all of the simpler models. If you are interested, you can extract the IC table used to make this decision (defaults to AICc comparison):

```
res$ictab
```

```
##           dAICc df
## gr.lagsat  0.0  5
## gr        16.4  3
## gr.lag    17.1  4
## gr.sat    20.8  4
```

Given the lagsat model, the inferred exponential phase - highlighted in red - has a slope that corresponds to the estimated growth rate, obtained from the best model:

```
res$best.model
```

```
## [1] "gr.lagsat"
```

```
res$best.slope
```

```
## [1] 1.000003
```

A variety of other diagnostics are available, including the R2 for the best model:

```
res$best.model.rsqr
```

```
## [1] 0.9949958
```

The standard error associated with the slope estimate:

```
res$best.se
```

```
## [1] 0.1003313
```

And the number of observations falling within the exponential phase identified in the best model, as well as the R2 for just the exponential portion of the model:

```
res$best.model.slope.n
```

```
## [1] 5
```

```
res$best.model.slope.r2
```

```
## [1] 0.9998889
```

These diverse diagnostics can be used in downstream analyses to assess the quality of the model fits, and the reliability of the exponential growth rate estimates.

It's also possible to access information from the entire suite of models that were fit, not just the one selected as the best model. For example, here's a summary of the lagged growth model (saturated, floored, and lag/sat models can be similarly accessed):

```
summary(res$models$gr.lag)
```

```
##
## Formula: y ~ lag(x, a, b, B1, s = 1e-10)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## B1   2.3282     0.7505   3.102 0.017269 *
## a    1.0000     0.3084   3.243 0.014194 *
## b    0.6964     0.1009   6.900 0.000231 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5341 on 7 degrees of freedom
##
## Number of iterations to convergence: 3
## Achieved convergence tolerance: 1.49e-08
```

We can also get summary values for all of the models, including the slopes (growth rates), and the standard error, R2, and number of observations associated with the slope estimates:

```
res$slopes
```

```
##        gr    gr.lag    gr.sat gr.lagsat
## 0.5606061 0.6964286 0.5961677 1.0000030
```

```
res$ses
```

```
##         gr     gr.lag     gr.sat  gr.lagsat
## 0.07157291 0.10093405 0.09890251 0.10033133
```

```
res$slope.rs
```

```
##        gr    gr.lag    gr.sat gr.lagsat
## 0.8846440 0.8729339 0.8880262 0.9998889
```

```
res$slope.ns
```

```
##        gr    gr.lag    gr.sat gr.lagsat
##        10         7         9         5
```

Looking at these, we can see that the growth rate estimates from the linear, lagged, and saturating models are all significantly lower than the estimate from the lagged and saturated model, which captures the correct slope.

This is also reflected in the standard error values associated with each slope estimate, obtained via:

```
res$best.se
```

```
## [1] 0.1003313
```

```
res$ses
```

```
##         gr     gr.lag     gr.sat  gr.lagsat
## 0.07157291 0.10093405 0.09890251 0.10033133
```

2.3 Applying this approach to many time series

Often we want to extract growth rates from a whole bunch of individual populations/time series. The

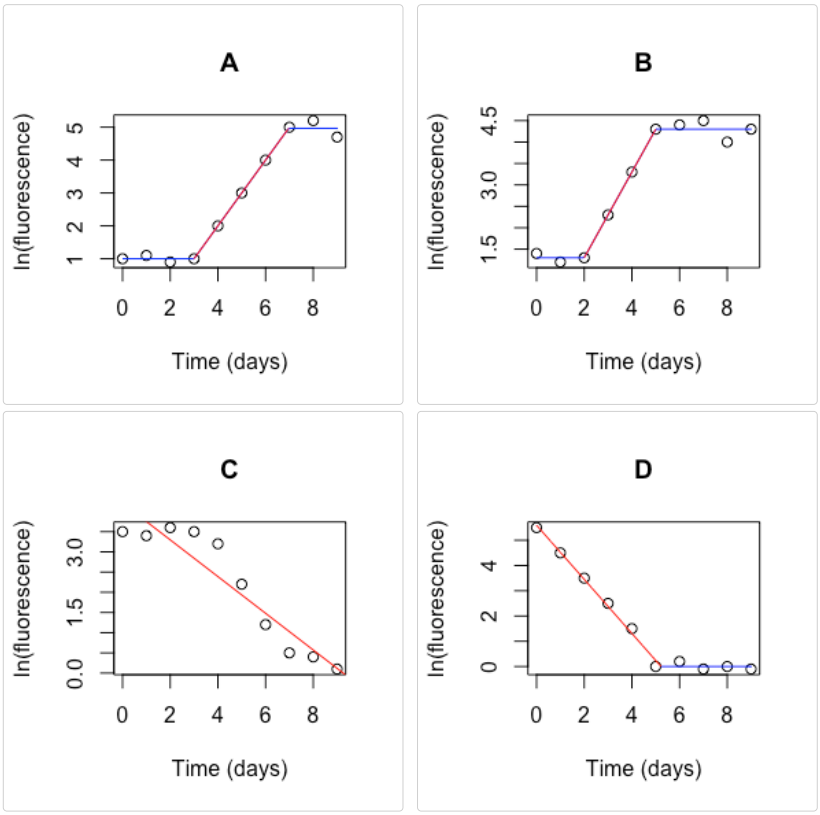
`get.growth.rate()` function is set up to make automating this process fairly easy, as we will see in this section. This approach takes advantage of `dplyr`.

First, we specify the data frame (`sdat`) containing all of our time series, then we pipe this to the `group_by()` command using the `%>%` syntax. The `group_by` command from `dplyr` is used to identify the column(s) that together identify unique individual time series. For the current example, this is just the treatment column `trt`. In a more complicated data set, this might include several columns that identify species, culture conditions, replicates, etc.

The result is a grouped data frame, which we directly pass to the `do()` command, again using `%>%`. This command applies the `get.growth.rate()` function to each individual time series identified by the grouping variables from the previous step. We pass the time column (`$dtime`) and the ln(abundance) column (`$ln.fluor`) to `get.growth.rate()`, and can also specify additional options, including the id of each population/time series, and plotting options.

In this instance, we request that the best fitting models are plotted, but leave the file path (`fpath`) option as `NA`, so the plots are displayed rather than saved to a directory. When working with a larger data set, the number of plots can be unwieldly, so it is recommended that you save rather than immediately display them.

```
gdat <- sdat %>% group_by(trt) %>%
        do(grs=get.growth.rate(x=.$dtime,y=.$ln.fluor,
                               id=.$trt,plot.best.Q=T,fpath=NA))
```



The resulting data frame is a complex structure - it contains the columns used for grouping (in this case, just `trt`), as well as a new column called `grs`. The entries in `grs` consist of the result of applying `get.growth.rate()` to each time series, including the identity of the best fit model, slope estimates, model contents, etc.

