opm: An R Package for Analysing OmniLog® Phenotype MicroArray Data

Johannes Sikorski
Leibniz Institute DSMZ

Lea A.I. Vaas
Leibniz Institute DSMZ

Benjamin Hofner
Universität
Erlangen-Nürnberg

Markus Göker
Leibniz Institute DSMZ

Abstract

The OmniLog® Phenotype Microarray system is able to monitor simultaneously, on a longitudinal time scale, the phenotypic reaction of single-celled organisms such as bacteria, fungi, and animal cell cultures to up to 2,000 environmental challenges spotted on sets of 96-well microtiter plates. The phenotypic reactions are recorded as respiration kinetics with a shape comparable to growth curves. Tools for storing the curve kinetics, aggregating the curve parameters, recording associated metadata of organisms and experimental settings as well as methods for analysing graphically and statistically these highly complex data sets are increasingly in demand.

The **opm** R package facilitates management, visualization and statistical analysis of Phenotype Microarray data. Raw measurements can be easily input into R, combined with relevant meta-information and accordingly analysed. The kinetics can be aggregated by estimating curve parameters using several methods. Some of them have been specifically adapted for obtaining robust parameter estimates from Phenotype Microarray data. Containers of **opm** data can easily be queried for and subset by using the integrated metadata and other information. The raw kinetic data can be displayed with customized plotting functions. In addition to 95% confidence plots and enhanced heat-map graphics for visual comparisons of the estimated curve parameters, the package includes customized methods for user-defined simultaneous multiple comparisons of group means. It is also possible to discretize the curve parameters and to export them for reconstructing character evolution or inferring phylogenies with external programs. Tabular and textual summaries suitable for, e.g., taxonomic journals can also be automatically created and customized. Export and import in the YAML (or JSON) markup language (or as character-separated values) facilitates the data exchange among labs. All methods are exemplified using real-world data sets that are part of the **opm** R package or are included in the accompanying data package **opmdata**.

This is the tutorial of **opm** in the version of September 11, 2013.

Keywords: Bootstrap, Cell Lines, **grofit**, Growth Curves, **lattice**, Metadata, Microbiology, Respiration Kinetics, Splines, YAML, JSON, CSV.

1. Introduction

1.1. Preamble for "eager to start" readers

Readers who want to jump right into examples for applying **opm** to their data will find an overview of what the package can do for them in Figure 2. Next, Figure 5 should be looked at, as it lists the names of the functions that can be used in each step of the possible **opm** work flows. Examples for each step would then be found in the according subsections of Section 3. An overview of these sections is provided in Section 2.1.

The single most important problem users reported to us when applying **opm** was that the input files could not be read. This was uniformly due to the use of multi-plate CSV files. But the most recent versions of the OmniLog® software can batch-export one plate per CSV file, and **opm** can split multi-plate CSV into files that can be input by the package. See Section 2.2 for details and Section 3.2 for a usage example.

Details on the scientific background could well be skipped during a first reading. The interested user would nevertheless find them in Section 1.2, including references for important methods.

All web resources regarding **opm** are linked on its main website `http://opm.dsmz.de/`.

1.2. Scientific introduction

The phenotype is regarded as the set of all types of traits of an organism (Mahner and Kary 1997). The phenotype is of high biological relevance, as it is the phenotype which is the object of selection and, hence, is the level at which evolutionary directions are governed by adaptation processes (Mayr 1997). It is also the phenotype which is of direct relevance to humans, for example in exploiting microorganisms for industrial purposes or in the combat of pathogenic organisms (Broadbent, Larsen, Deibel, and Steele 2010; Mithani, Hein, and Preston 2011). In the study of single-cell living beings, such as bacteria, fungi, plant or animal cells, it is an important field of research to study the phenotype by measuring physiological activities as a response to environmental challenges. These can be single carbon sources, which may be utilized as nutrients and hence trigger cellular respiration, or substances such as antibiotics, which may slow down or even inhibit cellular respiration, indicating a successful inhibitory effect on potentially pathogenic organisms. The intensity of cellular respiration correlates with the production of NADH engendering a redox potential and thus a flow of electrons in the electron transport chain. To measure cellular respiration in an experimental assay, this flow of electrons can be utilized to reduce a tetrazolium dye such as tetrazolium violet, thereby producing purple colour (Bochner and Savageau 1977). In principle, the more intense the colour, the larger the physiological activity.

The Phenotype MicroArray (PM) system is capable of measuring a large number of phenotypes in a high-throughput system that uses such as tetrazolium detection approach. About 2,000 distinct physiological challenges, such as the metabolism of single carbon sources for energy gain, the metabolism under varying osmolyte concentrations, and the response to varying growth-inhibitory substances are included in the PM microtiter plates (Bochner, Gadzinski, and Panomitros 2001; Bochner 2009). The system is applicable, in principle, to each kind of cultivated cells and also to environmental probes, even though some kinds of cells, such as those from plant cell cultures, well cause a reduction of the dye but are too large to be handled in the 96-well layout (Vaas, Marheine, Sikorski, Göker, and Schumacher 2013b). The OmniLog® PM system records the colour formation in an automated setting (every 15 minutes) throughout the duration of the experiment, which may last up to several days. Thus

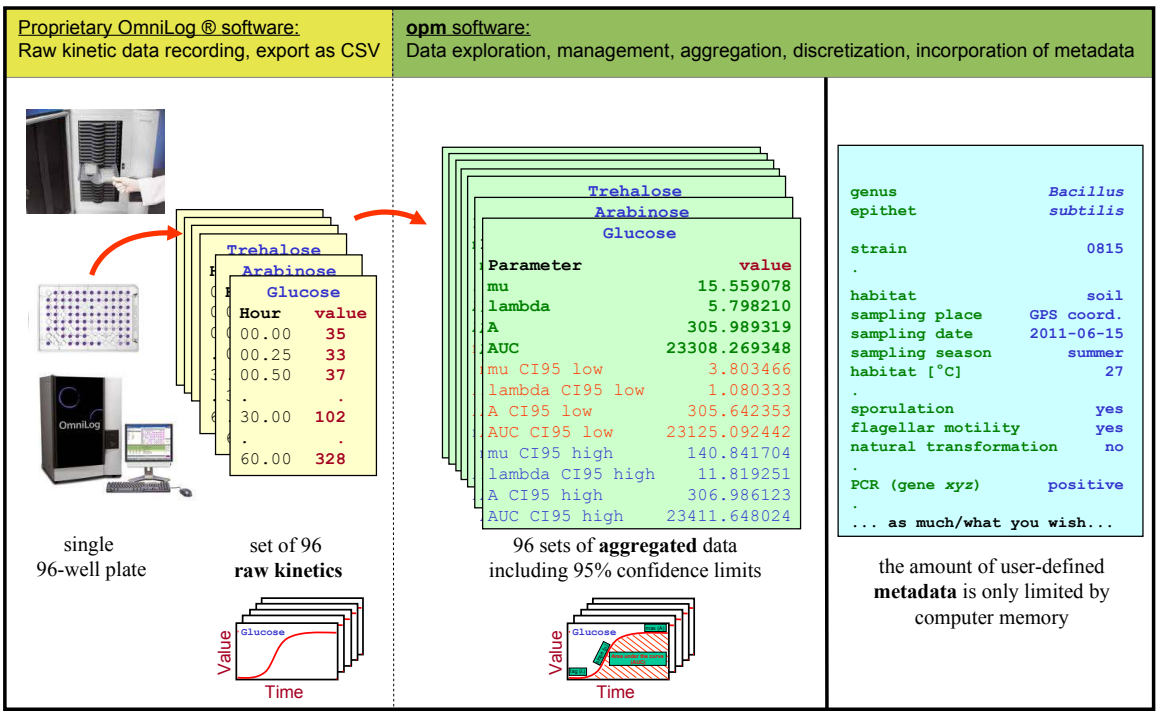

Figure 1: Overview of the data assembly from an PM experiment and the possible additions using **opm**. The raw colour-formation values result in sets of 96 raw kinetics per plate. Using **opm**, they can be augmented by the information coded in the shape characteristics. This yields 96 sets of parameters per plate, each containing four robustly estimated parameters that describe distinct aspects of the respective curve shape. The **opm** package also offers tools for further combining this bundle of raw, aggregated and also discretized data of each single kinetic with meta-information on the organisms and/or experiments. Based on this meta-information, a variety of visual and statistical comparison tools for either the raw or the aggregated data are available in **opm**.

the experimenter ends up with high-dimensional sets of longitudinal data, the PM respiration kinetics. For a detailed introduction into the experimental setup for obtaining OmniLog® PM respiration kinetic data we refer to the OmniLog® website (http://www.biolog.com/) and the associated hardware and software manuals. Briefly, 96-well microtiter plates with substrates, dye, and bacterial cells are loaded into the OmniLog® reader, a hardware device which provides the appropriate incubation conditions and also automatically reads the intensity of colour formation during tetrazolium reduction. The OmniLog® reader is driven by the *Data Collection* software. The stored results files, which are in a proprietary format, are then imported into the *Data Management*, *File Management/Kinetic Analysis*, and *Parametric Analysis* software packages for data analysis.

In the case of positive reactions, the kinetics are expected to appear as (more or less regularly) sigmoidal curves in analogy to typical bacterial growth curves (Figure 4). The intrinsic higher level of data complexity contains additional valuable biological information which can be extracted by exploring the shape characteristics of the recorded curves (Brisbin, Collins, White, and McCallum 1987). These curve features can, in principle, unravel fundamental differences or similarities in the respiration behaviour of distinct organisms, which cannot be identified by the traditional end-point measurements alone. But the meta-information of interest on the organisms and experimental conditions must also be available for a biologically meaningful data analysis and an according statistical assessment.

The motivation for the here presented **opm** package originated from (i) the need to overcome the limited graphical and analysis functions of the proprietary OmniLog® PM software and (ii) the desirability of an analysis system for this kind of data in a free statistical software environment such as R (R Development Core Team 2011). At the moment, the visualisation of the kinetics by the proprietary OmniLog® PM software is of limited quality, especially when simultaneously comparing the curves from more than two experiments. Its calculation of curve parameters is rather crude (Vaas, Sikorski, Michael, Göker, and Klenk 2012; Bi-OLOG Inc. 2009). The statistical treatment of raw kinetic data and curve parameters would involve cumbersome manual and hence error-prone manipulations of data in typical spreadsheet applications before they may be imported into appropriate statistical software. Finally, the amount of organismic or experimental metadata that can be added to the raw data is extremely limited.

Based on a previous study (Vaas *et al.* 2012) the here presented **opm** package (Vaas, Sikorski, Hofner, Fiebig, Buddruhs, Klenk, and Göker 2013a) offers functionalities for a fast, robust and comprehensive evaluation of PM respiration kinetics suitable for a wide range of experimental questions.

Using customized input functions, raw kinetic data can be transferred into R, stored as S4 objects (Chambers 1998) containing single or multiple OmniLog® PM plates and further processed. The package features the statistically robust calculation and attachment of aggregated curve parameters including their (bootstrapped) confidence intervals. Moreover, infrastructure is provided to merge this with any kind of additional metadata. These complex data bundles can then be exported in YAML format (http://www.yaml.org/), which is a human-readable data serialization format that can be read by most common programming languages and facilitates fast and easy data exchange between laboratories. Its subset, JSON (http://www.json.org/), can also be used, for instance if a proper YAML parser is unavailable. As **opm** is also able to generate R matrices and data frames, output in CSV (character-separated values) is also easy.

