Analysing growth curves and other user-defined data in opm

Markus Göker
Leibniz Institute DSMZ

Abstract

This is the tutorial on the analysis of growth curves and other user-defined kinetics with the **opm** package in the version of November 27, 2015. It is explained how any kinds of growth or respiration measurements can be input into **opm**. Data without a real structuring into plates and wells can nevertheless be studied with **opm** by using a *virtual* arrangement into plates and wells. This convention is not an oddity but rather the appropriate means to enable the visual and statistical comparisons of interest and to disable those that make no sense for the data. We also show how Phenotype Microarray (PM) data with user-defined plate types can be analysed. These include plates suitable for an OmniLog® PM system but measured with another instrument. Analysing such data visually and statistically requires in some cases adaptations of function arguments whose defaults are targeting PM data. All these practically relevant issues are explained in detail.

Keywords: Growth Kinetics.

1. Introduction

A detailed description of the OmniLog® Phenotype Microarray (PM) system, its measuring procedure and data characteristics are found in the vignette "**opm**: An R Package for Analysing Phenotype Microarray and Growth Curve Data" (called "main tutorial" in the following). How substrate information stored within **opm** can be accessed and used for advanced visual and statistical analyses is explained in the vignette "Working with substrate information in **opm**" (called "substrate tutorial" in the following). The description of the methods below do not presuppose that the user is already familiar with the usage of **opm**. But for details on its approaches to visualisation and statistical analysis we will refer to the main tutorial, the substrate tutorial as well as the entries of the **opm** manual. Especially the concepts behind the different classes of **opm** objects are only explained in the main tutorial, and the methods available for these classes are only explained in the main and substrate tutorial.

In addition to visual inspection or statistical comparative analyses of PM data, as described in the main tutorial and the substrate tutorial, users might be interested in analysing data other than PM data, or analysing PM with user-defined plate types. To work with user-defined PM plates only requires registering these plates, i.e. storing a mapping from well coordinates to substrate names, and optionally also a full, descriptive name for the plate. The analysis of data other than PM data, such as growth curves, additionally requires inputting these data and converting them to OPMX objects. If these data are not really structured into

plates and wells, a *virtual* arrangement into plates and wells must be established, as well as a virtual positioning of the plates in a reader, which is used for identifying each plate. This nomenclature may be unusual for data that have not been measured in plate readers, but presents no problems in practice. Users should be aware, however, which kinds of comparisons can be made within and between plates of the same plate type. Indeed, the arrangement into virtual plates and wells is the appropriate means to ensure that the visual and statistical comparisons of interest for the data are enabled and those that make no sense are disabled. Note that some defaults of the plotting functions are only suitable for PM data. Hence, the functions should be called slightly distinctly.

Besides these slight restrictions, which are illustrated with examples below, non-PM data can be analysed with **opm** almost like PM data.

Plates suitable for an OmniLog® PM system but measured with another instrument also need special care when inputting, even though it is obvious that they are structured into plates and wells. An according example is also included in this tutorial.

2. Preparation

As usual, **opm** must be loaded before any analysis can begin:

```
R> if ("package:opm" %in% search())
     detach(package:opm, unload = TRUE)
R> library("opm")
```

3. Growth-curve data input

3.1. User-entered data frames

In the following we will use the growth-measurements data set from Vaas, Marheine, Sikorski, Göker, and Schumacher (2013) as exemplar. These data have been entered by hand and then input into R with one of the functions for reading Comma-Separated Values (CSV), yielding a data frame that comes with **opm**:

```
R> data(potato)
R> head(potato)
```

```
  Genotype  Treatment Replicate Time  FM DM
1  07-08-1 0.16M NaCl         1    2 597 44
2  07-08-1 0.16M NaCl         2    2 550 40
3  07-08-1 0.16M NaCl         3    2 633 48
4  07-08-1 0.16M NaCl         4    2 490 31
5  07-08-1 0.16M NaCl         5    2 617 47
6  07-08-1 0.16M NaCl         1    4 585 55
```

The measurements are in "long" format and must be reshaped using the eponymous function into "wide" format. The "long" format was deliberately chosen for demonstrating the use of the `reshape` function. We reshape separately for the Dry Mass (DM) and Fresh Mass (FM) measurements within the data set:

```
R> potato.fm <- reshape(potato, v.names = "FM", drop = "DM", direction = "wide",
     idvar = c("Genotype", "Treatment", "Replicate"), timevar = "Time")
R> potato.dm <- reshape(potato, v.names = "DM", drop = "FM", direction = "wide",
     idvar = c("Genotype", "Treatment", "Replicate"), timevar = "Time")
```

For `reshape`, "long" format means that each measurement is stored in a separate record with one entry per row (see above). Thus here for each data point the "Genotype", "Treatment" and "Time" entries have to be repeated, resulting in a data frame with 540 rows in 6 columns. A call to `reshape` can rearrange the data set into a form where the columns "Genotype", "Treatment" and "Replicate" are kept and the columns "Time" and either "FM" or "DM", respectively, are merged, resulting in 9 columns representing the measurement times:

```
R> head(potato.fm)
```

```
   Genotype  Treatment Replicate FM.2 FM.4 FM.6 FM.8 FM.10 FM.12 FM.14 FM.16 FM.18
1   07-08-1 0.16M NaCl         1  597  585  882  844  1291  1847  2232  2560  2808
2   07-08-1 0.16M NaCl         2  550  614  908 1103  1240  1798  2184  2832  2501
3   07-08-1 0.16M NaCl         3  633  570  855 1200  1392  1827  2360  2522  3113
4   07-08-1 0.16M NaCl         4  490  681 1087  994  1478  1921  2315  2317  2761
5   07-08-1 0.16M NaCl         5  617  707  962  849  1446  1853  2335  2564  2426
46  07-08-1 0.32M NaCl         1  395  551  392  342   322   322   368   336   274
```

Thus the dimensions of the data dwindled to 60 rows in 12 columns. Now the data are in the right arrangement for the next step, the conversion into OPMX or MOPMX objects. When entering data manually, users who directly choose a format analogous to the "wide" format can, of course, skip the conversion with `reshape`. Thus directly using the "horizontal" input format of `opmx` is recommended for manually entering data.