We can process this complex data structure further to obtain the different output that we're interested in. Say for example that we just want the best slope (growth rate) estimates, the identity of the best model, and the standard error associated with the growth rate estimate:

```
gdat %>% summarise(trt,mu=grs$best.slope,best.model=grs$best.model,
                   best.se=grs$best.se)
```

```
## # A tibble: 4 × 4
##   trt       mu best.model best.se
##   <chr>  <dbl> <chr>        <dbl>
## 1 A      1.00  gr.lagsat   0.100
## 2 B      1.00  gr.lagsat   0.196
## 3 C     -0.455 gr          0.0585
## 4 D     -1.07  gr.flr      0.0382
```

The result lists each population/time series, the best growth rate estimate (`mu`), and the identity of the model that produced this estimate.

However, you can select any diagnostics you want by including them in the `summarise()` command. For example, we could also request the R2 of the best model, and the number of observations falling within the exponential phase:

```
gdat %>% summarise(trt,mu=grs$best.slope,best.model=grs$best.model,
                   best.se=grs$best.se,best.R2=grs$best.model.rsqr,
                   nobs.exp=grs$best.model.slope.n)
```

```
## # A tibble: 4 × 6
##   trt       mu best.model best.se best.R2 nobs.exp
##   <chr>  <dbl> <chr>        <dbl>   <dbl>    <int>
## 1 A      1.00  gr.lagsat   0.100    0.995        5
## 2 B      1.00  gr.lagsat   0.196    0.991        4
## 3 C     -0.455 gr          0.0585   0.883       10
## 4 D     -1.07  gr.flr      0.0382   0.996        6
```

2.4 Method specification

In some situations it may be desirable to only invoke particular models/methods when estimating growth rate. For example, appropriately detecting lags or saturation requires longer time series, otherwise there's a greater chance of overfitting the data - so two or three observations won't be sufficient. Indeed, situations with only 2 timepoints will throw an error message, although the algorithm still proceeds with the linear model.

In these cases, we might prefer to tell the algorithm to just calculate growth rate using a simple linear regression, and not bother with the more complicated approaches. Here's an example (note that I've also turned off the plotting feature):

```
# Only use the linear method:
gdat <- sdat %>% group_by(trt) %>%
        do(grs=get.growth.rate(x=.$dtime,y=.$ln.fluor,id=.$trt,plot.best.Q=F,
                               methods=c('linear'))) %>%
        summarise(trt,mu=grs$best.slope,best.model=grs$best.model)
gdat
```

```
## # A tibble: 4 × 3
##   trt       mu best.model
##   <chr>  <dbl> <chr>
## 1 A      0.561 gr
## 2 B      0.418 gr
## 3 C     -0.455 gr
## 4 D     -0.656 gr
```

Or maybe we want to use the lagged, saturated, or floored models, but not linear or lagsat:

```
# Only use the linear method:
gdat <- sdat %>% group_by(trt) %>%
        do(grs=get.growth.rate(x=.$dtime,y=.$ln.fluor,id=.$trt,plot.best.Q=F,
                               methods=c('lag','sat','flr'))) %>%
        summarise(trt,mu=grs$best.slope,best.model=grs$best.model)
gdat
```

```
## # A tibble: 4 × 3
##   trt       mu best.model
##   <chr>  <dbl> <chr>
## 1 A      0.696 gr.lag
## 2 B      0.623 gr.sat
## 3 C     -0.455 gr.flr
## 4 D     -1.07  gr.flr
```

2.5 Model selection

We can also indicate which of three information criteria to use when selecting the 'best model' - AIC, AICc, or BIC. The default if no option is specified is to use AICc, which adjusts for the effects of small sample sizes, but converges on AIC in the limit of large sample sizes. Generally, the basic linear model is favored more often using

AICc than AIC, such that small lag or saturated phases may be ignored. Here's an example invoking AIC instead of AICc:

```
gdat <- sdat %>% group_by(trt) %>%
        do(grs=get.growth.rate(x=.$dtime,y=.$ln.fluor,id=.$trt,plot.best.Q=F,
                               model.selection=c('AIC'))) %>%
        summarise(trt,mu=grs$best.slope,best.model=grs$best.model)
gdat
```

```
## # A tibble: 4 × 3
##   trt       mu best.model
##   <chr>  <dbl> <chr>
## 1 A      1.00  gr.lagsat
## 2 B      1.00  gr.lagsat
## 3 C     -0.455 gr
## 4 D     -1.07  gr.flr
```

Note that if one model strongly outperforms the rest, the choice of IC to use likely won't change the outcome.

3 Fit Thermal Performance Curve to growth rate data

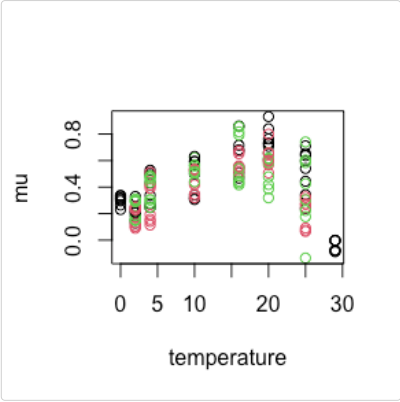
First, we'll load an example data set. It contains information on the exponential growth rate (mu) of one focal strain of a diatom (isolate.id) collected from Narragansett Bay, RI in 2017. This isolate was grown at a range of different temperatures, over three separate dilution periods, and replicated three times at each level. The final column, 'best.model' reflects the model used to provide the growth rate estimates, using functions from the `growthTools` package as described in the preceding section.

Load and examine data set:

```
data("example_TPC_data")
head(example_TPC_data)
```

```
##       experiment.ID   isolate.id temperature dilution replicate        mu
## 1  201802_RI03_CH_0 CH30_4_RI_03           0        1        R1 0.2957019
## 2  201802_RI03_CH_0 CH30_4_RI_03           0        1        R2 0.3171575
## 3  201802_RI03_CH_0 CH30_4_RI_03           0        1        R3 0.2308663
## 4  201802_RI03_CH_0 CH30_4_RI_03           0        1        R4 0.2649214
## 5 201802_RI03_CH_10 CH30_4_RI_03          10        1        R1 0.3404418
## 6 201802_RI03_CH_10 CH30_4_RI_03          10        1        R2 0.3192375
##   best.model
## 1     gr.sat
## 2     gr.sat
## 3         gr
## 4     gr.sat
## 5     gr.lag
## 6         gr
```

```
plot(mu~temperature,col=dilution,example_TPC_data)
```



Many different parametric equations have been proposed to capture the shape of this relationship, which is generally left-skewed and unimodal. The growthTools package currently contains two specific equations: * a

modified version of the equation prposed by Norberg (2004). See `?nbcurve()` for the modified version. * the double-exponential model from Thomas et al. 2017, which is similar to a model proposed by Hinshelwood many years ago. See `?decurve()` for details.