Data evaluation includes the graphical display of the data such as the raw respiration curve kinetics or the confidence intervals of aggregated curve parameters. With sophisticated selection methods the user is able to sort, group and arrange the data according the specific experimental questions in the plotting and analysis framework. Since most addressed experimental questions require to statistically compare not only single curves, but distinct groups of curves, the package provides adapted methods for performing simultaneous multiple comparisons of group means (Bretz, Hothorn, and Westfall 2010). Because the definition of groups using stored metadata is highly flexible, the user is enabled to individually define contrast tests (Hsu 1996).

For further specific graphical or statistical analysis according to the needs of the users, the **opm** package organises and maintains the data such that any additional data exploration using other packages in the R environment are easily applicable.

The work flows described below include the input of raw kinetic data and integration of corresponding metadata, conversion into suitable storage formats, the computation of a set of four parameters sufficient for comprehensively describing the curves' shape (aggregated data), manipulating and querying the constructed objects, visualizing both raw kinetics and aggregated data, statistical comparison of group means, discretization of the curve parameters and corresponding export methods, obtaining additional information the substrates and setting global options.

2. Methods

2.1. Overview

In the following the possible work flows (see Figure 2) for generating an R object that contains the kinetic raw data from one to several OmniLog® plates along with the corresponding metadata of interest, and optionally the aggregated and potentially also discretized curve parameters, are described. It is explained how to analyse either raw data, metadata, aggregated data (curve parameters), or combinations of all of them, as stored in the respective R objects, by graphical and/or statistical approaches.

The raw kinetic data can be exported by the proprietary OmniLog® software *File Management/Kinetic Analysis* as CSV files and imported into the **opm** package using `read_opm()`. This is explained in detail in section Section 2.2, whereas corresponding code examples are found in Section 3.2.

Batch processing many files is also possible, even without starting an interactive R session. This includes storage of the **opm** data in the YAML (or JSON or CSV) format, as detailed in Section 2.3. Example code is given in Section 3.3.

All kinds of PM data can be enriched with metadata (see Figure 1). The underlying principles are described in Section 2.4, whereas example code for metadata management is included in Section 3.4.

To statistically analyse the biological information coded in the shape characteristics of the kinetics, four descriptive curve parameters are estimated, which is explained in detail in Section 2.5, whereas example code for curve-parameter estimation is provided in Section 3.5.

The user may be interested to query or subset the objects generated by **opm**. The underlying

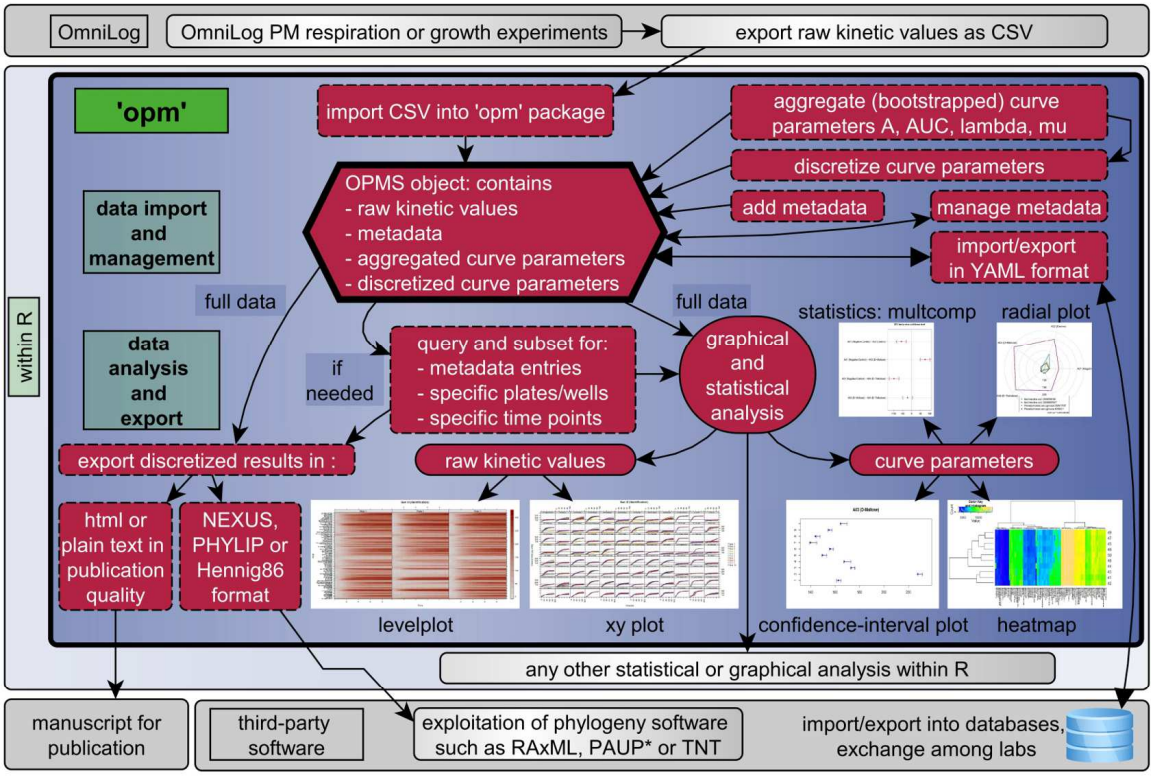

Figure 2: A depiction of the work flows possible within **opm** and its potential interplay with base R, add-on packages for R and third-party software. See Figure 5 for the functions that can be used in the respective steps. The package allows the user full flexibility with respect to the type of information added to the created R objects and to the order of steps in which this is achieved. For example, it is possible to add first the metadata and to perform some of the later described analysis and second to aggregate the raw kinetics and go on with analysis of the aggregated values. Discretization might frequently not be of interest because it causes loss of information. Since experimental frameworks can be imagined where only very limited meta-information is available, it is also feasible to work without metadata at all.

principles are described in Section 2.6, whereas example code for object management can be obtained from Section 3.6.

The raw kinetic data can be plotted either as level plots or as XY plots, as explained in Section 2.7. The estimated curve parameters can be plotted either as confidence-interval plots, radial plots or heat maps, which is described in Section 2.8. See Section 3.7 and Section 3.8, respectively, for plotting example code.

To statistically compare curve parameters, tools for the multiple comparison of groups means have been adapted to PM data. This allows for testing statistical hypothesis involving groups of plates or wells. The principles are described in Section 2.9, and example code is included in Section 3.9.

The aggregated data can be discretized and exported for phylogenetic analysis or reconstruction of character evolution with external phylogeny software. The principle is outlined in Section 2.10, whereas application examples are provided in Section 3.10.1.

The methods implemented in **opm** for classifying reactions as either "positive", "negative" or "weak" (ambiguous) are described in Section 2.11. Example code, including the export of discretization results as publication-ready tables, is included in Section 3.10.3. Textual reports with or without formatting markup can also be produced, as exemplified in Section 3.10.2. The discretization settings can be modified in detail; see Section 3.10.4.

Furthermore, substrate informations can be accessed, including CAS numbers, KEGG and Metacyc IDs as far as they are available. Detailed explanations and code examples are included in the vignette "Working with substrate information in **opm**".

Finally, it is possible to modify settings that have an effect on multiple functions and/or on frequently used arguments. See Section 2.12 for details and Section 3.10.3 for a code example.

After a successful installation of **opm**, the complete R code extracted from this vignette (as well as all other vignette files) can be found *via* `opm_files("doc")`.

2.2. Data import

The proprietary OmniLog® PM data analysis software *File Management/Kinetic Analysis* (BiOLOG Inc. 2009) can export the kinetic raw data from single or multiple plates as CSV files. These contain a small amount of associated run information that has been entered at the interface of the OmniLog® PM *Data Collection* software, which controls the OmniLog® reader. Currently this generation of CSV files involves the creation of intermediary files with the extension "d5e" from the original ones with the extension "oka". For use with **opm**, the raw kinetic data should be exported into a single CSV file for each measured plate. The **opm** package currently does not support the input of several plates from PM-mode runs stored in a single CSV file, but it offers the function `split_files()` for splitting old-style CSV files containing multiple plates. (We refer to the CSV exports from the currently distributed OmniLog® PM *File Management/Kinetic Analysis* software as "old style". Forthcoming versions are expected to export the data in a slightly different CSV format we call "new style". Please contact your local representative of the vendor for the latest software version.)

As of version 0.4-0, **opm** also supports the input of MicroStation® CSV files (frequently used in conjunction with EcoPlate® assay for microbial community analysis) (Vaas *et al.* 2013a). These files contain only end-point measurements but potentially several plates, which can nevertheless be input together with their potentially also rich meta-information.

The easiest way to load the raw kinetic data (as CSV files or as YAML or JSON) into R in a single step is using the function `read_opm()` (see Figure 2). If raw data from only one single-plate OmniLog® PM are imported, the resulting object belongs to the S4 class OPM. This class for holding single-plate OmniLog® PM data originally only includes the (limited) meta-information read from the original input CSV files, but an arbitrary amount of metadata can be added later on (see Figure 2). If multiple plates are imported, the resulting object automatically belongs to the S4 class OPMS. In the OPMS class, data may have been obtained from distinct organisms and/or replicates, but must correspond to the same plate type and must contain the same wells (see Figure 2). The function `read_opm()` has an argument "convert" which controls how sets of plates with distinct types are treated; for instance, the function can return a list of OPMS objects, one for each encountered plate type.

The entire S4 class hierarchy used by **opm** is shown in Figure 3. A number of S3 helper classes are also used by several functions. Users come in direct contact only with the OPM, OPMA, OPMD and OPMS classes (see Figure 5). Once such objects are created they could also be stored in files using `save()` and read again using `load()` but *not* using `dump()` and `source()` instead, respectively. We would nevertheless recommend storage in YAML format.

2.3. Batch conversion of many files

To process and store huge numbers of raw data files, the function `batch_opm()` reads all OmniLog® CSV files (or YAML or JSON files previously generated with **opm**) within a given list of files and/or directories and converts them to **opm** YAML (or JSON or CSV) format. It is possible to let **opm** automatically include metadata (Section 2.4) and aggregated values (curve parameters) (Section 2.5) as well as discretized values (Section 2.11) during this conversion. Alternatively, graphics files containing the output of `xy_plot()` or `level_plot()` can be batch-produced; see Section 2.7 for Details. File selection and unselection using regular expressions or globbing patterns is integrated in the function. The result from each file conversion is reported in detail, and a *demo* mode is available for viewing the attempted file selections and conversions before actually running the (potentially time consuming) conversion process. The package is accompanied by a command-line script "run_opm.R", enabling the users to run the batch conversion without starting an interactive R session. This script is guaranteed to run at least under UNIX-like operating systems. On such systems it can also be run in parallel, making use of multi-core machines.

2.4. Integration of metadata

The interface of the *Data Collection* software of the OmniLog® reader is restricted in size and contains only comparatively few fields for entering accompanying information to each plate such as on the organism under study or the culture conditions. Further, not all of these fields are exported together with the raw measurements. The few metadata that come along with the imported CSV file can be accessed *via* `csv_data()`. But for most experimental designs it is clearly necessary to add much more meta-information to the kinetic data. It has to be emphasized that metadata can include all kind of describing characteristics of the observed organism(s) such as taxonomic affiliation, geographical and/or ecological origin, and of the performed experimental setting such as culture conditions, genetic modifications, physiological information of any kind and so on.