The main function for converting user-defined data frames to OPMX or MOPMX objects is `opmx`, which can directly be applied to the objects created in the last step. This works because the "horizontal" input format of `opmx` corresponds to the "wide" format of `reshape`.

```
R> potato.fm <- opmx(potato.fm, position = c("Genotype", "Replicate"),
     well = "Treatment", prefix = "FM.",
     full.name = c(fm = "Growth experiment, fresh mass"))
R> potato.dm <- opmx(potato.dm, position = c("Genotype", "Replicate"),
     well = "Treatment", prefix = "DM.",
     full.name = c(dm = "Growth experiment, dry mass"))
```

The data frames passed to `opmx` contain all substrate information in their "Treatment" column. Its content will be interpreted as substrate names for wells, which are virtual in our case. Hence, **opm** registers the mapping from well coordinates to substrate names on the fly. The substrate names are taken directly from the data frame in "horizontal" format and registered after sorting. The plate type must be provided, however. As it is not within the data frame, the short name of the plate type is taken from the `full.name` argument, whose main purpose is to enter the full, descriptive name of the plate type. That is, a virtual plate with virtual wells, yielding a user-defined plate type, will be registered. The `prefix` argument helps identifying the columns with measurements over time.

"Genotype" and "Replicate" go to the metadata of the resulting object and together identify each plate. In the case of PM data, this is done using the position of the plate within the

OmniLog® reader. Thus the relevant argument here is `position`, which must be supplied unless there is a column of that name. If so, its content is used literally, otherwise it is newly constructed from the columns explicitly given in the `position` argument, yielding a grouping of plates equivalent to the combination of factor levels in these columns. Hence the "plate position" is usually also virtual, but just acts as an identifier of the plate.

The registered plate type can be queried as follows:

```
R> plate_type(TRUE) # shows all existing user-defined plates

[1] "CUSTOM:DM" "CUSTOM:FM"

R> listing(wells(plate = c("CUSTOM:FM", "CUSTOM:DM")))

CUSTOM:FM:
- Growth experiment, fresh mass
- A01: 0.16M NaCl
  A02: 0.32M NaCl
  A03: 0.5M Sorbitol
  A04: Control
CUSTOM:DM:
- Growth experiment, dry mass
- A01: 0.16M NaCl
  A02: 0.32M NaCl
  A03: 0.5M Sorbitol
  A04: Control
```

Note the prefix "CUSTOM:", which is used to distinguish user-defined plate types from those that come with **opm**. The object resulting from `listing` can be output with `to_yaml` or `saveRDS` for externally storing plate types in files. Indeed, please keep in mind that the definition of plate types is only available in the current R session. The definitions will be lost once the session is terminated. This can be circumvented by placing code for loading **opm** and for registering the plates of interest in an `.Rprofile` file or in the global `Rprofile.site` file. See the R documentation on how such files are used. Of course, registering of plates can always be done at the beginning of an R file that makes use of these plates.

With the resulting `potato.dm` and `potato.fm` objects the user can now follow the **opm** work flow for processing PM data. Please continue in Section 4 and the following sections for plotting and statistical analysis of the estimated curve parameters.

It is possible to first register the plate, as shown in Section 3.2, and then convert the data *via* `opmx`. This makes sense in conjunction with the "horizontal" format if another ordering of wells should be enforced. Otherwise `opmx` takes the substrate names directly from the data frame in "horizontal" format and registers them after sorting.

3.2. Direct registration of plate types

An example input file that comes with **opm** contains growth-curve data derived from an experiment with two *Escherichia coli* strains (Deutsche Sammlung von Mikroorganismen (DSM) 18039 = K12 and the type strain DSM 30083^{T}) on increasing Glucose concentrations. Here we are dealing with a real plate with real wells, but the registering procedure would be

the same for virtual plates with virtual wells. Thus, it will here be shown how to prepare a plate map and register it as a new plate type. Section 3.2 then shows how to import the data and subsequently convert them to an OPMX or MOPMX object. Each combination of strain and Glucose concentration was repeated twice on the plate. It will thus be shown how to define a numbering of these repetitions suitable for later on using the `split` function to split the object into one object per repetition.

The **opm** package offers several ways to set up a user-defined plate layout. The function `register_plate` is useful for both customised PM plates and measurements from quite different experiments such as growth curves and other kinds of kinetics.

For small data sets it might be feasible to type the substrate allocation manually into a character vector, as done in the following. The short name of the plate type will be "growth", as simply given by the named function argument. Here two arguments of the same name are passed to the function for registering the full plate name on the one hand and the well mapping on the other hand in a single call:

```
R> register_plate(
     growth = c(
       A01 = "Negative Control #1", A02 = "10mM Glucose #1",
       A03 = "20mM Glucose #1", A04 = "50mM Glucose #1",
       A05 = "100mM Glucose #1", A06 = "200mM Glucose #1",
       B01 = "Negative Control #2", B02 = "10mM Glucose #2",
       B03 = "20mM Glucose #2", B04 = "50mM Glucose #2",
       B05 = "100mM Glucose #2", B06 = "200mM Glucose #2",
       C01 = "Negative Control #3", C02 = "10mM Glucose #3",
       C03 = "20mM Glucose #3", C04 = "50mM Glucose #3",
       C05 = "100mM Glucose #3", C06 = "200mM Glucose #3",
       D01 = "Negative Control #4", D02 = "10mM Glucose #4",
       D03 = "20mM Glucose #4", D04 = "50mM Glucose #4",
       D05 = "100mM Glucose #4", D06 = "200mM Glucose #4"
     ),
     growth = "Growth on Glucose"
   )
R> listing(wells(plate = "custom:growth"))
```