Next we'll talk about how these equations are actually fit to the data, starting with the Norberg model.

3.1 Norberg model:

To fit the Norberg model to these data, we'll use the workhorse function `get.nbcurve.tpc()`. The name of the function indicates which model we're fitting (nbcurve), and that what we're fitting is a thermal performance curve (tpc)

3.1.1 Single curves:

This is easily done for a single thermal performance curve, in this example based on growth rates estimated previously in this vignette (over one dilution period above):

```
# Filter out a single data set:
sp1 <- example_TPC_data %>% filter(isolate.id=='CH30_4_RI_03' & dilution==1)

# obtain Norberg curve parameters, using a grid search algorithm to consider a range of possible
         initial
# parameter values
nbcurve.fit<-get.nbcurve.tpc(sp1$temperature,sp1$mu,method='grid.mle2')
```

The resulting object, of class 'tpc', contains a lot of information. A basic summary is available via the default print method:

```
class(nbcurve.fit)
```

```
## [1] "tpc"
```

```
# view summary (this calls print(nbcurve.fit) behind the scenes)
nbcurve.fit
```

```
## Thermal Performance Curve fit
## Model type:  nbcurve
##
## Estimated parameters:
##      parameter 2.5 %      97.5 %
## topt 20.37253  19.51247   21.2326
## w    103.776   103.7753   103.7767
## a    -1.494426 -1.71497   -1.273881
## b    0.1108879 0.09843273 0.1233431
## s    -2.36696  -2.611987  -2.121933
##
## Fit diagnostics:
## R2 =  0.8406, logLik =  30.33986 (df =  5 ), nobs =  32
```

More detailed information on this fit can be directly accessed, including:

TPC model type (in case you forget…), parameter estimates specific to this model type (Norberg), and corresponding Fisher information confidence intervals

```
nbcurve.fit$type
```

```
## [1] "nbcurve"
```

```
# Model parameters
nbcurve.fit$cf
```

```
## $topt
## [1] 20.37253
##
## $w
## [1] 103.776
##
```

```
## $a
## [1] -1.494426
##
## $b
## [1] 0.1108879
##
## $s
## [1] -2.36696
```

```
# Fisher information confidence intervals for these parameters:
nbcurve.fit$cf_ciFI
```

```
##             2.5 %      97.5 %
## topt  19.51247148  21.2325974
## w    103.77533420 103.7766640
## a     -1.71497025  -1.2738809
## b      0.09843273   0.1233431
## s     -2.61198734  -2.1219334
```

Estimates of cardinal temperature traits (Topt, Tmin, Tmax, maximum growth rate), which are generally not model-specific, and associated confidence intervals

```
# Cardinal thermal traits:
nbcurve.fit[c('topt','tmin','tmax','umax')]
```

```
## $topt
## [1] 20.37253
##
## $tmin
## [1] -75.16319
##
## $tmax
## [1] 28.61281
##
## $umax
## [1] 0.6281357
```

```
# Confidence intervals for (some) of these traits (more to come):
nbcurve.fit$topt_ci
```

```
##    2.5 %   97.5 %
## 19.51247 21.23260
```

Diagnostics describing the quality of the model fit

```
# Diagnostics
nbcurve.fit[c('rsqr','logLik','aic','nobs','ntemps')]
```

```
## $rsqr
## [1] 0.8405531
##
## $logLik
## 'log Lik.' 30.33986 (df=5)
##
## $aic
## [1] -50.67972
##
## $nobs
## [1] 32
##
## $ntemps
## [1] 8
```

```
# See also the class constructor function for full details on output fields:
# ?new_tpc()
```

Cardinal thermal traits will be the same across all TPC models we consider, where they exist and can be estimated. However, the number and identity of model parameters varies with the TPC equation under consideration. That is why model parameters are contained in the vector `$cf` within the `tpc` object. For example, the double exponential TPC model uses 5 parameters instead of 4.

Note that the output also provides an estimate of umax, the maximum growth rate (achieved at a temperature of Topt). Using the same method employed to generate confidence bands around TPC regressions, we can also obtain an estimated 95% confidence interval around umax. This can be accessed as well:

```
nbcurve.fit$umax_ci
```

```
##     2.5 %    97.5 %
## 0.5693230 0.6869483
```

Given a set of parameter estimates in a `tpc` object, we can make predictions for growth rate at one or more specific temperatures, using the `predict()` function, as follows:

```
# by default, the results are predicted growth rates for the observed temperatures underlying the
        original fit
head(predict(nbcurve.fit))
```

```
##   temperature        mu
## 1           0 0.1792301
## 2           0 0.1792301
## 3           0 0.1792301
## 4           0 0.1792301
## 5          10 0.4003956
## 6          10 0.4003956
```

```
# but one or more user-supplied temperatures can be used instead, e.g.:
predict(nbcurve.fit,newdata=data.frame(temperature=c(12)))
```

```
##   temperature        mu
## 1          12 0.4565808
```

```
predict(nbcurve.fit,newdata=data.frame(temperature=c(12,14)))
```

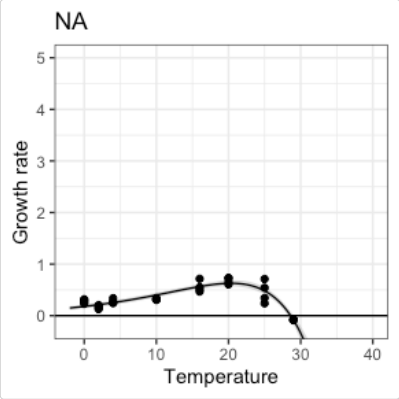
```
##   temperature        mu
## 1          12 0.4565808
## 2          14 0.5128339
```

```
# standard errors around these values can also be requested:
predict(nbcurve.fit,newdata=data.frame(temperature=c(12,14)),se.fit=TRUE)
```

```
##   temperature        mu     se.fit
## 1          12 0.4565808 0.02340301
## 2          14 0.5128339 0.02323758
```

The `predict()` function is useful for another primary objective, which is to visualize a fitted thermal performance curve. This is handled by a plot method available for 'tpc' objects, which contains both the fitted equation, and optionally also the underlying observations and an approximate confidence band around this curve. This can be a good way to visually assess the quality of the fit. The default plot looks like this:

```
plot(nbcurve.fit)
```



These plots can be customized in various ways (see later section on Visualizing Thermal Performance Curves). Alternatively, you can use the predict function to generate the values shown in the plotted curve and then design your own plots.