To this end, the **opm** user can integrate the metadata into OPM and OPMS objects using the

function `include_metadata()` (among other functions for this task; see Figure 5). Usually, the metadata are kept in a data frame which can conveniently be saved to, and generated directly from, a CSV file. For an unambiguous match between the raw kinetic data in the OPMS object and the collected metadata, a unique identifier is needed. This is, by default, provided by the combination of *Setup Time* and *Position*, which should unequivocally identify certain plates. *Setup Time* indicates the date and time at the precision of seconds of starting the batch read in the OmniLog® reader. *Position* indicates the position of the plate in the OmniLog® reader. (For instance, *10-A* indicates the plate sliding carriage number 10 in slot A of the reader, but for **opm** the meaning is irrelevant, as these entries only serves as identifiers.) Both *Setup Time* and *Position* are automatically recorded by the OmniLog® reader *Data Collection* software and are exported by the OmniLog® PM *File Management/Kinetic Analysis* software into CSV files together with the raw kinetic data.

To facilitate the user-friendly compilation of metadata, `collect_template()` generates a data frame (and additionally, if requested, a CSV file) in which each line represents a single PM plate. The function `collect_template()` by default automatically includes the *Setup Time* and *Position* of each plate into the data frame or file providing a structured template for the addition of metadata. The user can subsequently add further columns describing any metadata of interest on any PM plate of interest. The resulting data frame can then be queried for the information specific to each plate, and the corresponding row integrated into OPM or OPMS objects using `include_metadata()`. Whereas this function will usually result in non-nested metadata entries, **opm** allows one, in principle, to deal with arbitrarily nested meta-information. Other functions for generating and modifying plate meta-information are listed in Figure 5. Thereby, the amount of meta-information added (and plates analysed) is only limited by the available computer memory.

The user can provide additional information to the metadata data frame on the fly by calling the function `edit()`, which opens the R editor enabling the user to modify and add data. Beside changing the metadata entries by using the R editor, the function `map_metadata()` offers a secure way to map metadata within OPMS objects. The replacement function `metadata()<-` enables the user to set the entire meta-information, or specific entries, directly. If a data frame is used on the right side of the assignment whose number of rows is identical to the number of plates within the OPMS object on the left side, each data-frame row is specifically added to the corresponding plate.

There are no restrictions regarding the stored metadata *values* except for the fact that it usually makes not much sense to store factors. It is safer to store character vectors instead because otherwise conversions might easily result in integer vectors instead of factors. Where appropriate, factors would be created on-the-fly from character vectors by those methods that have to integrate metadata in data frames. A `map_metadata()` method is available that conducts an according cleaning of metadata entries. With respect to the stored metadata *names*, there are only very few restrictions, which are explained in Section 2.12. In contrast to data frames it is not advisable to access metadata entries by position instead of by name.

2.5. Aggregating data by estimating curve parameters

Descriptive curve parameters from the kinetic raw data can be calculated and included in OPM and OPMS objects using the function `do_aggr()`. Curve parameters can be extracted using a spline-based fitting procedure implemented in **opm**. (Extraction of curve parameters

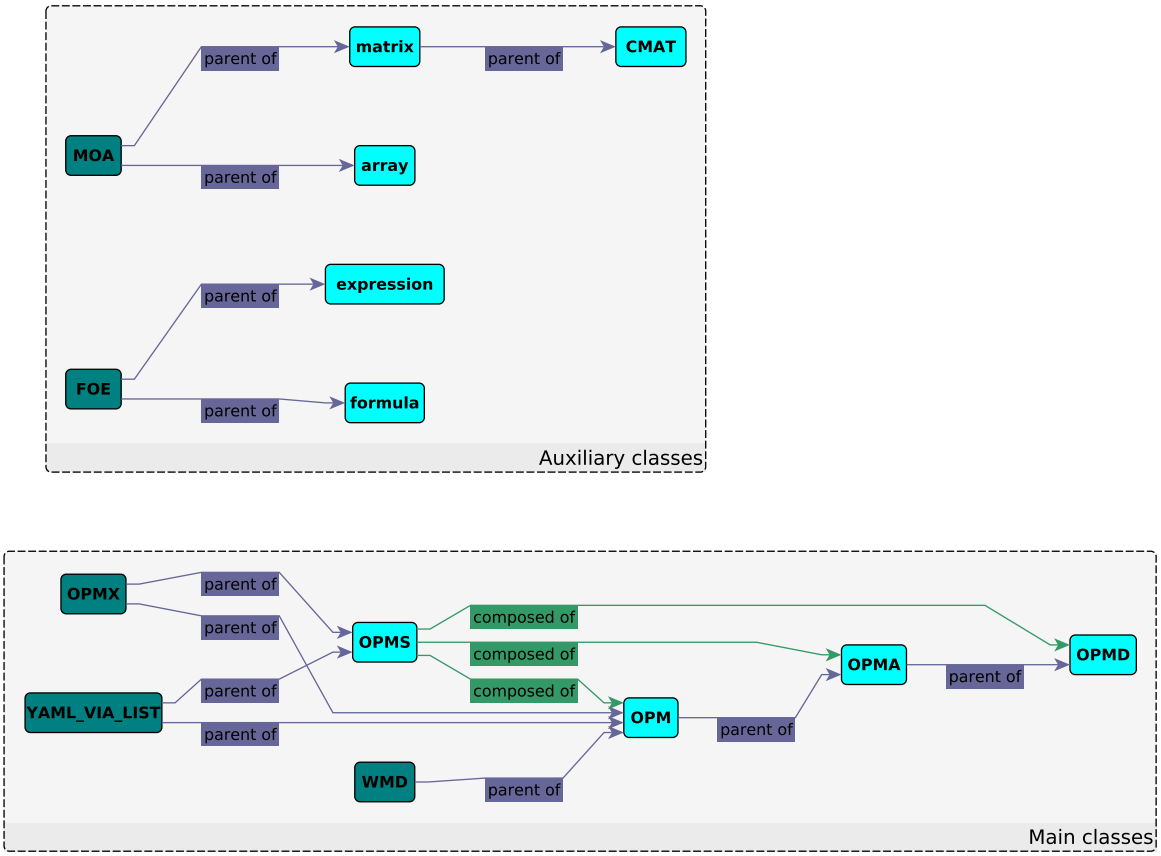

Figure 3: This picture shows the S4 class hierarchy used by **opm**. Class names are shown in bold within the boxes. Boxes with dark background indicate virtual classes, those with light background indicate real classes whose objects can be created and manipulated by some code. Arrows indicate either inheritance relationships (pointing from the parent to its child class) or object composition (pointing from the container class to its element class). Note particularly that OPM, which only contains raw data, csv data and metadata, is the parent class of OPMA, which also contains aggregated data (and has methods for dealing with them). OPMD inherits from OPMA and stores discretized curve parameters in addition to aggregated values. OPMS is a container class that holds OPM, OPMA and/or OPMD objects. These can usually co-occur in a single OPMS object but for some calculations the additional information in OPMA or OPMD objects is strictly required. The query functions `has_aggr()` and `has_disc()` are available for checking from which kinds of objects an OPMS is composed. See their help pages (e.g., `?has_aggr`) and Section 3 for further details. The non-virtual classes in the upper part of the figure are either well-known in R (e.g., matrices) or not directly manipulated by the user (CMAT).

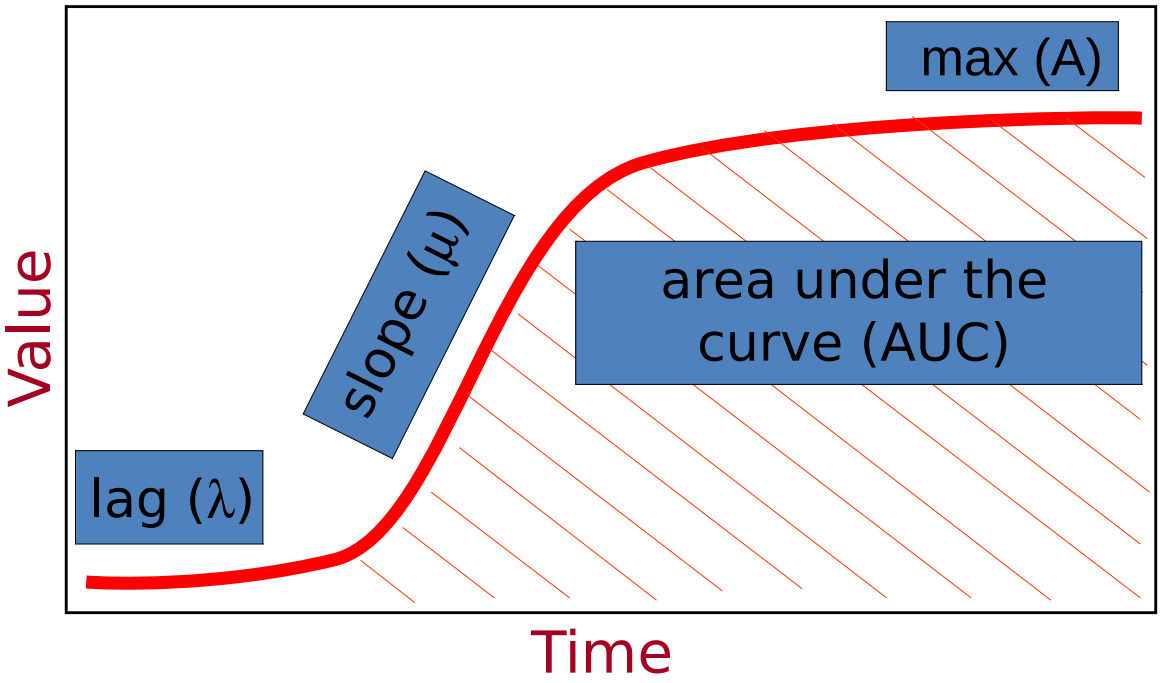

Figure 4: A schematic depiction of a typical respiration (or growth) curve and the parameters estimated by **opm**. The descriptive curve parameters are the lag phase λ, the respiration (or growth) rate μ (corresponding to the steepness of the slope), the maximum cell respiration (or growth) A (corresponding to the maximum value of the curve) and the area under the curve AUC. Note that many respiration curves, even if representing a clearly positive reaction, do not correspond to this idealized scheme. The parameters can nevertheless be robustly estimated from deviating curves, particularly *via* spline fits (Vaas *et al.* 2012, 2013a).

through the fit of sigmoid functions proved for several PM curve shapes to yield biologically unrealistic values (Vaas *et al.* 2012) and have therefore not been implemented.) Three different modelling alternatives for the splines exist (Vaas *et al.* 2013a): (low-rank) cubic smoothing splines (Reinsch 1967) as implemented in `smooth.spline` from the **base** package, thin plate splines (Wood 2003, a generalization of smoothing splines) and P-splines (Eilers and Marx 1996). The latter two are implemented in the package **mgcv**. Their settings have been specifically adapted for the application to PM data. It is also possible to access methods from the package **grofit** (Kahm, Hasenbrink, Lichtenberg-Frate, Ludwig, and Kschischo 2010) or to use a native implementation which is faster but only estimates two of the four parameters. For historical reasons, "grofit" is the default but it is recommended to use the optimized "splines" methods.

The descriptive curve parameters λ, μ, A and AUC estimated by **opm** are shown in Figure 4. In addition to the point estimates for the parameters from both model and spline, confidence limits can be calculated (for the spline-based approach *via* bootstrapping), with 95% being the default value (Efron 1979). But confidence intervals and according group means can also be calculated from experimental repetitions, as explained in Section 2.8. Attaching the aggregated data to an OPM object yields an object of the class OPMA, which can also be stored within an OPMS container object.