However, manually entering the well mapping is error prone and not efficient when dealing with data sets containing more than a few wells. Alternatively, a user-designed plate can also be registered with a plate map given as matrix. The matrix then directly represents the allocation of the used substrates on the plate. Because of the repetitions in the substrate names (note the numbering, which is necessary here to generate unique substrate names, and later on important to split the plate), the texts can be generated with fewer lines of code than above:

```
R> # create constant part of the substrate names
R> growth <- c("Negative Control", "10mM Glucose", "20mM Glucose",
     "50mM Glucose", "100mM Glucose", "200mM Glucose")
R> # create repetitions and assign according numbers
R> growth <- paste(rep(growth, each = 4), rep(1:4, 4), sep = " #")
R> # create matrix that mirrors the plate layout
R> growth <- matrix(growth, nrow = 4, ncol = 6,
```

```
     dimnames = list(LETTERS[1:4], 1:6))
R> # register this plate type and show the result
R> register_plate(growth = growth, growth = "Growth on Glucose")
R> listing(wells(plate = "custom:growth"))
```

Plates with other layouts can be put together in the same way but using other or more or fewer row or column names. Plates with a distinct repetition structure, or no repetitions of substrates at all, can be put together in the same way, if the way a substrate numbering is introduced is either modified or omitted. Instead of a matrix, a data frame could be used as well. We will try this here after showing how to delete a plate type by providing a `NULL` argument:

```
R> register_plate(growth = NULL)
R> growth <- as.data.frame(growth)
R> register_plate(growth = growth, growth = "Growth on Glucose")
R> listing(wells(plate = "CUSTOM:GROWTH"))
```

3.3. Input of TECAN data

The *E. coli* data for which we have registered a full plate name and a mapping from well coordinates to substrate names in Section 3.2 are contained in an exemplar input file that comes with **opm**. It can be found, and input into R, as follows:

```
R> tecan.file <- opm_files("growth")
R> tecan.file <- grep("tecan", tecan.file, ignore.case = TRUE, value = TRUE)
R> tecan <- read.table(tecan.file)
R> head(tecan)
```

```
  V1    V2    V3    V4    V5    V6    V7
1 <> 1.000 2.000 3.000 4.000 5.000 6.000
2  A 0.087 0.088 0.087 0.088 0.085 0.084
3  B 0.087 0.088 0.087 0.086 0.087 0.085
4  C 0.083 0.082 0.081 0.083 0.079 0.077
5  D 0.083 0.083 0.081 0.082 0.080 0.079
6 <> 1.000 2.000 3.000 4.000 5.000 6.000
```

This file was output by an Infinite® F200 PRO instrument as distributed by the TECAN corporation. After recording the data, the Magellan™ software generates such a file *via* the "save as `.asc`" option in the "edit" menu.

The resulting format is not particularly useful within R but can be converted using the "rectangular" mode of `opmx`:

```
R> tecan <- opmx(tecan, "rectangular", plate.type = "growth", position = "1A",
     interval = 1)
R> tecan
```

```
Class                       OPM
From file
```

```
Hours measured              71
Number of wells             24
Plate type                  CUSTOM:GROWTH
Position                    1A
Setup time                  Fri Nov 27 01:03:04 2015
Metadata                    0
Aggregated                  FALSE
Discretized                 FALSE
```

Note that we have to refer to the previously registered plate type, "growth". If several plates of this plate type are to be dealt with, the `position` argument is important for identifying each plate. The format of the `position` entry can, in principle, be arbitrarily selected by the user, but the shown format is the recommended one, i.e. an integer followed by a single letter.

The optional `interval` argument provides the time interval between two consecutive measurements. In the given example one measurement per hour was recorded, thus the default fits perfectly when assigning integers in increasing order, starting at 0. If `interval` is stated explicitly, it is ideally provided in hours. Time series with irregular intervals can be entered with the same argument (by directly providing each time point). See the manual for further details on the usage of the `interval` argument.

The "rectangular" formats also comprise time points separated by rows containing only `NA` values or empty strings, time points defined as a given number of rows, and measurements without indicators of well coordinates. Again, see the manual for more information.

The generated OPM object can now be split according to the repetition structure of the wells (as apparent from the substrate names we have stored by calling `register_plate`), and metadata can be added that describe each resulting plate:

```
R> tecan <- split(tecan)
R> metadata(tecan) <- data.frame(Replicate = c(1, 2, 1, 2),
     Strain = rep(c("DSM30083", "DSM18039"), each = 2),
     stringsAsFactors = FALSE)
R> dim(tecan)

[1]  4 72  6
```

3.4. Input of PerkinElmer data

An example file for an EcoPlate™ analysis not conducted with an OmniLog® PM system but measured with a PerkinElmer, Inc., plate reader is included in **opm**. It contains anonymised end-point measurements sent to us by an **opm** user who asked for a data-conversion approach. The output of that reader is regular but contains a tab-separated part as well as bottom part with the same measurements in a different layout as well as technical information on the run:

```
R> pe.file <- opm_files("growth")
R> pe.file <- grep("perkin_elmer", pe.file, ignore.case = TRUE, value = TRUE)
R> pe <- readLines(pe.file)
R> cat(head(pe), sep = "\n")
```

```
Plate        Repeat        Well        Type        Time        595nm (A)
1        1        A01        M        00:00:23,44        0,094
1        1        A02        M        00:00:24,16        0,080
1        1        A03        M        00:00:24,88        0,048
1        1        A04        M        00:00:25,60        0,088
1        1        A05        M        00:00:26,32        0,065
```

```
R> cat(tail(pe), sep = "\n")
```

```
Assay ID:  ..... 7318
Measured on ..... 15.03.2015 18:08:35
Notes for the assay run


Run started by: christoph.probst
```