3.1.2 Multiple curves

We can also automatically apply this approach to multiple dilution periods or otherwise independent thermal performance curves (e.g., from different species or populations):

```
# First, let's look at an example with multiple dilutions but the same strain:
sp1b <- example_TPC_data %>% filter(isolate.id=='CH30_4_RI_03')

# apply get.nbcurve to the entire data set, grouping by isolate and dilution
nb.res <- sp1b %>% group_by(isolate.id,dilution) %>%
        do(tpcs=get.nbcurve.tpc(.$temperature,.$mu,method='grid.mle2'))
```

The result, `res` is a `data.frame` that contains all of our grouping variables, as well as a column `tpcs`. Each entry of this column is a single object of class `tpc`. So, we can access the same information as above for a single curve:

```
head(nb.res)
```

```
## # A tibble: 3 × 3
## # Rowwise:
##   isolate.id   dilution tpcs
##   <fct>           <int> <list>
## 1 CH30_4_RI_03        1 <tpc>
## 2 CH30_4_RI_03        2 <tpc>
## 3 CH30_4_RI_03        3 <tpc>
```

```
# e.g., for the first dilution period
nb.res$tpcs[[1]]
```

```
## Thermal Performance Curve fit
## Model type:  nbcurve
##
## Estimated parameters:
##      parameter 2.5 %      97.5 %
## topt 20.37253  19.51247   21.2326
## w    103.776   103.7753   103.7767
## a    -1.494426 -1.71497   -1.273881
## b    0.1108879 0.09843273 0.1233431
## s    -2.36696  -2.611987  -2.121933
##
## Fit diagnostics:
## R2 =  0.8406, logLik =  30.33986 (df =  5 ), nobs =  32
```

```
nb.res$tpcs[[1]]$topt
```

```
## [1] 20.37253
```

Or we can extract a table of values across multiple curves, using `dplyr` techniques. Note here that if you want to

access model-specific parameters, such as the Norberg model's `a` parameter, you have to reach an extra level deep into the `tpcs`, accessing first the list of coefficients, `cf`, and then the specific parameter of interest. Hence the `tpcs$cf$a`.

```
# process results
nb.res2<-nb.res %>% summarise(isolate.id,dilution,topt=tpcs$topt,tmin=tpcs$tmin,
                              tmax=tpcs$tmax,umax=tpcs$umax,rsqr=tpcs$rsqr,
                              a=tpcs$cf$a,b=tpcs$cf$b,w=tpcs$cf$w)
nb.res2
```

```
## # A tibble: 3 × 10
##   isolate.id   dilution  topt   tmin  tmax  umax  rsqr     a      b     w
##   <fct>           <int> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>
## 1 CH30_4_RI_03        1  20.4 -75.2   28.6 0.628 0.841 -1.49 0.111  104.
## 2 CH30_4_RI_03        2  18.6  -2.19  27.0 0.566 0.964 -1.71 0.0719  29.2
## 3 CH30_4_RI_03        3  15.5  -1.80  26.4 0.591 0.712 -1.01 0.0344  28.2
```

Again, a variety of other diagnostics are available, including the log likelihood of the model fit, AIC, the total number of observations the regression is based on, how many unique temperature treatments are involved, and approximate confidence intervals for the coefficients of the model (based on Fisher information). These can be accessed by including different terms in the `summarise` command, for example:

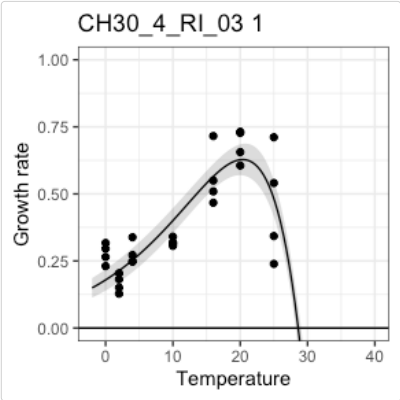
```
nb.res %>% summarise(isolate.id,dilution,ntemps=tpcs$ntemps,
                     topt=tpcs$topt,topt.lwr=tpcs$topt_ci[1],topt.upr=tpcs$topt_ci[2])
```

```
## # A tibble: 3 × 6
##   isolate.id   dilution ntemps  topt topt.lwr topt.upr
##   <fct>           <int>  <int> <dbl>    <dbl>    <dbl>
## 1 CH30_4_RI_03        1      8  20.4     19.5     21.2
## 2 CH30_4_RI_03        2      6  18.6     17.9     19.3
## 3 CH30_4_RI_03        3      6  15.5     12.4     18.6
```

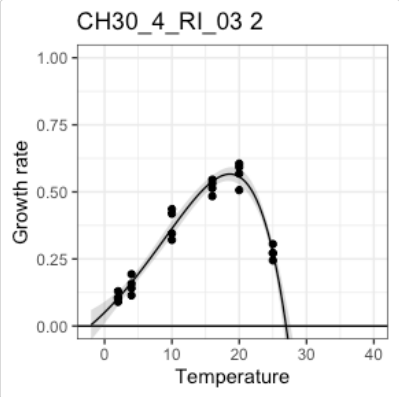
We can also generate plots of all of the different fits, using the information in `nb.res`. Basically, for each entry in `nb.res`, generate a plot of the `tpc` and label it with the corresponding isolate and dilution. Each entry in `nb.res.graphs` is then a `ggplot` graph that can be displayed.

```
nb.res.graphs<-nb.res %>% group_by(isolate.id,dilution) %>%
                          do(graphs=plot(.$tpcs[[1]],main=paste(.$isolate.id,.$dilution),
                                         ylim=c(0,1)))
nb.res.graphs$graphs[1:3]
```

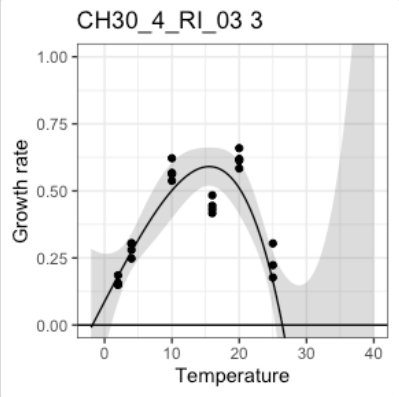
```
## [[1]]
```




```
##
## [[2]]
```




```
##
## [[3]]
```



3.2 Double exponential model:

Many different equations capturing the shape of thermal performance curves have been proposed (see for example Grimaud et al. 2017). The above examples use an equation proposed by Norberg et al. 2004. The current version of the `growthTools` package also allows users to fit a different parametric equation, known as the double exponential model (see Thomas et al. 2017, Global Change Biology, and Kremer et al. in prep). In brief, it models net population growth rate as the difference between two processes that depend exponentially on temperature: birth and death.