2.6. Manipulation of OPM and OPMS data

As usual, data analysis starts with data exploration for which the user may wish to query and subset OPM and OPMS objects (Figure 5). It is easy to select specific wells and time points from OPM or OPMS objects. It also straightforward to select specific OPM objects from an OPMS object that contains them. To this end, OPM and OPMS methods for the generic function `subset()` and R's bracket operator have been implemented. Particularly powerful are the options for metadata-based subsetting. The composition of OPM and OPMS objects, and the implemented methods of the classes, permit queries for the presence of a specific metadata key or a specific value of a specific metadata key, or a specific combination of values and/or keys, and also enable the user to subset OPMS objects accordingly.

But a plethora of methods for querying other aspects of OPM and OPMS objects have also been implemented. Standard operations such as sorting objects and making them unique are also available for the OPM and OPMS classes. Of course, OPMS objects can not only be subset but it it also possible to build up larger OPMS objects by combining OPMS and OPM objects using specialized methods for the `c()` generic function and the `+` operator as well as the very flexible function `opms()`. Moreover, an OPMS method for `merge()` has been implemented, that allows for concatenating PM measurements that represent subsequent runs of the same plate. This has successfully been applied to slow-growing organisms in the bacterial genus *Geodermatophilus*, which had to be measured three times consecutively in the OmniLog® instrument (up to twelve days in total) (Montero-Calasanz, Göker, Pötter, Rohde, Spröer, Schumann, Gorbushina, and Klenk 2012; Montero-Calasanz, Göker, Rohde, Schumann, Pötter, Spröer, Gorbushina, and Klenk 2013).

It is also possible to convert the OPM or OPMS objects to other objects for an independent exploration by the user. This can be done within R, based on a variety of distinct data-frame or matrix objects that can be generated. But export in some useful file formats is also possible.

2.7. Plotting functions for raw data

The function `xy_plot()` displays the raw measurements on the y-axis in dependency on the time on the x-axis. For each well one sub-panel is drawn, and the user is free to colourize the plotted curves by either their affiliation to a specific plate or by a combination of metadata entries of choice. By default the panels are arranged according to the factual microtiter plate dimensions (eight rows labelled A to H × twelve columns labelled 01-12), but other user-defined arrangements are easily feasible because the plates can be subset by selecting specific wells. Every panel is annotated with the microtiter plate numbering (A01 to H12) and additionally or alternatively with the substrate name (given the plate type, the **opm** package can translate all well coordinates to substrate names, see also vignette "Working with substrate information in **opm**"). Thus, the function enables the user to compare the curve data in a customized and useful arrangement (Vaas *et al.* 2012, 2013a).

The function `level_plot()` provides false-colour level plots from the raw respiration measurements over time. Each respiration curve can be displayed as a thin horizontal line, in which the measured respiration value (in OmniLog® units) is represented by colour, while the x-axes indicates the measurement times. With increasing respiration measurement values, the displayed colour changes (by default) from light yellow into dark orange and brownish. The user can obtain an overview in a compacted design (Vaas *et al.* 2012, 2013a). This plot

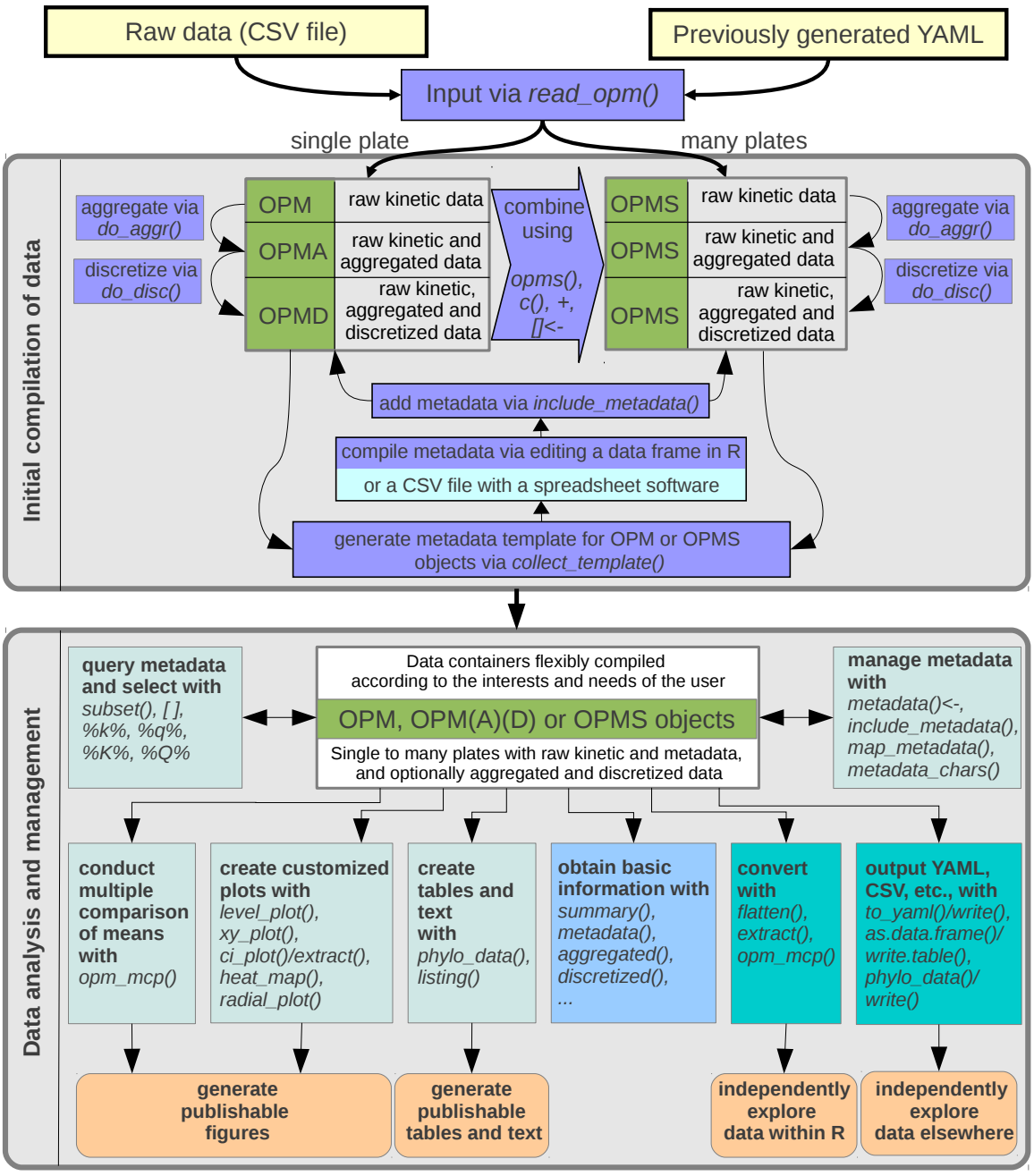

Figure 5: This scheme provides a detailed overview of the possible strategies and appropriate functions for data analysis using the **opm** package. Beginning with one to several CSV files containing raw kinetic data exported by the proprietary OmniLog® software *File Management/Kinetic Analysis*, or YAML or JSON files that have been generated in previous **opm** runs, a variety of work flows are possible for setting up OPM or OPMS objects. Additionally, methods for metadata management, plotting the data in a customized manner, querying and sub-setting the generated objects, statistical comparison of multiple group means, and data-conversion tools including discretization, report generation and output in files are provided.

offers a display format which is especially powerful in uncovering general differences between plates, for example longer lag phases or smaller AUC values across the majority of wells. By default one sub-panel in the level plot corresponds to one complete plate comprising 96 lines, but as in the case of `xy_plot()` plotting could also be preceded by creating subsets of the plates.

2.8. Plotting the aggregated data

For the graphical representation of the aggregated data, namely point estimators and corresponding confidence limits for the curve parameters of selected curves, the function `ci_plot()` is available. The characteristics of different curves assembled into a single overview facilitates the interpretation and comparison of user-defined data subsets arranged according to the technical and/or biological repetition structure or other aspects of the experimental design (Vaas *et al.* 2012).

When analysing empirically obtained measurements such as PM data it is important to consider possible systematic variations and to control for those by normalization. For a PM experiment the purpose of such a normalization is to minimize systematic variations in the aggregated curve parameters so as to more easily recognize biological differences, as well as to allow for the comparison of parameters across plates processed in different experimental runs. The underlying ideas are mainly derived from DNA-microarray experiments for measuring gene-expression levels (Quackenbush 2002).

Using `extract()` the user can select certain aggregated or discretized values into common matrices or data frames. If applied a second time to a previously generated data frame, `extract()` can compute point estimates and their respective confidence intervals for individually defined experimental groups. Optionally, normalization by subtracting, or dividing through, the plate-wise means (across all 96 wells) or well-wise means (across all plates that contain this well) can be conducted beforehand. Although this method is intended mainly as a helper function for `ci_plot()`, it can be quite useful for specific normalization purposes, for example when data were derived before and after servicing the Omnilog® facility, which might result in shifting the measurements by a certain amount. In conjunction with `extract()`, `ci_plot()` allows for visualizing point estimates and confidence intervals of groups of parameter estimates. For visualizing differences between groups and their confidence intervals, see `opm_mcp()` as described in Section 2.9.

Additionally, the package offers the possibility of plotting the aggregated curve parameters as a heat map *via* the function `heat_map()`. Heat maps appear particularly powerful for visualizing the outcomes of PM experiment because dendrograms inferred from both the substrates and the plates can be used to rearrange the plot. Since the user is free to define the metadata to be used for the annotation of the plot and the clustering analysis, this tool provides a powerful feature for data exploration in specialized contexts. For instance, the naming scheme of the individual plates can be devised by selecting associated metadata. It is also possible to automatically construct row groups by selecting the same or other meta-information. `heat_map()` is mainly a wrapper for the `heatmap()` functions from either the **stats** or the **gplots** R package, but contains some useful adaptations to PM data. It facilitates the selection of a clustering algorithm and the construction of row and column groups, and provides more appropriate default solutions for row and column descriptions sizes. (We suppose that in most situations the pictures produced by `heat_map()` should not need to be

manually adapted in these respects.)

Finally, **opm** enables the user to plot aggregated values as radial plots using an eponymous function, which is mainly a wrapper for the `radial.plot()` function from the **plotrix** package adapted to the typical **opm** objects. `radial_plot()` displays a plot of radial lines, polygons or symbols, or a combination of these, centred at the midpoint of the plot frame, the lengths, vertices or positions corresponding to the numeric magnitudes of the data values.

2.9. Statistical comparisons of group means

Besides comparing single curves, the user may also be interested in statistically comparing the mean values of distinct groups of curves. For example, imagine the comparison of four different bacteria using GEN-III microplates. Assume that for each bacterial strain, ten replicates have been performed. (An according example dataset is actually available in the **opmdata** package.) Do these four bacteria differ in, e.g., the mean value of, e.g., curve parameter A of, e.g, well A01? Here, a statistical comparison of four groups (four organisms), each containing ten values (curve parameter A of 10 replicates of well A01), would need to be performed. Statistically, this requires simultaneous inferences across multiple questions (Hothorn, Bretz, and Westfall 2008). To address this issue the function `opm_mcp()` performs simultaneous multiple comparisons of group means by internally calling `glht()` from the **multcomp** package (Hothorn *et al.* 2008) but providing an easier interface for it, specifically adapted to the typical objects used within **opm**. By referring to available metadata and/or the substrate names, the user is able to define groups of interest, set up a model of choice and perform multiple comparison of group means on individually specified contrasts (Bretz *et al.* 2010; Hsu 1996). The choice of appropriate models and contrasts will be explained in detail below. As comparisons of the different curve parameters are performed separately, it is possible to ask very specific questions on differences between curve shapes.