Moreover, the comma has been used decimal separator (the data were probably measured with the language of the operating system set to German). Special care is needed when inputting such data, but as we know that two measurements of a 96-well-plate are contained, we can actually use the `read.table` function that comes with R.

```
R> pe <- read.table(pe.file, strip.white = TRUE, sep = "\t",
     # fix the column data types and skip some columns
     colClasses = c("integer", "integer", "character", "NULL", "NULL", "numeric"),
     # re-use names but adapt last column to opmx()
     col.names = c("Plate", "Repeat", "Well", "Skip", "Skip", "T_1"),
     # skip all irregular rows after the tab-separated part
     nrows = 192L, header = TRUE,
     # use comma as decimal separator
     dec = ",")
R> head(pe)
```

```
  Plate Repeat Well   T_1
1     1      1  A01 0.094
2     1      1  A02 0.080
3     1      1  A03 0.048
4     1      1  A04 0.088
5     1      1  A05 0.065
6     1      1  A06 0.098
```

Before the conversion to an `OPMS` object, we register the plate type as a user-defined plate type even though EcoPlate™ is identical to it. This makes sense because the scale is distinct from an OmniLog® instrument, and the distinct plate types prevent comparing apples and oranges.

```
R> # copy the plate type
R> register_plate(myeco = wells(plate = "eco"))
R> # convert to OPMS
R> pe <- opmx(pe, "horizontal", full.name = c("custom:myeco" = "Ecoplate"),
     position = "Repeat", well = "Well", file = pe.file)
R> pe <- do_aggr(pe)
```

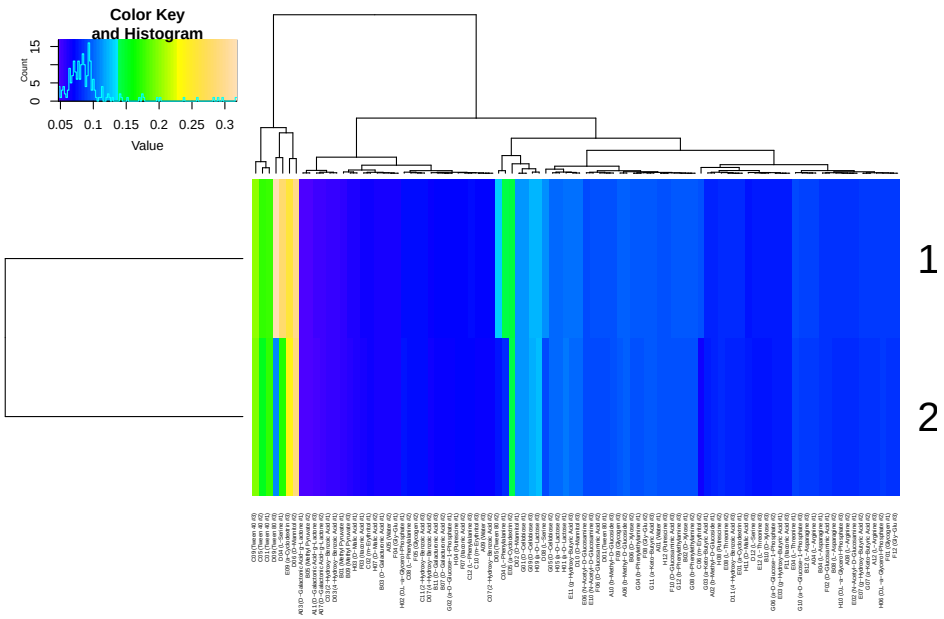

Figure 1: Heat map of the EcoPlate™ well measured using a PerkinElmer, Inc., plate reader and not yet split into the within-plate repetitions. This is not an annotated plot, as no metadata have been added (the data are anonymous anyway).

It it useful to add the file name to the `csv_data` to ease the identification of the plates. Because we have only end-point measurements here, aggregating means copying, but it needs to be done to enable the application of the methods that work on aggregated data.

The code specific to irregular input formats ends here. The next step is typical for all EcoPlate™ data. As this plate type contains repetitions of substrates, it is often used to measure distinct organisms or experimental conditions, or distinct replicates of the same organism or experimental condition, in one run. The `split` function for `OPMX` objects is designed for separating such plates into their individual components, allowing for their subsequent comparison. The effect can be illustrated with heat maps drawn before and after the split.

```
R> heat_map(pe, NULL) # no metadata available
```

And now the split:

```
R> pe.sep <- split(pe)
R> heat_map(pe.sep, NULL, extract.args = list(rm.num = TRUE))
```

The `rm.num` argument is necessary to strip the numbers from the end of the substrate names. These numbers are used in **opm** for indicating the within-plate repetitions and are the basis for the `split` operation.

The next step in a real analysis would be to add metadata. To better identify each plate, the file name should be added to the keys within `csv_data` for finding metadata:

```
R> opm_opt(csv.keys = c(opm_opt("csv.keys"), "File"))
```

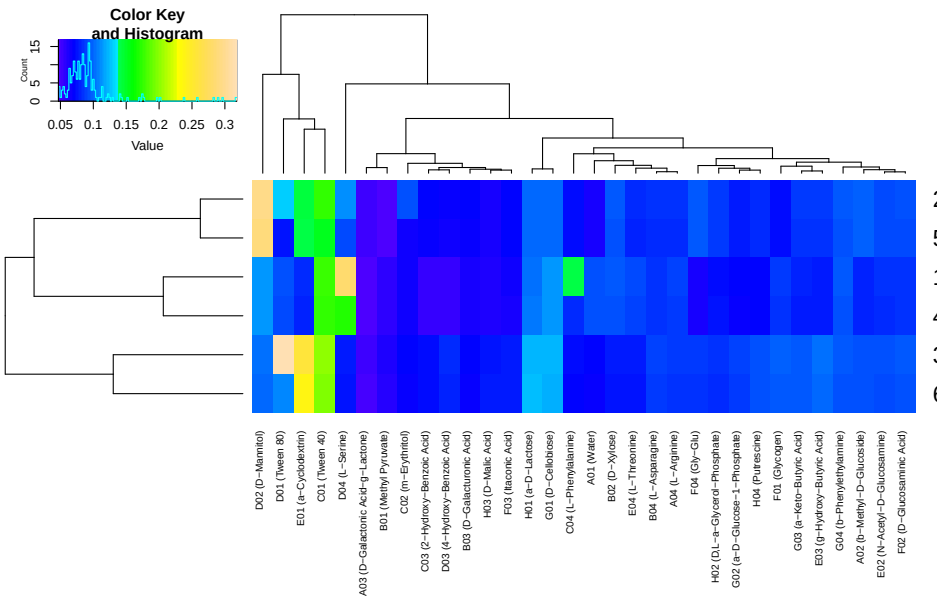

Figure 2: Heat map of the EcoPlate™ well measured using a PerkinElmer, Inc., plate reader and split into the within-plate repetitions. This makes sense if the three are independent repetitions of the samples, or distinct samples. Again, this is not an annotated plot, as no metadata can be added (except for those that indicate the split structure).