Right now, model fitting using this approach is pretty slow; this model is a 5 parameter model (whereas Norberg uses 4 parameters), and several of the parameter estimates tend to covary strongly, so convergence takes many iterations of the optimization algorithm. There are ways to speed things up (by providing a smaller grid of initial parameter guesses to grid.mle2 behind the scenes), but this comes at an elevated risk of finding a local rather than globally optimal set of parameter estimates.

3.2.1 Fit Multiple curves

You can access this functionality essentially the same way as fitting the Norberg curve (`get.nbcurve.tpc()`), except using `get.decurve.tpc()` inside of the `dplyr` command.

```
# fit decurve to multiple TPCs
de.res <- sp1b %>% group_by(isolate.id,dilution) %>%
          do(tpcs=get.decurve.tpc(.$temperature,.$mu,method='grid.mle2'))
```

Again, note that although the class of the results of `get.decurve.tpc` and `get.nbcurve.tpc` is the same (`tpc`), some of the contents of the resulting object differ. For example:

```
nb.res$tpcs[[1]]$type
```

```
## [1] "nbcurve"
```

```
nb.res$tpcs[[1]]$cf
```

```
## $topt
## [1] 20.37253
##
## $w
## [1] 103.776
##
## $a
## [1] -1.494426
##
## $b
## [1] 0.1108879
##
## $s
## [1] -2.36696
```

```
de.res$tpcs[[1]]$type
```

```
## [1] "decurve"
```

```
de.res$tpcs[[1]]$cf
```

```
## $topt
## [1] 20.71794
##
## $b1
## [1] 0.2036075
##
## $b2
## [1] 0.08491235
##
## $d0
## [1] 2.731322e-07
##
## $d2
## [1] 0.1800175
##
## $s
## [1] 0.09322711
```

As before, we can summarise the results of the multiple double exponential curves that were fit above:

```
# summarise the numerical results
de.res2 <- de.res %>% summarise(isolate.id,dilution,topt=tpcs$topt,
                                tmin=tpcs$tmin,tmax=tpcs$tmax,rsqr=tpcs$rsqr,
                                b1=tpcs$cf$b1,b2=tpcs$cf$b2,d0=tpcs$cf$d0,d2=tpcs$cf$d2)
de.res2
```

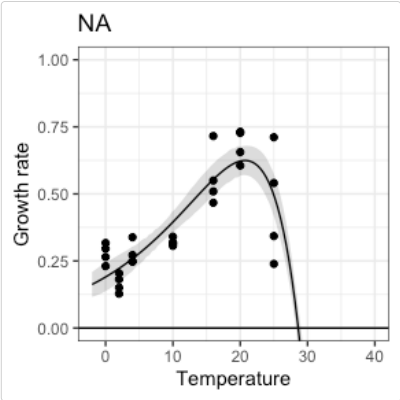
```
## # A tibble: 3 × 10
##   isolate.id   dilution  topt    tmin  tmax  rsqr    b1     b2         d0     d2
##   <fct>           <int> <dbl>   <dbl> <dbl> <dbl> <dbl>  <dbl>      <dbl>  <dbl>
## 1 CH30_4_RI_03        1  20.7 -159.    28.6 0.842 0.204 0.0849    2.73e-7 0.180
## 2 CH30_4_RI_03        2  19.0   -1.17  26.6 0.965 2.01  0.0160    1.97e+0 0.235
## 3 CH30_4_RI_03        3  15.6   -1.85  26.4 0.713 7.08  0.0480    1.93e+0 0.0577
```

And again, you can predict values on the fitted curves or make plots, just like with the Norberg model.

```
head(predict(de.res$tpcs[[1]]))
```

```
##   temperature        mu
## 1           0 0.1902190
## 2           0 0.1902190
## 3           0 0.1902190
## 4           0 0.1902190
## 5          10 0.3949437
## 6          10 0.3949437
```

```
plot(de.res$tpcs[[1]],ylim=c(0,1))
```

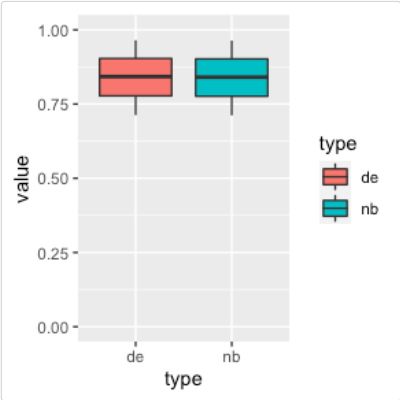


Finally, it should also be straightforward to fit both types of model to your data, then try to decide which model performs best, using criteria such as R2 or AIC. For now though, here's a basic visual comparison:

```
nb.res3<-melt(nb.res2,id.vars=c('isolate.id','dilution'))
de.res3<-melt(de.res2,id.vars=c('isolate.id','dilution'))
res2<-rbind(data.frame(type='nb',nb.res3),data.frame(type='de',de.res3))
res2[res2$variable=='rsqr',]
```

```
##    type   isolate.id dilution variable     value
## 13   nb CH30_4_RI_03        1     rsqr 0.8405531
## 14   nb CH30_4_RI_03        2     rsqr 0.9635391
## 15   nb CH30_4_RI_03        3     rsqr 0.7117982
## 34   de CH30_4_RI_03        1     rsqr 0.8423461
## 35   de CH30_4_RI_03        2     rsqr 0.9648146
## 36   de CH30_4_RI_03        3     rsqr 0.7127087
```

```
ggplot(res2[res2$variable=='rsqr',],aes(x=type,y=value))+
  geom_boxplot(aes(fill=type))+
  scale_y_continuous(limits=c(0,1))
```