At this point, it is necessary to highlight the power and flexibility of simultaneous multiple comparison procedures and encourage the user to apply contrast tests on individually designed sets of mean comparisons rather than to employ the probably more popular classical ANOVA approaches, which perform F-tests. In general, such F-tests *only* provide global information about main effects and interaction effects. That is, only the significance of a result yields evidence for a difference in the means among any of the considered treatments. For example, in the framework of PM data, a significant F-test on the effect of the substrate would indicate that at least two of the substrates cause distinct respiration. Considering that each PM experiment encounters up to 96 different substrates per plate (overall up to 2,000), this information would obviously be nearly useless. Moreover, F-tests neither provide information about effect sizes nor do they ease to address comparisons of particular interest (Schaarschmidt and Vaas 2009).

We thus opine that the very most underlying questions in PM experiments are best expressed as a set of particular mean comparisons, resulting in a multiple-comparison problem (Hochberg and Tamhane 1987). However, if an increasing number of hypotheses is tested, with the number of true hypotheses unknown, the probability of at least one wrong testing decision also increases. That is, if an increasing number of groups is compared to each other, conclusions on significant differences between a pair of groups are increasingly likely to be wrong. Thus the so-called family-wise error-rate, which is essentially the probability of at least one false rejection among all the null hypotheses, needs to be controlled (Tukey 1994). The here

internally employed functions from the package **multcomp** provides solutions for all listed difficulties, since it allows for testing a user-defined set of contrasts based on a broad range of model types while internally controlling the family-wise error-rate.

Users of multiple-comparison procedures, especially of simultaneous multiple contrast tests as applied here, are encouraged to have a look at the books by (Hochberg and Tamhane 1987) and (Hsu 1996).

Especially in situations where groups are defined by more than one metadata entry the evaluation of differences of treatment means may result in quite complex models. Then, the application of *cell-means models* (also known as *pseudo-one-way layouts*) as discussed in (Schaarschmidt and Vaas 2009)) is strongly encouraged. In this approach estimators for treatment and variance are derived from a model with all treatments combined in a single factor. Technically, this requires the merging of several defining metadata variables into a single one. This can be done by creating new metadata entries from given ones and storing them back in an OPM or OPMS object. An according example is given in Section 3.4. Alternatively, merging can be done when selecting metadata for creating data frames. The computation of multiple comparisons using a *cell-means model* is shown in Section 3.9.

The function `opm_mcp()` internally reshapes the data into a "flat" data frame containing one column for the chosen parameter value, one column for the well (substrate) name and optionally additional columns for the selected metadata. For performing the testing procedure, a model has to be stated that specifies the factor levels that determine the grouping (Searle 1971; Hothorn *et al.* 2008). The `opm_mcp()` function allows for applying such testing directly to OPMS objects, obtaining these factors from stored metadata.

2.10. Discretizing the aggregated data and export for phylogenetic analysis

Whereas the main data-analysis strategies of the **opm** package are based on quantitative, continuous data (as described in the previous chapters), users may nevertheless be interested in discretizing the estimated curve parameters. Discretization transfers continuous data into discrete ones. For example, continuous values ranging from 0 to 400 could be discretized into the three states "low" (from 0 to 100), "intermediate" (from 101 to 200), and "high" (from 201 to 400). Discretizing the data is necessary for analysing them with external programs that cannot deal with continuous characters. Indeed, phylogeny software such as PAUP* (Swofford 2003) and RAxML (Stamatakis, Ludwig, and Meier 2005) is limited to at most 32 distinct character states. (To the best of our knowledge, a maximum-parsimony algorithm applicable directly to continuous data has only been implemented in TNT (Goloboff, Farris, and Nixon 2008).) Phylogenetic studies of PM data, or at least reconstructions of PM character evolution, are of interest because such phenotypic information is frequently used for taxonomic purposes in microorganisms, and here phylogenetic inference methods might be superior to clustering algorithms (Felsenstein 2004). But tabular or textual descriptions of physiological reactions classified into negative, weak (ambiguous) and positive reactions (see Section 2.11 for details) are of even greater relevance in current microbial taxonomy (Tindall, Kämpfer, Euzéby, and Oren 2006).

The **opm** package includes data transformations (implemented in the `discrete()` methods) for coding continuous characters by assigning them to a given number of equal-width categories within a given range. For example, for the parameter A (the maximum curve height)

the theoretically possible range between 0 and 400 OmniLog® units could be used. The data should then be analysed under ordered (Wagner) maximum parsimony in PAUP* (Farris 1970) or with the options for ordered multi-state phenotypic characters in RAxML (Berger and Stamatakis 2010), or corresponding settings in other programs, to minimize the loss of information caused by discretizing the values. For this reason, this kind of unsupervised, equal-width-intervals discretization (Dougherty, Kohavi, and Sahami 1995; Ventura and Martinez 1995), even though simple, appears appropriate for this task. In this context, it also makes not much sense to let a discretization method determine the number of categories because they are not dictated by some property of the data but by the limitations of the subsequently to apply analysis software. The **opm** package offers appropriate functions for data export.

2.11. Determining positive and negative reactions and displaying them as text or table

If users wanted to discretize the parameters into "positive" and "negative" results, this would apparently make most sense for the parameter A because here it is not of interest when and how fast a reaction starts (which would be coded in λ and μ, respectively) or how much overall respiration was achieved (as coded in AUC) but whether or not a reaction takes place at all. Unfortunately, PM data frequently result in a continuum of A values between clearly negative and clearly positive reactions. For instance, the distribution of A in the example datasets distributed with the **opm** and **opmdata** packages is obviously bimodal, but contains a large number of intermediary values. For this reason, the `discrete()` methods and their more user-friendly wrapper `do_disc()` offer a gap-mode discretization by interpreting a given range of values (within the overall range of observations) as "ambiguous". Values below would then be coded as negative, values above the range as positive, and values within the range as either missing information or an intermediary state, "weak". This range could be determined by some discretization approach known from the literature (Dougherty *et al.* 1995; Ventura and Martinez 1995).

The **opm** package offers its automated determination using k-means partitioning as implemented in **Ckmeans.1d.dp** (Wang and Song 2011), using an exact algorithm for one-dimensional data. Alternatively, an algorithm implemented in `best_cutoff()` is available, but it requires measurement replicates (which are highly recommended, if not mandatory, anyway) accordingly annotated in the metadata. Both methods are accessible *via* `do_disc()`. Export as richly annotated, publication-ready HTML table or text is possible using `phylo_data()` and `listing()`. If analysis with phylogenetic programs was of interest, in the case of an intermediary state the data should then be analysed as described above. If intermediary values were coded as missing information they could be analysed under either Wagner or unordered (Fitch) maximum parsimony in PAUP* (Farris 1970; Fitch 1971) or with the options for binary phenotypic characters in RAxML (Berger and Stamatakis 2010), or corresponding settings in other programs.

2.12. Global settings

It is possible to modify settings that have an effect on multiple functions and/or on frequently used arguments globally using `opm_opt()`. This allows the user to adopt **opm** to personal preferences and to thereby substantially decrease coding effort. It is checked that the novel

values inherit from the same class(es) than the old ones. Usage examples are provided in several sections (e.g., Section 3.10.3). The function `param_names()` yields the spelling of the curve parameters used by **opm**. It also displays the set of names that are used by some methods that have to compile metadata entries with other columns. It is thus not impossible, but discouraged, to use these names as metadata keys. The same holds for (non-syntactical) names starting with an underscore and followed by capital letters, as such names are temporarily used by some methods in intermediary objects together with the metadata.

3. Program application

3.1. Overview

Before starting, the **opm** package should be loaded into an R session as follows:

```
R> library(opm)
```

The example dataset distributed with the package (Vaas *et al.* 2012) comprises the results from running 114 GEN-III plates (BIOLOG Inc.) in the PM mode of the OmniLog® reader. The organisms used were two strains of *Escherichia coli* (DSM 18039 = K1 and the type strain DSM 30083^{T}) and two strains of *Pseudomonas aeruginosa* (DSM 1707 and 429SC (Selezska, Kazmierczak, Müsken, Garbe, Schobert, Häussler, Wiehlmann, Rohde, and Sikorski 2012)). The strains with a DSM number could be ordered from the Leibniz Institute DSMZ – German Collection of Microorganisms and Cell Cultures (`http://www.dsmz.de/`).

Each strain was measured in two biological replicates, each comprising ten technical replicates, yielding a total of 80 plates. To additionally investigate the impact of the growth age of cultures on the technical and biological reproducibility of the PM respiration kinetics, strain *E. coli* DSM 18039 was grown on solid LB medium for nine different durations, from 16.75 h (t1) to 40.33 h (t9), respectively. For each growth duration four technical replicates were performed except for t9 (which was repeated only twice), yielding 34 plates for this time-series experiment. All biological and experimental details of this dataset have been described previously (Vaas *et al.* 2012).

Two subsets of the data, `vaas_1` and `vaas_4`, are included in **opm**. Use `?vaas_1` and `?vaas_4` to view their help pages, and have a look at the objects as follows:

```
R> vaas_1
R> vaas_4
```

The entire dataset, stored in the object `vaas_et_al`, comes with the supporting package **opmdata** and can (if that package is installed, of course) be loaded using:

```
R> data(vaas_et_al, package = "opmdata")
```

To view its help page, use `?opmdata::vaas_et_al`.

The metadata included in these objects comprise seven entries. The entry *Experiment* denotes the biological replicate or the affiliation to the time-series experiments. The keys *Species* and

Strain refer to the organism used for the respective experiment (see above), and *Slot* (either *A* or *B*) indicates whether the plate was placed in the left or the right half of the OmniLog® reader. (Note that for an assessment of the reproducibility of the curves the slot is occasionally of relevance.) Two additional entries contain the index of the time point and the corresponding sample point in minutes for the time series experiment. The key *Plate number* indicates the technical replicate (per biological replicate). The combination of the keys *Strains*, *Species*, *Experiment* and *Plate number* results in a unique label which unequivocally annotates every single plate.

3.2. Data import

The following code describes the import of the OmniLog® CSV file(s) into the **opm** package. In the **opm** manual and all help pages, all functions relevant for data import are contained in a family of functions called "IO-functions" with according cross-references.

The CSV files with the OmniLog® raw data should be stored in one to several user-defined folders. Setting the working directory of R to the parent folder of these using `setwd()` frequently facilitates file selection, but in principle the user can provide any number of paths to input files and/or directories containing such files to the function `read_opm()`, which can load several CSV files (and also YAML or JSON files generated by **opm**) at once. A restriction of the input functions is that they can solely read CSV files that only contain the measurements from a single plate per file (either a PM plate or a single GEN-III plate measured in either PM- or identification modus). But the package contains a function `split_files()`, which can be used to split CSV files with multiple plates into one file per plate.

To illustrate the file import step by step, a set of input CSV example files is provided with the package. Before starting, remember that the **opm** package must be loaded. Then use the built-in function `opm_files()` to find the example files in your R installation:

```
R> files <- opm_files("testdata")
R> files
```

Afterwards check whether this returned a vector of nine file names, including the full path to their location in the file system. (It might fail in very unusual R installation situations; in that case, the files must be found manually.) For demonstration purposes, the test data contain data from EcoPlate® Gen-III, PM01 and PM20 plate types. One of these files contains multiple plates and acts as an example for `split_files()`; the other ones can be read directly.