This works because of the use of the `file` argument in the call of `opmx` conducted above. Apart from that, the processing of metadata is as usual. Note that after the application of `split` the `csv_data` entries for several plates, i.e. those that come from the same original plate, are identical.

4. Visualisation of growth curves

Visualising raw measurements of growth curves with the methods intended for PM data is straightforward, but some adaptations are necessary due to the deviations between the distinct kinds of data. For instance, the expected maximum for PM data can seldom be used for delimiting the `y` axes, the data are not measured in OmniLog® units, and a negative control might not be present:

```
R> library("gridExtra")
R> plot.fm <- xy_plot(potato.fm, theor.max = FALSE, rcr = 1,
     include = "Genotype", main = list(in.parens = FALSE),
     ylab = "Fresh cell mass [mg]", neg.ctrl = FALSE)
R> plot.dm <- xy_plot(potato.dm, theor.max = FALSE, rcr = 1,
     include = "Genotype", main = list(in.parens = FALSE),
     ylab = "Dry mass [mg]", neg.ctrl = FALSE)
R> grid.arrange(plot.fm, plot.dm, ncol = 2)
```

The result is shown in Figure 3. The TECAN data (which contain a negative control) can be visualised in the same way as the potato data, yielding Figure 4. Note the `rm.num` argument,

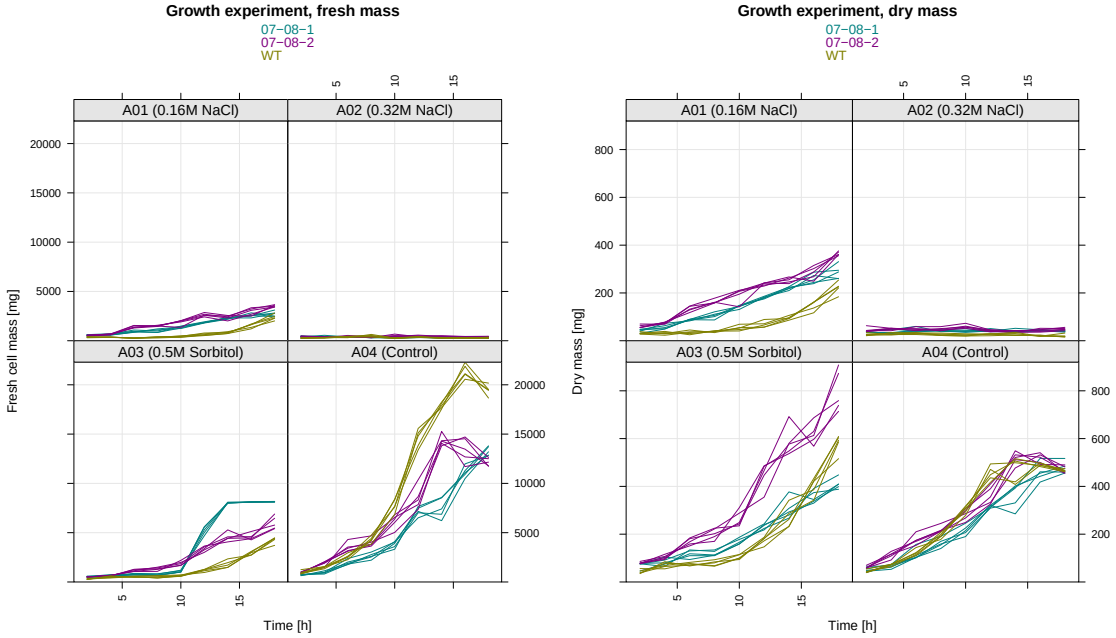

Figure 3: Potato cell-line growth measurements, recorded as fresh mass (left) and dry mass (right), visualised using the `xy_plot` method. See Section 2.7 and Section 3.7 in the main tutorial for details on this kind of plotting. Considering the fresh weight, the plot indicates that the wild type grows better than the genetically modified cell lines under non-stress (control) conditions. It also indicates that the stresses impair growth but that the genetically modified cells grow better then the wild type under moderate stress conditions. The results for the dry mass are similar except for the behaviour under Sorbitol stress.