3.3 Allow for multiple grouping variables:

We could also apply this general approach to a larger data set, with more species and additional grouping variables (caution - this can take a few minutes to run):

```
table(example_TPC_data[,c('isolate.id','dilution')])
```

```
##               dilution
## isolate.id      1  2  3
##   CH30_4_RI_03 32 24 24
##   TH2_15_RI_03 32 24 24
```

```
# create informative ID column for each combo of unique strain and dilution period:
example_TPC_data$id<-paste(example_TPC_data$isolate.id,example_TPC_data$dilution)

res2 <- example_TPC_data %>% group_by(isolate.id,dilution) %>%
        do(tpcs=get.nbcurve.tpc(.$temperature,.$mu,method='grid.mle2'))

clean.res <- res2 %>% summarise(isolate.id,dilution,topt=tpcs$topt,
                                tmin=tpcs$tmin,tmax=tpcs$tmax,rsqr=tpcs$rsqr,
                                a=tpcs$cf$a,b=tpcs$cf$b,w=tpcs$cf$w)
data.frame(clean.res)
```

```
##     isolate.id dilution     topt       tmin     tmax      rsqr          a
## 1 CH30_4_RI_03        1 20.37253 -75.163191 28.61281 0.8405531 -1.4944256
## 2 CH30_4_RI_03        2 18.62345  -2.193116 26.96102 0.9635391 -1.7050288
## 3 CH30_4_RI_03        3 15.54443  -1.802635 26.40737 0.7117982 -1.0070292
## 4 TH2_15_RI_03        1 19.13414 -24.304844 28.98711 0.9173149 -1.2113888
## 5 TH2_15_RI_03        2 17.36326 -72.395537 25.73787 0.8422184 -1.1095140
## 6 TH2_15_RI_03        3 15.24952  -2.818088 75.95296 0.3671215  0.4247727
##             b         w
## 1  0.11088791 103.77600
## 2  0.07190043  29.15413
## 3  0.03440944  28.21000
## 4  0.07847143  53.29196
## 5  0.10826745  98.13341
## 6 -0.03887415  78.77105
```

4 Visualize Thermal Performance Curves

In the earlier section on fitting the Norberg thermal performance curve, there was a brief example of generating a plot of the resulting fit. Now we will examine the plotting function in greater detail. There are two critical functions within the package that allow us to predict and plot values from a given TPC fit.

4.1 Predict values from a TPC

The first of these is the `predict()` function. Basically, we can provide the function with the output from a single `get.nbcurve.tpc` fit and it will return a set of temperatures and predicted growth rates. For example:

```
# pull out the fit resulting from a single get.nbcurve.tpc:
fit<-nb.res$tpcs[[1]]

# generate predictions for growth rate at the experimental temperatures used in the original fit:
tmp<-predict(fit)
head(tmp)
```

```
##   temperature        mu
## 1           0 0.1792301
## 2           0 0.1792301
## 3           0 0.1792301
## 4           0 0.1792301
## 5          10 0.4003956
## 6          10 0.4003956
```

Alternatively, users can specify a custom set of temperature values - here at 20, 25, and 30 C.

```
predict(fit,newdata=data.frame(temperature=c(20,25,30)))
```

```
##   temperature         mu
## 1          20  0.6275080
## 2          25  0.4823347
## 3          30 -0.3385213
```

The function can also generate standard errors around the predicted growth rate values, using the delta method (see Bolker book, pg. 255).

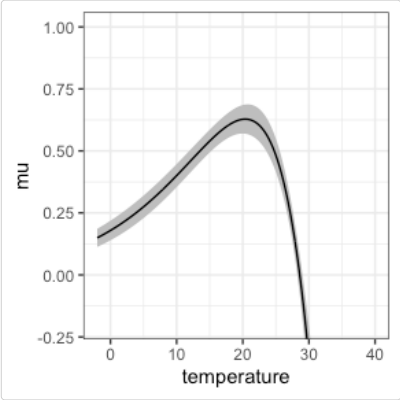
```
predict(fit,newdata=data.frame(temperature=c(20,25,30)),se.fit=TRUE)
```

```
##   temperature         mu     se.fit
## 1          20  0.6275080 0.02930842
## 2          25  0.4823347 0.03432288
## 3          30 -0.3385213 0.06753617
```

4.2 Plot values from a TPC

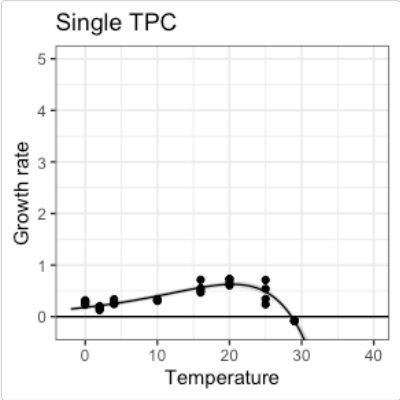
These predicted values can be used as the foundation for plotting fitted Norberg curves (or curves based on other TPC models, like the double exponential):

```
# generate predictions for a single curve, along with standard errors:
tmp<-predict(fit,newdata=data.frame(temperature=seq(-2,40,0.1)),se.fit=T)

# plot the predictions
ggplot(tmp,aes(x=temperature,y=mu))+
  geom_ribbon(aes(ymin=mu-1.96*se.fit,ymax=mu+1.96*se.fit),fill='gray')+
  geom_line()+
  coord_cartesian(ylim=c(-0.2,1))+
  theme_bw()
```



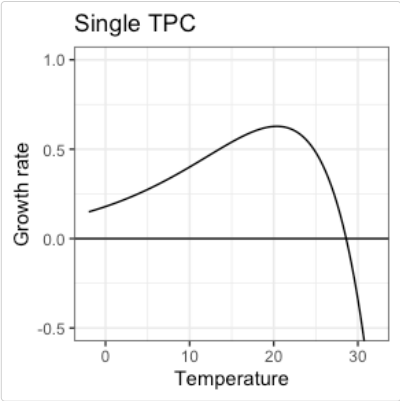
For ease, this capacity has been formalized using the `plot` method for `tpc` objects, which invokes `predict` behind the scenes, for example:

```
# use function to generate a single curve plot:
plot(fit,main='Single TPC')
```



We can also control whether the confidence band and raw data are displayed, or alter the range of the plot:

```
# use function to generate a single curve plot:
c1<-plot(fit,plot_ci = FALSE,plot_obs = FALSE,xlim=c(-2,32),ylim=c(-0.5,1),main='Single TPC')
c1
```

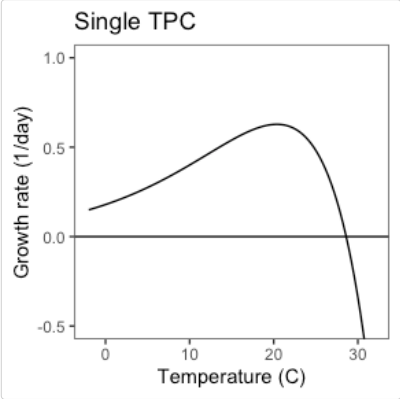


Finally, users can also save the ggplot2 object created and then subsequently modify the layout of the plot. For example, modify the axes labels to add units and remove the grid lines:

```
# demonstrate changes in layout
c1+scale_x_continuous('Temperature (C)')+
  scale_y_continuous('Growth rate (1/day)')+
  theme(panel.grid = element_blank())
```

```
## Scale for 'x' is already present. Adding another scale for 'x', which will
## replace the existing scale.
```

```
## Scale for 'y' is already present. Adding another scale for 'y', which will
## replace the existing scale.
```



4.3 Plot multiple TPCs at once:

The `plot()` function is suited to displaying the results of a single TPC, but cannot easily be used to combine multiple TPCs within a single plot. To accomplish this, it's better to take advantage of the `predict()` function, combined with the same `dplyr` capabilities that allow us to fit multiple TPCs in the first place.