As a demonstration of file splitting, consider the following code, which creates single-plate CSV files from the multi-plate file that comes with **opm**:

```
R> multi.plate.file <- grep("Multiple", files,
     value = TRUE, ignore.case = TRUE)
R> multi.plate.file
R> list.files()
R> split_files(multi.plate.file, '^("Data File",|Data File)', getwd())
R> list.files()
```

Three additional files should be visible in the working directory, consecutively numbered but using the base name of the input file. These could now be read with `read_opm()`, but we will use other files in the next paragraph and tidy up now:

```
R> rm(multi.plate.file)
R> unlink(grep("Multiple-0000", list.files(),
     value = TRUE, ignore.case = TRUE))
```

Using `read_opm()`, from a given vector of file and/or directory names, files can be easily selected and deselected using globbing or regular-expression patterns. For instance, for reading the three example files in "new style" CSV format (see Section 2.2), use the following code.

```
R> example.opm <- read_opm(files, include = "*Example_?.csv.xz")
R> summary(example.opm)
```

After performing this step, the OPMS object contains three plates, as indicated by the `summary()` function.

Instead of a single file name the user could also provide several file names to `read_opm()`, or a mixture of file and directory names. If these were contained as subdirectories of the current working directory, `read_opm(".")` or `read_opm(getwd())` would be sufficient to input these files. To filter the files with patterns, the arguments `exclude` and `include` are available. There is also a *demo* mode allowing the user to check the effect of each argument before actually reading files. One can use the `gen.iii` argument to trigger the automated conversion of the plate type to, e.g., GEN-III or "ECO" plates run in "PM" mode, or convert later on using the `gen_iii()` function itself. Plate-type conversions to one of the "PM" modes are disallowed (and are, to the best of our knowledge, not relevant in practice anyway). The plate type is crucial, as it is disallowed to integrate distinct plate types into a single OPMS objects. The reason is that comparing the same well positions from distinct plate types would be almost always equivalent to comparing apples and oranges.

If more than one plate of the same plate type is read, however, data from all files are automatically integrated into a single OPMS object. To read plates from several types at once, the `convert` argument is useful. If one uses `read_opm(..., convert = "grp")`, a named list is created with, as each list element, one OPM or OPMS object per plate type, depending on whether only a single plate of that plate type, or several such plates, have been found. For instance, for inputting all example files (except for the one with multiple plates), consider the following code:

```
R> many.plates <- read_opm(files, exclude = "*Multiple*", convert = "grp")
R> summary(many.plates)
R> summary(many.plates$PM01)
R> rm(many.plates) # tidy up
```

This yields the data from plates with distinct plate types in a single object. Note that the objects for each encountered plate type can easily be accessed *via* the names of the list. More example code is available *via* `opm_files("examples")`.

A single plate could also be imported using `read_single_opm`. But this might only occasionally be useful, as `read_opm` can cope with single files, too.

3.3. Batch conversion of many files

In addition to `read_opm()` and `read_single_opm()` (Section 3.2), which need to be called before an interactive exploration of PM data, batch-processing large numbers of files by converting them from CSV (or previously generated YAML or JSON) to YAML, JSON or CSV format, is also possible. This optionally includes aggregating the raw data by estimating curve parameters (Section 3.5), discretizing these parameters (Section 3.10.2) and integrating metadata (Section 3.4). Again there is a *demo* mode to first investigate the attempted conversions:

```
R> batch_opm(files, include = "*Example_?.csv.xz",
     aggr.args = list(boot = 100, method = "opm-fast"),
     outdir = ".", demo = TRUE)
```

The arguments `aggr.args`, `disc.args` and `md.args` control aggregation, discretization and metadata incorporation, respectively. Details on all three processes are given in the according sections, and for the exact use of these arguments see the **opm** manual, entering `?batch_opm`.

The following command would read three of the seven example input files, estimate two of the four curve parameters using the fast native method including 100 rounds of bootstrapping, and store the resulting YAML files (one per plate) in the current working directory (given by "."):

```
R> batch.result <- batch_opm(files, include = "*Example_?.csv.xz",
     aggr.args = list(boot = 100, method = "opm-fast"),
     outdir = ".")
```

By default, progress messages are printed to the screen. The return value, here assigned to the `batch.result` variable, also contains all information about the success of the individual file conversions.

The "run_opm.R" script distributed with the package is an Rscript-dependent command-line tool for non-interactively running such file conversions. Its location in the file system can be obtained using

```
R> opm_files("scripts")
```

Regarding its use, see the documentation of Rscript for details (enter `?Rscript` at the R prompt) and watch the help output of this script (try `system(opm_files("scripts"))`).

3.4. Integration and manipulation of metadata

Several ways for linking metadata to OPM or OPMS objects are possible. The easiest one is probably the batch-inclusion after creating a template with plate identifiers associating it with metadata. In the first step, either a data frame to be manipulated within R or a CSV file to be modified with a suitable editor are created. The **opm** package supports metadata integration by creating a template for such a table from an OPM or OPMS objects that contains plate identifiers in the first columns; by default the keys *Setup Time*, *Position* and *File*. These data must not be changed, ensuring that the package can later on link the metadata to the dedicated plates according to these identifiers.

In the **opm** manual and help pages, most functions relevant for metadata manipulation are contained in a family called "metadata-functions" with according cross-references. For the collection of a metadata template in a data frame to be manipulated in R, use this command:

```
R> metadata.example <- collect_template(files, include = "*Example_?.csv.xz")
```

For the generation of a metadata template file, the following command can be used:

```
R> collect_template(files, include = "*Example_?.csv.xz",
     outfile = "example_metadata.csv")
```

This will result in a file "example_metadata.csv" in the current working directory (whose name is accessible using `getwd()`). If other metadata have previously been collected, by default a pre-existing file with the same name will be reused. The pre-defined columns will be respected, novel rows be added, old metadata will be kept and identifiers for novel files will be included and their so far empty metadata columns are set to missing data (NA). You can also provide the location of another previously created metadata file with the `collect_template()` argument `previous`.

The generated CSV file could then be edited using external software; for the purpose of this tutorial, we load it directly and manipulate it in R. To avoid the usual changes in data format and header of the table during the import a customized import function was implemented as a wrapper for `read.delim()`:

```
R> metadata.example <- to_metadata("example_metadata.csv")
```

Per default, this expects CSV columns separated by tabulators, with the fields protected by quotes. To input other formats, consider the `sep` argument for defining an alternative column separator, as well as the `strip.white` argument for turning the removal of whitespace at the beginning and end of the fields on or off (which is relevant if a spreadsheet program exports CSV *without* quotes).

Now the user could add information to the data frame by calling `edit()`, which would open the R editor, or by any other way of manipulating data frames in R. New columns could be defined, or the existing metadata modified. But the first columns must remain unchanged because they are needed to identify individual PM plates for linking them to their meta-information. As an example, we here add an (arbitrary) *Colour* column with the values "blue", "red" and "yellow" and another (arbitrary) *Integer* column with the integer values 10, 20 and 30:

```
R> metadata.example$Colour <- c("blue", "red", "yellow")
R> metadata.example$Integer <- c(10L, 20L, 30L)
```

Now the metadata are ready to be included into the previously generated OPMS object:

```
R> example.opm <- include_metadata(example.opm, md = metadata.example)
```

The metadata could then be received as follows:

```
R> metadata(example.opm)
```

This returns the entire metadata entries as a list. By default only the added metadata are included in the object, but not the identifiers used for assigning data-frame rows to plates.

One might want to remove the file as it is not needed any more:

```
R> unlink("example_metadata.csv")
```

A couple of other functions have been implemented for manipulating metadata included in OPM and OPMS objects. For instance, the entire meta-information, or specific entries, can be set using the replacement function `metadata()<-`. Setting a specific entry named `key` to a specific value `value` in all plates would be accomplished by `metadata(example.opm, key) <- value`. If the right side of the assignment is a data frame with the same length as the OPMS object, each row would be specifically assigned to the OPM object with the same index. This comes handy for adding the `csv_data()` selected or all information from the OmniLog® CSV files to the metadata:

```
R> metadata(example.opm)
R> metadata(example.opm) <- to_metadata(csv_data(example.opm))[,
     c("Strain Name", "Sample Number")]
R> metadata(example.opm)
```

You might note that "Sample Number" is a misnomer in these datasets. (One of the fields in the interface of the *Data Collection* software of the OmniLog® reader had been defined as "Sample Number", but the operator entered species and strain designations into this field.) In this and similar cases, modifying metadata in-place is of interest, which can be accomplished using `map_metadata()`. This function would return a novel OPMS (or OPM) object. Its formula method is particularly powerful.

```
R> metadata(example.opm)
R> metadata(map_metadata(example.opm, Organism ~ `Sample Number`))
```

This works by converting the left side of the formula into a metadata key and evaluating the right side of the formula in the context of the metadata entries that have already been added. As result, a new metadata entry is created, with "Organism" as key and the entry from "Sample Number" as value. "Sample Number" must be quoted because it contains a special character (the blank).

But we have not yet removed the aptly named "Sample Number" entries. Here, it is useful that all operators (except for `$` and other high-precedence operators, which can be used for defining nested keys) on the left side, if present, would be changed by `map_metadata()` into a call to `list()`. The resulting list would be flattened and treated as a list of metadata keys. Hence it is possible to define several keys at once. The right side, once evaluated, would be recycled accordingly. Thus we can clean up our metadata in a single line of code:

```
R> metadata(map_metadata(example.opm,
     Organism + `Sample Number` ~ list(`Sample Number`, NULL)))
```

The deletion of "Sample Number" is accomplished by the assignment of `NULL`, as usual in lists. Instead of `+` almost all other operators could be used, and one could also write `c(Organism, 'Sample Number')` on the left side, which might be more intuitive. If `map_metadata()` is called without a mapping, it "cleans" the metadata by removing empty entries (by default including those that only contain `NA` values) and converting factors to character vectors.

But we have not yet stored an OPMS object with the cleaned metadata. This could be done using `example.opm <- map_metadata(example.opm, ...)`. In that case, however, direct assignment would also be possible:

```
R> metadata(example.opm) <- Organism + `Sample Number` ~
     list(`Sample Number`, NULL)
R> metadata(example.opm)
```

Assigning `NULL` to a metadata entry would remove that entry. We can achieve the same using an expression object:

```
R> metadata(example.opm) <- to_metadata(csv_data(example.opm))[,
     c("Strain Name", "Sample Number")] # reset
R> metadata(example.opm)
R> metadata(example.opm) <- expression(Organism <- `Sample Number`,
     rm(`Sample Number`))
R> metadata(example.opm)
```

Here, the assignment targets (names within the metadata) are specified directly using just the `<-` operator. Apparently, arbitrarily complex code could be put in such a metadata-modifying expression.

All metadata would be cleared by assigning an empty list, without specifying a key:

```
R> metadata(example.opm, "Organism") <- NULL
R> metadata(example.opm)
R> metadata(example.opm) <- list()
R> metadata(example.opm)
```

So keep in mind that formulas and expressions are very flexible for modifying metadata entries. They would allow for any other operation (such as numerical calculations) provided it can be applied to the selected predefined metadata content. The replacement function can also be used to copy metadata between OPM and/or OPMS objects.