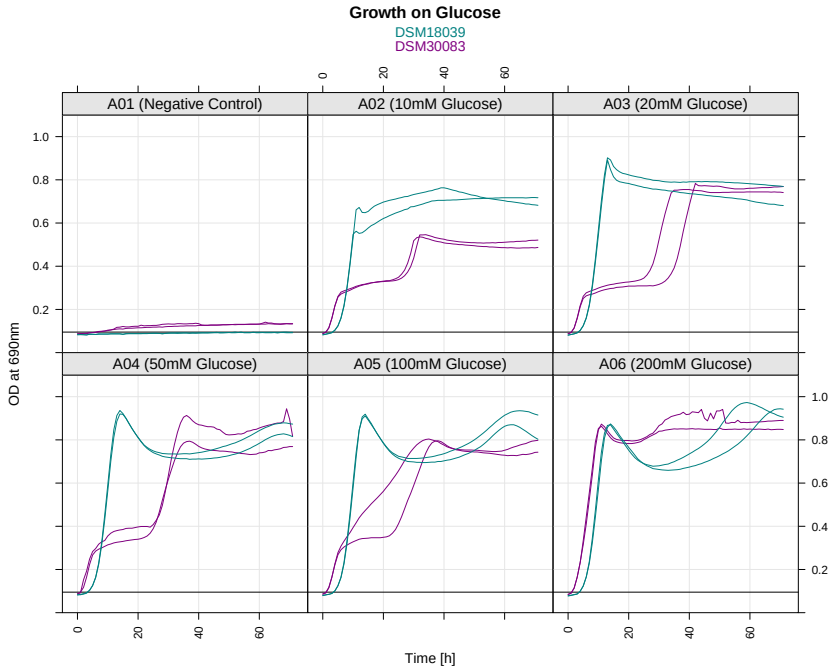

Figure 4: Growth of two *E. coli* strains on Glucose, visualised using the `xy_plot` method. See Section 2.7 and Section 3.7 in the main tutorial for details on this kind of plotting. The plot indicates that one of the strains outgrows the other unless high concentrations of Glucose are applied.

which causes the removal of the numbering from the end of the full well names (which is not needed any more after applying the `split` function as described in Section 3.3):

```
R> xy_plot(tecan, theor.max = FALSE, include = "Strain",
     main = list(in.parens = FALSE), ylab = "OD at 690nm", rm.num = TRUE)
```

5. Estimating parameters from growth curves

The next step is the estimation of curve parameters using `do_aggr`. See Section 2.5 and Section 3.5 in the main tutorial for details on aggregation methods. The main difference with respect to user-entered growth-curve data is that these may contain much fewer time points than PM measurements. Hence the question arises which spline estimation is optimal for such measurements, because all spline-fitting procedures have been optimised for estimating parameters from PM data. The following (informal) comparison, however, illustrates the differences, if any, between the methods. We first define a helper function for plotting parameters obtained with distinct approaches:

```
R> # A function not normally needed by the user!
R> plot_param_diff <- function(...) {
     x <- list(...)
     if (is.null(names(x)))
```

```
    names(x) <- seq_along(x)
  for (param in rev(param_names())) {
    y <- lapply(lapply(x, aggregated, param, ci = FALSE), unlist)
    n <- names(y)
    for (i in seq_along(y)[-1L])
      for (j in seq.int(1L, i - 1L)) {
        plot(y[[i]], y[[j]], pch = 19L, col = "darkgrey",
          main = sprintf("%s, %s/%s", param, n[[i]], n[[j]]))
        abline(line(y[[i]], y[[j]]), lty = "dashed")
      }
  }
  invisible(NULL)
}
```

In the next step, the distinct spline-fitting approaches have to be applied.

```
R> tp.tecan <- do_aggr(tecan, method = "splines",
     options = set_spline_options(type = "tp.spline"))
R> p.tecan <- do_aggr(tecan, method = "splines",
     options = set_spline_options(type = "p.spline"))
R> sm.tecan <- do_aggr(tecan, method = "splines",
     options = set_spline_options(type = "smooth.spline"))
```

After these preparations the methods can visually be compared as follows.

```
R> old.par <- par(mfrow = c(4, 3), mar = c(2, 2, 3, 2))
R> plot_param_diff(Tp = tp.tecan, P = p.tecan, Sm = sm.tecan)
R> par(old.par)
```

Apparently the distinct spline-fitting methods yield approximately the same estimates for AUC and A but not necessarily for the other parameters. A closer investigation of the deviations in the smoothing-spline results for λ and μ indicated that these are due to overfitting. The same issue has been observed with PM data, see Section 2.5 in the main tutorial.

Other differences between the methods might reflect biological issues that render it difficult to interpret the curves in terms of the four parameters. For instance, Figure 4 shows curves with two positive shifts instead of a single one. Such curves might better be described by two λ and two μ values instead of single ones. We have observed, however, that thin-plate splines usually pick the stronger of the two shifts, which is plausible. Note also that neither the interpretation of AUC nor the one of A suffer from that kind of problem. Selecting an A value that makes sense is hampered by an increase in intensity followed by a decrease again, as also visible in Figure 4. Thin-plate splines usually pick the actual maximum (of a smoothed curve), which is biologically reasonable.

We conclude that in the case of comparatively few data points the same method preferences should be applied as for PM data. For the forthcoming analyses we accordingly restrict ourselves to the results obtained with thin-plate splines.

```
R> tecan <- tp.tecan
R> rm(tp.tecan, p.tecan, sm.tecan) # tidy up
R> potato.fm <- do_aggr(potato.fm, method = "splines",
```