Recall that when we used `get.nbcurve.tpc` to fit multiple curves, we defined a set of grouping variables that uniquely identified the growth rate data for each individual TPC. The resulting output is a complex data frame that retains columns for each of these grouping variables, as well as a new column (which we called `tpcs`), which holds the individual Norberg curve fits as objects of class `tpc`.

We can take this output and process it using `predict()` as follows:

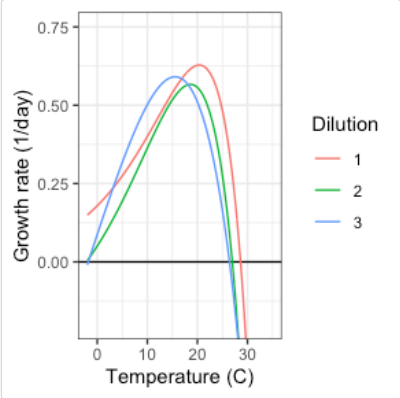
```
# generate predictions for multiple curves:
tmp2 <- nb.res %>% group_by(isolate.id,dilution) %>%
         do(predict(.$tpcs[[1]],newdata=data.frame(temperature=seq(-2,40,0.1)),se.fit=TRUE))
head(tmp2)
```

```
## # A tibble: 6 × 5
## # Groups:   isolate.id, dilution [1]
##   isolate.id   dilution temperature    mu se.fit
##   <fct>           <int>       <dbl> <dbl>  <dbl>
## 1 CH30_4_RI_03        1        -2   0.150 0.0186
## 2 CH30_4_RI_03        1        -1.9 0.151 0.0187
```

```
## 3 CH30_4_RI_03        1        -1.8 0.152 0.0188
## 4 CH30_4_RI_03        1        -1.7 0.154 0.0188
## 5 CH30_4_RI_03        1        -1.6 0.155 0.0189
## 6 CH30_4_RI_03        1        -1.5 0.157 0.0190
```

Subsequently, we can take this resulting data.frame and plot it with ggplot2, using the grouping variables to visually distinguish different curves:

```
# plot multiple curves simultaneously
multi.curve<-ggplot(tmp2,aes(x=temperature,y=mu))+
  geom_hline(yintercept = 0)+
  geom_line(aes(colour=factor(dilution)))+
  coord_cartesian(ylim=c(-0.2,0.75),xlim=c(-2,35))+
  scale_x_continuous('Temperature (C)')+
  scale_y_continuous('Growth rate (1/day)')+
  scale_colour_discrete('Dilution')+
  theme_bw()
multi.curve
```

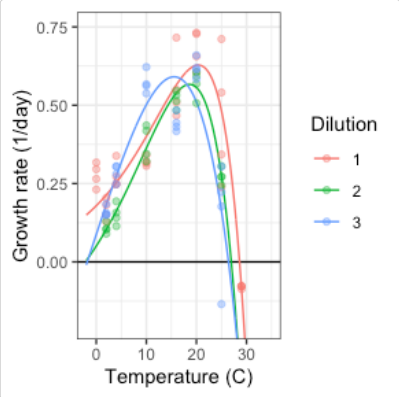


Finally, it is possible to ammend these multi-curve plots with the underlying observations used as the foundation of the regressions. This takes advantage of the fact that the original observations are contained in the `data` field reported in `tpc` objects.

```
# recapitulate corresponding data frame of observations:
tmp2.obs <- nb.res %>% group_by(isolate.id,dilution) %>% do(.$tpcs[[1]]$data)
head(tmp2.obs)
```

```
## # A tibble: 6 × 4
## # Groups:   isolate.id, dilution [1]
##   isolate.id   dilution temperature    mu
##   <fct>           <int>       <int> <dbl>
## 1 CH30_4_RI_03        1           0 0.296
## 2 CH30_4_RI_03        1           0 0.317
## 3 CH30_4_RI_03        1           0 0.231
## 4 CH30_4_RI_03        1           0 0.265
## 5 CH30_4_RI_03        1          10 0.340
## 6 CH30_4_RI_03        1          10 0.319
```

```
multi.curve+geom_point(data=tmp2.obs,aes(colour=factor(dilution)),alpha=0.4)
```



4.4 Saving plots

We can also save plots to a file folder, rather than displaying them directly:

```
#fpath<-'/Users/colin/Research/Software/growthTools/user/'
#plot(fit,plot_ci = T,plot_obs = T,ylim=c(-0.2,1),fpath = fpath,main='Single example TPC')
```

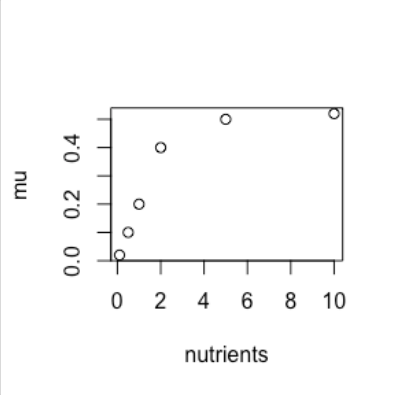
5 Fit Nutrient Performance Curves

Microbial growth rate can also vary with the availability (concentration) of essential nutrients, such as nitrogen or phosphorus. The growthTools package includes a growing set of routines for fitting canonical functions to these relationships (which we term 'nutrient performance curves' or NPCs), such as the saturating Monod curve (Monod 1942). The syntax is very similar to that used for fitting thermal performance curves.