Metadata can also be assigned specifically for subsets of OPMS objects, using the indexed assignment available for those objects:

```
R> metadata(example.opm[2]) <- list(Organism = "Elephas maximus",
     Size = "3 meters")
R> metadata(example.opm)
R> metadata(example.opm[2]) <- list()
R> metadata(example.opm)
```

Readers may have noted that `metadata()` always returns a list, not a data frame. This is because metadata need not contain the same entries, even within a single OPMS object, and can be nested. It is possible, however, to get the metadata as data frame by using `to_metadata()`. Missing entries would then be filled with `NA` values, and nested metadata entries would yield data-frame columns of the mode "list". This might or might not be suitable for further processing. For statistical analysis, the appropriate way is to extract only those metadata entries that are present in *all* OPMS elements, and usually also only those that are not themselves lists. Methods such as `extract()` are based on this principle.

The following code, making use of the `metadata.example` data frame generated above, adds a new metadata entry with the key "Character" containing the integer values from the metadata entry called "Integer" converted to character mode It then includes a new metadata entry with the key "Times 10" containing the entry "Integer" multiplied by 10.

```
R> example.opm <- include_metadata(example.opm, md = metadata.example)
R> metadata(example.opm)
R> example.opm <- map_metadata(example.opm, Character ~ as.character(Integer))
R> metadata(example.opm)
R> example.opm <- map_metadata(example.opm, `Times 10` ~ (Integer * 10))
R> metadata(example.opm)
```

Note that `map_metadata()` can also be used with character vectors as mapping objects. Making use again of the exemplar generated above, the key *Colour* could be changed to *Colony colour* as follows:

```
R> example.opm <- include_metadata(example.opm, md = metadata.example)
R> md.map <- metadata_chars(example.opm, values = FALSE)
R> md.map
```

This yields a character vector including itself as `names` attribute, thus implying an identity mapping. Next the new labels will be defined and will then be exchanged with the old ones using `map_metadata()`.

```
R> md.map["Colour"] <- "Colony colour"
R> example.opm <- map_metadata(example.opm, md.map, values = FALSE)
R> metadata(example.opm)
```

The keys should have been changed to *Colony colour* now but the values should have remained unaffected. In addition to mapping based on character vectors, a mapping function could also have been used. By setting their argument `values` to `TRUE`, the functions `metadata_chars()` and `map_metadata()` could be used as well to modify values instead of key. For instance, assume any entries "red" in the field denoted *Colony colour* should be changed to "green":

```
R> md.map <- metadata_chars(example.opm, values = TRUE)
R> md.map
R> md.map["red"] <- "green"
R> example.opm <- map_metadata(example.opm, md.map, values = TRUE)
R> metadata(example.opm)
```

This command will transform all entries in the table with the value "red" to "green". Other values, as well as the keys, should be unaffected.

Frequently metadata entries will be used as factors in statistical models. This always requires that the chosen metadata entry is present in all considered OPM object and sometimes requires that entries have to be combined. For instance, for setting up a *cell-means model* (see Section 2.9 and Section 3.9 for further details), factors used for defining the groups of interest have to be merged. This might already be done during the initial step when setting up the metadata data frame *before* including the metadata into an OPM or OPMS object using `include_metadata()`. Here, the function `interaction()` could be used to concatenate columns (but it should be taken into account that metadata entries should better not be represented as factors). As a result, two metadata entries would be merged into a single one:

```
R> metadata.example$Colour.Position <- as.character(interaction(
     metadata.example$Colour,
     metadata.example$Position, sep = ".", drop = TRUE))
```

This is not recommended, however, unless all statistical comparisons of interest, or at least the group definitions of interest, were already known at this stage. Even more tedious would be to go back to the initial metadata compilation add a later stage. Using the metadata mapping functions, metadata entries can instead by merged at any time *after* including them into an OPM or OPMS with `include_metadata()`. For instance, the following code operates directly in the OPMS object, merging the . "Colony colour" (which had previously been renamed from "Colour", see above) and "Integer" entries into a new one:

```
R> metadata(example.opm) <- Col.Int ~ paste(`Colony colour`, Integer, sep = ".")
R> metadata(example.opm)
```

As result, a new metadata entry named "Col.Int" is created with the general string concatenation tool `paste()`.

3.5. Aggregating data by estimating curve parameters

As mentioned above, the package brings along an OPM object, named `vaas_1`, containing a full 96-well plate, aggregated data (curve parameters), and metadata:

```
R> data(vaas_1)
R> vaas_1
```

In the **opm** manual and help pages, the functions relevant for data aggregation are contained in a family called "aggregation-functions" with according cross-references. Primarily `do_aggr()` should be used for aggregation because it generates the kinds of objects that allow for the predefined work flows. `vaas_1` already contains aggregated data but we will re-calculate some for demonstration purposes. For invoking the fast estimation method, use:

```
R> vaas_1.reaggr <- do_aggr(vaas_1, boot = 100, method = "opm-fast")
```

This will only estimate two of the four parameters, namely A and AUC. (Screen messages output by `boot.ci()` might be annoying but can usually be ignored.) Information about the data aggregation settings is available *via* `aggr_settings()`:

```
R> aggr_settings(vaas_1)
R> aggr_settings(vaas_1.reaggr)
```

and the aggregated data can be extracted as a matrix *via* `aggregated()`, e.g.:

```
R> summary(aggregated(vaas_1))
R> summary(aggregated(vaas_1.reaggr))
```

The default settings of `do_aggr()` includes 100-fold bootstrapping of the data to obtain confidence intervals. As this is a time-consuming intensive process (particularly if **grofit** is used), it may be split over several cores on a multi-core machine if `mclapply()` from the **parallel** R package can be run with more than one core, which is possible on all operating systems except for Windows.

One can also specify different spline fitting methods using `method = "spline"`, which is the recommended setting (for historical reasons, it is not the default) (Vaas *et al.* 2013a). Options such as the spline type, the number of knots used for the spline and other options for the splines can be easily set using the function `set_spline_options` (which can only be used in conjunction with `method = "spline"`). To essentially reproduce the results from `method = "grofit"` we could use smoothing splines (and for the sake of computing time in this vignette we only use 10 bootstrap replicates to compute confidence intervals for the parameter estimates):

```
R> op <- set_spline_options(type = "smooth.spline")
R> vaas_1.aggr2 <- do_aggr(vaas_1, boot = 10, method = "spline", options = op)
```

Other spline types can be specified *via* the `type` argument in the function `set_spline_options`. But the defaults have been optimized for PM data.

3.6. Manipulation of OPM and OPMS data

In the **opm** manual and help pages, the functions relevant for retrieving information contained in OPM or OPMS objects are included in a family called "getter-functions" with according cross-references.

For instance, the user may wish to select specific wells from the input plates, which are present in a 96-well layout, numbered from A01 to H12. The function `dim()` provides the dimensions of an OPMS object as a three-element vector comprising (i) number of contained OPM or OPMA objects, (ii) the number of time points (of the first contained plate; these values need not be uniform within an OPMS object), and (iii) the number of wells (which must be uniform within an OPMS object).

To extract, for example, only the data from wells G11 and H11 together with the negative-control well A01 from the dataset `vaas_et_al` the bracket operator defined for the OPMS class has to be invoked as follows:

```
R> data("vaas_et_al", package = "opmdata")
R> vaas.small <- vaas_et_al[, , c("A01", "G11", "H11")]
R> dim(vaas.small)
```

R users should be familiar with this subsetting style, which was modelled after the style for multidimensional arrays, even though the internal representation is quite different. Following the `dim()` function, in the first indexing position the plates, in the second the time points, and in the third position the wells are selected. Moreover, in the second indexing position lists could be used, and in the third indexing position a formula could be applied. This allows for creating sequences of well coordinates as, e.g., in `vaas_et_al[, , ~c(A08:B02, B05)]`, which would select eight wells.

After metadata have been added and adapted (see Section 3.4), OPM and OPMS objects can be queried for their content. Specialized infix operators `%k%` and `%q%` (for `%K%` and `%Q%` see `¿%K%‘` and `¿%Q%‘`, respectively) have been modelled in analogy to R's `%in%` operator. The user may be interested whether an OPM or OPMS object contains a specific value associated with a specific metadata key, or the key associated with any value, or combinations of keys and/or values. `%k%` allows the user to search in the metadata keys. The user can test whether

all given keys are present as names of the metadata. `%q%` tests whether all given query keys are present as names of the metadata and refer to the same query elements.

Some examples using `vaas_et_al` are given in the following. This OPMS object contains a metadata key *Experiment* with the three possible values *Time series*, *First replicate*, and *Second replicate*, and a metadata key *Species* with either *Escherichia coli* or *Pseudomonas aeruginosa* as values.

Examples for questions that could be asked on these metadata are given in the following. Which plates within `vaas_et_al` have *Experiment* as metadata key?

```
R> "Experiment" %k% vaas_et_al
R> vaas_et_al %k% "Experiment" # equivalent
R> vaas_et_al %k% ~ Experiment # equivalent
R> (~ Experiment) %k% vaas_et_al # equivalent, parentheses needed
```

Note that the arguments can be swapped and that a formula can be used. Next, which plates within `vaas_et_al` have *Experiment* and *Species* as metadata key?

```
R> c("Experiment", "Species") %k% vaas_et_al
R> vaas_et_al %k% ~ c(Experiment, Species) # equivalent
```

The formula method works by evaluating the right side of the formula in the context of the metadata entries and reporting whether or not this yielded an error. For this reason, `vaas_et_al %k% ~ Experiment + Species` would fail because there is no `+` operator for character strings.

Which plates within `vaas_et_al` have *Experiment* and *Species* as metadata key with the respective values *First replicate* and *Escherichia coli*?

```
R> c(Experiment = "First replicate",
     Species = "Escherichia coli") %q% vaas_et_al
R> vaas_et_al %q% ~ Experiment == "First replicate" &
     Species == "Escherichia coli"
```

Again the two solutions are equivalent, but note the differences in the syntax that has to be used. The formula method allows, in principle, for arbitrarily complex expressions.

Which plates within `vaas_et_al` have *Species* as metadata key associated with the value *Escherichia coli* or the value *Bacillus subtilis*?

```
R> list(Species = c("Escherichia coli", "Bacillus subtilis")) %q% vaas_et_al
R> vaas_et_al %q% ~ Species %in% c("Escherichia coli", "Bacillus subtilis")
```

In addition to conducting queries with alternatives, using lists as queries would also allow for nested queries (as the metadata entries could also be nested). Within formulas, nested keys should be separated by the `$` operator.

The results of these infix operators are reported as logical vector with one value per plate; the usual R functions such as `all()`, `any()` or `which()` could be applied to work on these vectors. They could also be used directly as the first argument of the bracket operator for OPMS objects to create subsets:

```
R> vaas.e.coli.1 <- vaas_et_al[c(Experiment = "First replicate",
     Species = "Escherichia coli") %q% vaas_et_al]
R> summary(vaas.e.coli.1)
R> rm(vaas.e.coli.1) # tidy up
```

Alternatively, the user may wish to subset a certain part of the data set using the function `subset()`, which is based on these kinds of querying for metadata keys and their values. Prior to this, the user could check the keys of the metadata:

```
R> metadata_chars(vaas_et_al, values = FALSE)
```

The values in the metadata could be obtained by using `values = TRUE`. Additionally, the user can check the values of special keys in the metadata:

```
R> metadata(vaas_et_al, "Species")
```

The resulting vectors could then also be used for mapping old metadata keys or values to novel ones (for details see Section 3.4).