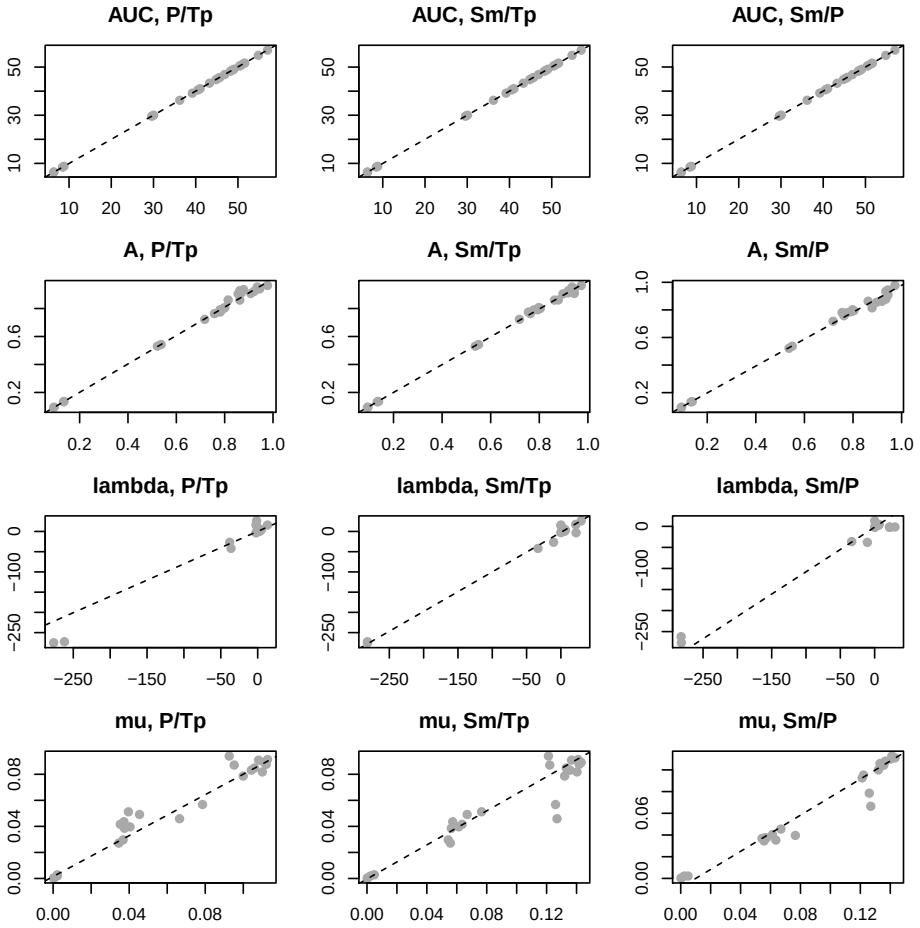

Figure 5: Comparison of curve parameters estimated using different spline-fitting options. There are obviously quite high correlations between the AUC values and comparatively high correlations for A. In the case of λ and μ the correlations are lower, with thin-plate and P-splines corresponding somewhat more strongly to each other than to smoothing splines. Negative λ values can occur in the case of negative reactions.

```
     options = set_spline_options(type = "smooth.spline"))
R> potato.dm <- do_aggr(potato.dm, method = "splines",
     options = set_spline_options(type = "smooth.spline"))
```

Plots based on the estimated curve parameter could now be generated, such as heat maps or confidence-interval plots. As there is no difference to plotting PM data, we refer to Section 2.8 and Section 3.8 in the main tutorial for details.

6. Statistical analysis of growth curves

In the following we will use `opm_mcp` to assess whether the conclusions from the graphical analysis (Figure 4 and Figure 3) can be confirmed for the example data. The `opm_mcp` method is extensively documented in Section 2.9 and particularly Section 3.9 of the main tutorial, hence we here restrict ourselves to just the necessary function calls.

The following code answers the main question regarding the potato data sets, i.e. for which treatment (well) significant or insignificant differences between the genotypes are found, and how large the according effect size is.

```
R> dm.mcp <- opm_mcp(potato.dm, ~ J(Well, Genotype), m.type = "aov",
     linfct = c(Pairs.Well = 1), max = 7, in.parens = FALSE)
R> fm.mcp <- opm_mcp(potato.fm, ~ J(Well, Genotype), m.type = "aov",
     linfct = c(Pairs.Well = 1), max = 7, in.parens = FALSE)
```

Using the dedicated `plot` method, the results can be visualised as demonstrated in Figure 6.

```
R> old.par <- par(mfrow = c(2, 1), mar = c(1, 15, 2, 1))
R> plot(fm.mcp)
R> plot(dm.mcp, main = "")
R> par(old.par)
```

Results obtained with `opm_mcp` confirm the suspicion from Figure 3 that the stresses impair growth but that, when measured using fresh mass (Figure 6, upper section), the genetically modified cells grow better then the wild type under moderate stress conditions. In contrast, the wild type outgrows the genetically modified lineages under non-stress (control) conditions. In the dry-mass measurements (Figure 6, lower section) some differences are less apparent. For instance, there is no significant difference between the lineages under control conditions. See the publication by Vaas *et al.* (2013) for an in-depth interpretation of these results.

The TECAN data can be analysed in an analogous fashion.

```
R> tecan.mcp <- opm_mcp(tecan, ~ J(Well, Strain), m.type = "aov",
     linfct = c(Pairs.Well = 1), full = FALSE)
```

Plotting the object accordingly yields Figure 7:

```
R> old.mar <- par(mar = c(3, 15, 3, 2))
R> plot(tecan.mcp)
R> par(old.mar)
```