5.1 Fit a single Monod curve:

Here's an example of what one of these data sets on growth vs. nutrient concentration might look like:

```
dat<-data.frame(nutrients=c(0.1,0.5,1,2,5,10),mu=c(0.02,0.1,0.2,0.4,0.5,0.52))
plot(mu~nutrients,data=dat)
```



We can fit a Monod function to these data using 'mle2'. Note that currently a grid.mle2 method is not implemented, as these fits are lower dimension and generally more resilient than for thermal performance curves.

```
fit.monod<-get.monod.npc(nutrients=dat$nutrients,mu=dat$mu,method='mle2',fix_intercept = T)
```

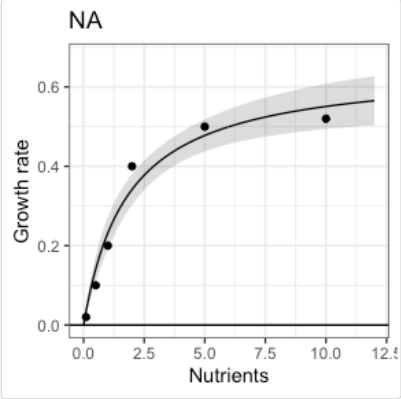
As with thermal performance curves, there are built-in methods for printing summary information from the fit, as well as displaying a plot of the fitted relationship.

```
print(fit.monod)
```

```
## Nutrient performance Curve fit
## Model type:  monod
##
```

```
## Estimated parameters:
##      parameter 2.5 %     97.5 %
## umax 0.6504832 0.551041  0.7499253
## k    1.808149  1.021674  2.594623
## s    -3.327181 -3.892985 -2.761377
##
## Fit diagnostics:
## R2 =  0.9658, logLik =  11.44946 (df =  3 ), nobs =  6
```

```
plot(fit.monod)
```



Finally, the `get.monod` function includes the option of either fixing the intercept at 0 (so no growth is allowed when a nutrient is absent), or allowing the intercept term z to be independently estimated. For example:

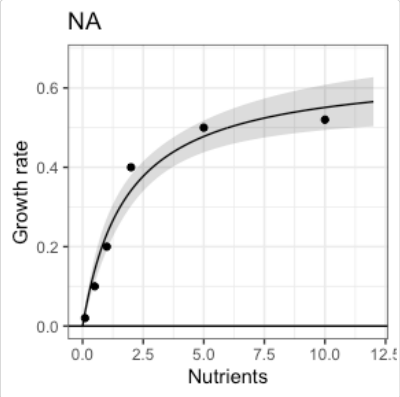
```
fit.monod.2<-get.monod.npc(nutrients=dat$nutrients,mu=dat$mu,method='mle2',fix_intercept = F)
print(fit.monod)
```

```
## Nutrient performance Curve fit
## Model type:  monod
##
## Estimated parameters:
##      parameter 2.5 %     97.5 %
## umax 0.6504832 0.551041  0.7499253
## k    1.808149  1.021674  2.594623
## s    -3.327181 -3.892985 -2.761377
##
## Fit diagnostics:
## R2 =  0.9658, logLik =  11.44946 (df =  3 ), nobs =  6
```

```
print(fit.monod.2)
```

```
## Nutrient performance Curve fit
## Model type:  monod
##
## Estimated parameters:
##      parameter 2.5 %     97.5 %
## umax 0.6504832 0.551041  0.7499253
## k    1.808149  1.021674  2.594623
## s    -3.327181 -3.892985 -2.761377
##
## Fit diagnostics:
## R2 =  0.9658, logLik =  11.44946 (df =  3 ), nobs =  6
```

```
plot(fit.monod.2)
```



5.2 Fit multiple Monod curves:

Again, as with thermal performance curves, the `get.monod.npc` function works with tools from the `tidyverse` so we can fit multiple data sets at once to characterize their nutrient performance curves (npc's). For example:

```
monod.tmp<-data.frame(sps=c(rep('A',6),rep('B',6)),nutrients=rep(c(0.1,0.5,1,2,5,10),2),mu=c(c(0.02,0.1,0.2,0.4,0.5,0.52),3*c(0.02,0

res <- monod.tmp %>% group_by(sps) %>%
  do(npcs=get.monod.npc(.$nutrients,.$mu,method='mle2'))

res2 <- res %>% summarise(sps,umax=npcs$cf$umax,k=npcs$cf$k,
                          z=npcs$cf$z,s=npcs$cf$s,rsqr=npcs$rsqr)
res2
```

```
## # A tibble: 2 × 2
##   sps    rsqr
##   <chr> <dbl>
## 1 A     0.966
## 2 B     0.966
```

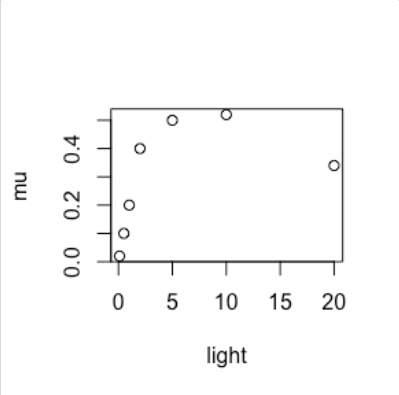
6 Fit Light Performance Curves (LPCs)

Autotrophic microbial growth rates can also vary with light level. The growthTools package includes a growing set of routines for fitting canonical functions to these relationships (which we term 'light performance curves'), such as the Eilers-Peeters model (1988). The syntax is very similar to that used for fitting thermal and nutrient performance curves.

6.1 Fit a single Eiler-Peeters curve:

Here's an example of what one of these data sets on growth vs. light might look like:

```
dat<-data.frame(light=c(0.1,0.5,1,2,5,10,20),mu=c(0.02,0.1,0.2,0.4,0.5,0.52,0.34))
plot(mu~light,data=dat)
```



We can fit an Eiler-Peeters function to these data using 'mle2'. Note that currently a grid.mle2 method is not

implemented, as these fits are lower dimension and generally more resilient than for thermal performance curves.

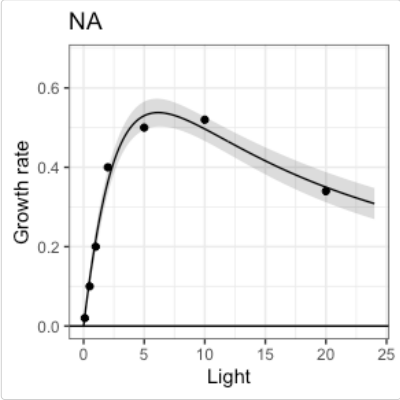
```
fit.ep<-get.ep.lpc(light=dat$light,mu=dat$mu,method='mle2',fix_intercept = T)
```

As with thermal performance curves, there are built-in methods for printing summary information from the fit, as well as displaying a plot of the fitted relationship.

```
print(fit.ep)
```

```
## Light performance Curve fit
## Model type:  ep
##
## Estimated parameters:
##      parameter 2.5 %     97.5 %
## a    0.2537729 0.2008718 0.3066739
## umax 0.5378785 0.5028663 0.5728907
## Lopt 6.151432  5.481033  6.821832
## s    -3.796706 -4.320538 -3.272874
##
## Fit diagnostics:
## R2 =  0.9846, logLik =  16.64438 (df =  4 ), nobs =  7
```

```
plot(fit.ep)
```



6.2 Fit multiple light curves:

Again, as with thermal performance curves, the `get.ep.npc` function works with tools from the `tidyverse` so we can fit multiple data sets at once to characterize their light performance curves (npc's). See examples for Monod NPC.