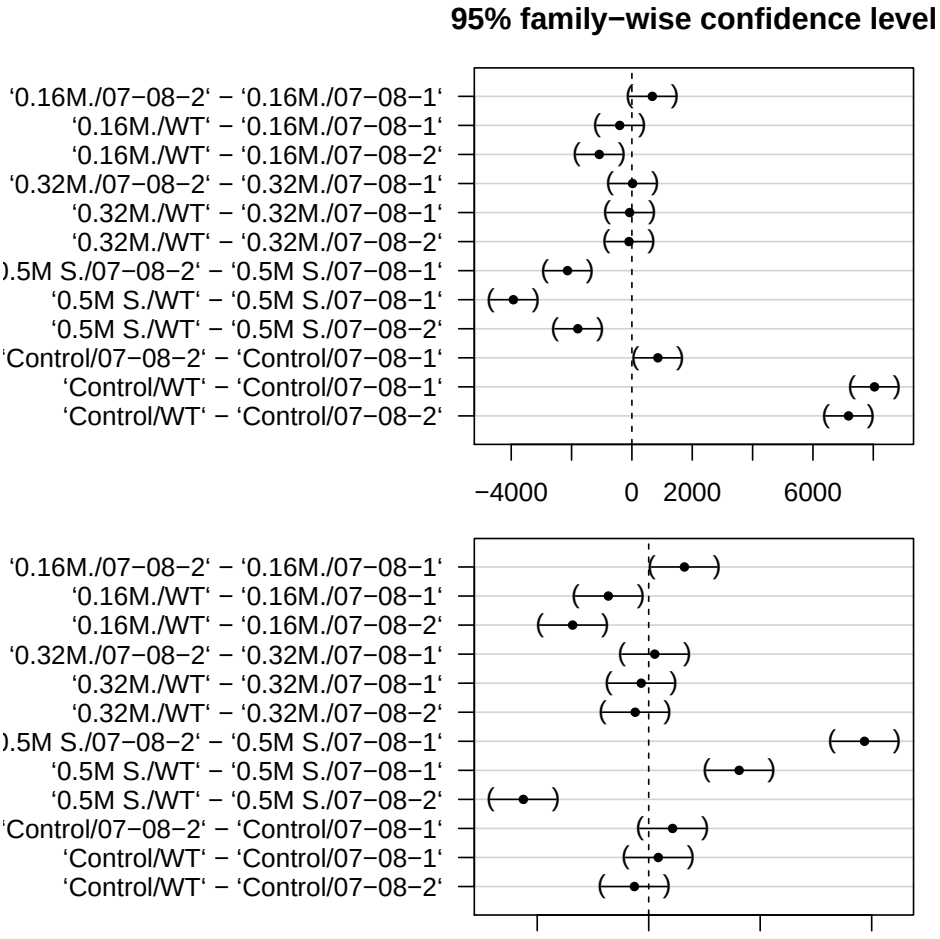

Figure 6: Point estimates and 95% confidence intervals in a “Pairs”-type comparison of group means for the fresh-mass (upper section) and dry-mass (lower section) potato cell-line example data. Significant differences are those whose confidence intervals do not cross the dotted line at `x = 0`; effect sizes are also easily visible. Compare the outcome with Figure 3 and see the main text for an interpretation.

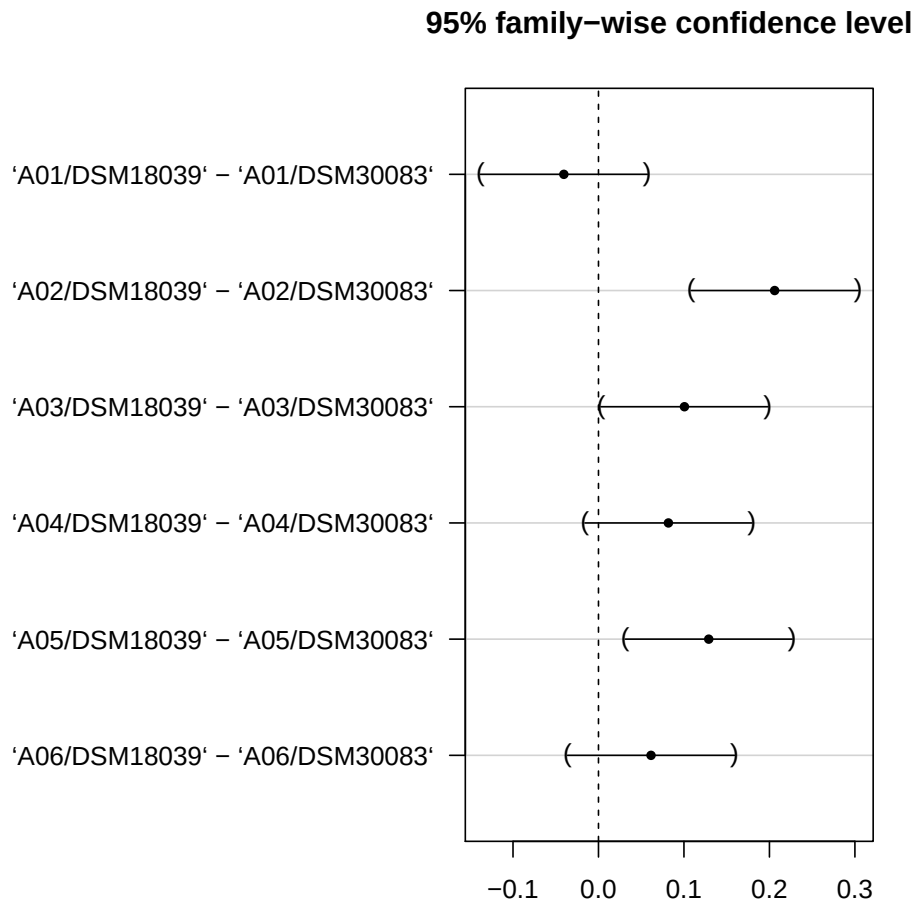

Figure 7: Point estimates and 95% confidence intervals in a “Pairs”-type comparison of group means for the TECAN example data obtained from two *E. coli* strains. Significant differences are those whose confidence intervals do not cross the dotted line at `x = 0`; effect sizes are also easily visible. Compare the outcome with Figure 4 and see the main text for an interpretation.

The plot in Figure 4 indicated that one of the *E. coli* strains outgrows the other unless high concentrations of Glucose are applied. This can be confirmed with the `opm_mcp` analysis (Figure 7), as significant differences between the two strains only occur for moderate Glucose concentrations (wells "A02" and "A03") but neither in the control treatment nor for high concentrations of the sugar.

7. Acknowledgements

The author is grateful to Victoria Michael (Deutsche Sammlung von Mikroorganismen und Zellkulturen (DSMZ)) for providing growth curves measured with a TECAN instrument. Cordial thanks are addressed to Lea A.I. Vaas (DSMZ) for providing the potato growth measurements and background information on these data, as well as for clarifying some technical issues related to the TECAN data. This tutorial did benefit from discussions on the application of spline-fitting with Benjamin Hofner (Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)) and Johannes Sikorski (DSMZ).

References

Vaas LAI, Marheine M, Sikorski J, Göker M, Schumacher HM (2013). "Impacts of pr-10a Overexpression at the Molecular and the Phenotypic Level." *International Journal of Molecular Sciences*, **14**, 15141–15166. doi:10.3390/ijms140715141. URL http://www.mdpi.com/1422-0067/14/7/15141.

Affiliation:

Markus Göker
Leibniz Institute DSMZ – German Collection of Microorganisms and Cell Cultures
Braunschweig

Telephone: +49/531-2616-272
Fax: +49/531-2616-237
E-mail: markus.goeker@dsmz.de
URL: www.dsmz.de