The presented plotting results of `xy_plot()` and `level_plot()` (see Section 3.7) show selected subsets of `vaas_et_al`. In our example below, the function `subset()` extracts the plates which contain the value *First replicate* in the metadata key *Experiment* and the value *6* in the key *Plate number*, resulting in one representative technical repetition and thus four plates (because four strains were involved) from the data set `vaas_et_al`:

```
R> vaas.1.6 <- subset(vaas_et_al,
     query = list(Experiment = "First replicate", 'Plate number' = 6))
R> summary(vaas.1.6)
```

Providing the desired combination of metadata keys and values as a list offers much flexibility, and using a formula would offer a maximum of flexibility, but other approaches are also implemented. The selection of plates could be based on the presence of keys only (like `%k%` described above; it makes not much sense for `vaas_et_al` whose plates are uniform regarding the keys), and/or use nested queries (like `%q%` with a list described above; makes of course more sense if the metadata contain nested entries).

The `subset()` function also has a "time" argument that allows one to create a subset containing only the time points that were common to all plates. This is useful because deviations regarding the overall measurement hours might exist. See the manual for details, using `help('[', package = "opm")`.

In addition to plate-wise querying and subsetting of OPMS objects, a number of conversion functions for selected content of all plates have been implemented. The **opm** manual and help pages list them in a family of functions called "conversion-functions" with according cross-references.

For instance, the user may wish to explore the aggregated curve parameters (lag phase λ, steepness of the slope μ, maximum curve height A, and area under the curve AUC). These may be exported either as matrix or data frame using `extract()`:

```
R> vaas.mu <- extract(vaas_et_al, dataframe = TRUE,
     as.labels = NULL, subset = "mu")
```

To extract also the full or partial set of metadata, it is sufficient to add a list of desired metadata:

```
R> vaas.mu <- extract(vaas_et_al, dataframe = TRUE,
     as.labels = list("Experiment","Number of sample time point",
       "Plate number", "Slot", "Species", "Strain", "Time point in min"),
     subset = "mu")
```

This only works if this meta-information is present for the plates under study. Once a data frame is exported, these metadata will be contained in additional columns; once a matrix is exported, they will be used to construct the row names. The metadata could also be selected using a formula; see the help pages for details, particularly `?metadata`. A peculiarity of `extract()` is that formulas can be used to trigger the joining of selected metadata entries (converted to data-frame columns) into new ones, using the pseudo-function `J()` within the formula. For instance, the following code would create a new entry called "Species.Strain":

```
R> vaas.mu <- extract(vaas_et_al, dataframe = TRUE,
     as.labels = ~ J(Species, Strain), subset = "mu")
```

This is applied by `opm_mcp()`, see Section 3.9. The behaviour during joining of factors is modified using `opm_opt(comb.key.join = ...)` and `opm_opt(comb.value.join = ...)`. The default curve parameter returned by `extract()` can be set with `opm_opt(curve.param = ...)`.

Finally, note that methods for a variety of generic R functions such as `unique`, `sort`, `duplicated`, `anyDuplicated` and `merge` are available for OPMS objects. As specified using `sort(by = ...)`, sorting can be done based on selected metadata or on `csv_data()` entries such as the setup time. The latter is of use in conjunction with the `merge` method, which is able to concatenate OPM objects from subsequent runs of the same plate. Enter `help(sort, package = "opm")` and `help(merge, package = "opm")` at the R prompt for details.

3.7. Plotting functions for raw data

In the **opm** manual and help pages, the functions relevant for plotting are contained in a family called "plotting-functions" with according cross-references. The function `xy_plot()` displays the respiration curves as such (see Figure 6). In our example the selected OPMS object `vaas.1.6` is the subset of the dataset `vaas_et_al` constructed in Section 3.6:

```
R> xy_plot(vaas.1.6, main = "E. coli vs. P. aeruginosa",
     include = list("Species", "Strain"))
```

Using the argument `main`, the user can include a main title in the plot; if it is omitted, by default the title is automatically constructed from the plate type. Likewise, the well coordinates are automatically converted to substrate names (details can be set using additional arguments). The content of the legend (mainly a description of the assignment of the colours to the curves) is also determined automatically.

The argument `include` refers to the metadata and allows the user to choose which entries should be used for assigning curve colours and accordingly be included in the legend. Character vectors, lists and formulas are allowed as `include` argument. See Section 3.4 and

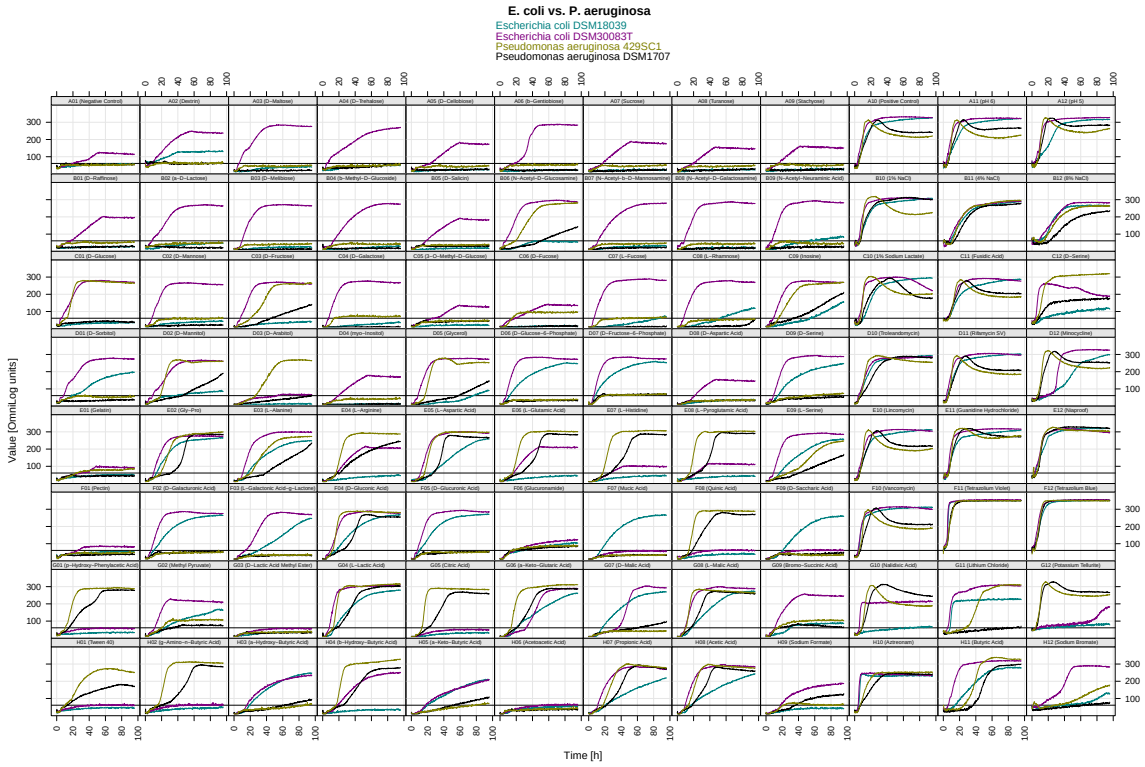

Figure 6: PM curves from the 6th technical repetition of the first biological repetition plotted using `xy_plot()` and by default arranged according to the factual plate layout (see (Vaas *et al.* 2012) for the difference between technical and biological repetitions). The respective curves from all four strains are superimposed; the affiliation to each strain is indicated by colour (see the legend). The x-axes show the measurement time in hours, the y-axes the measured colour intensities in OmniLog® units.

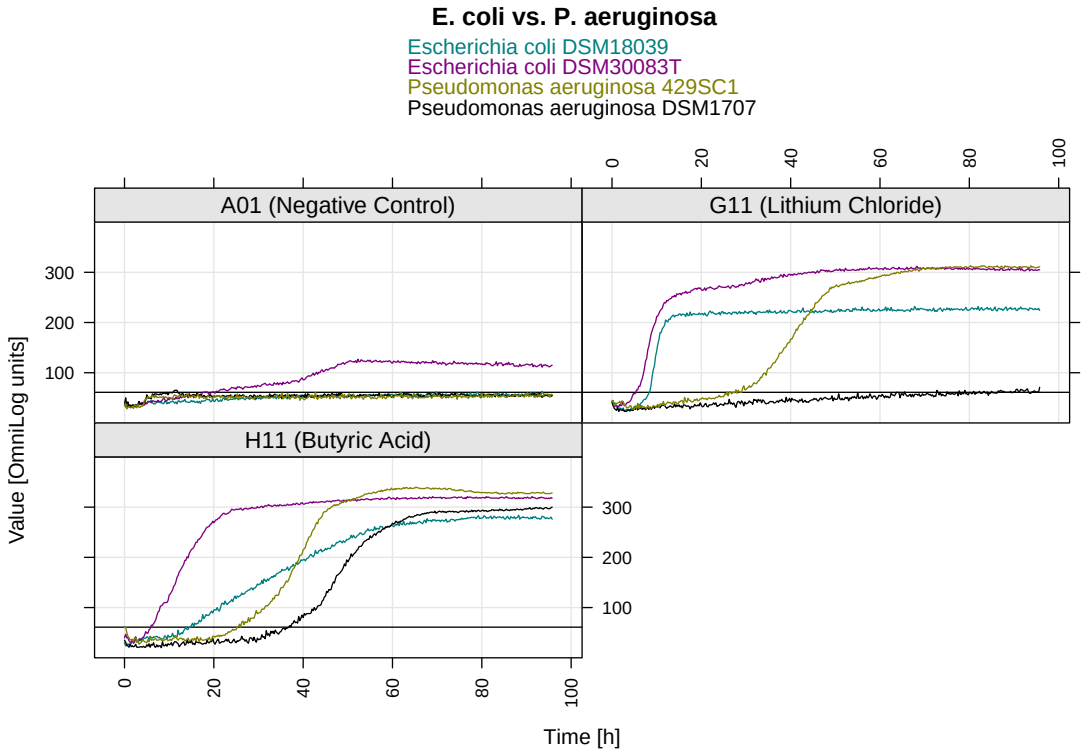

Figure 7: Selected PM curves from the 6th technical repetition from the first biological repetition plotted using `xy_plot()`. The respective curves from all four strains are superimposed, the affiliation to each strain indicated by colour (see the legend). The x-axes show the measurement time in hours, the y-axes the measured colour-value units.

`?metadata` in the help pages for details. In the example the combination of species and strain is used, yielding four distinct colours. If `include` is not used, the colours are assigned per plate. Several predefined colour palettes are available in **opm** (accessible *via* `select_colors()`) with a maximum of 24 distinct colours. If more colours were needed, the user should set up a larger own colour vector and pass it as the argument `col` to `xy_plot` or preferably use `opm_opt(colors = ...)`.

The plotting of sub-panels (see Figure 7) works as described above; the only difference is the previous manipulation of the dataset:

```
R> xy_plot(vaas.1.6[, , c("A01", "G11", "H11")],
     main = "E. coli vs. P. aeruginosa", include = list("Species", "Strain"))
```

The function `level_plot()` (see Figure 8) provides false-colour level plots from the raw respiration measurements over time:

```
R> level_plot(vaas.1.6, main = "E. coli vs. P. aeruginosa",
     include = list("Species", "Strain"))
```

Again, a main title can be set explicitly. Furthermore, the argument `include` again refers to the metadata and allows the user to choose the information to be included in the header

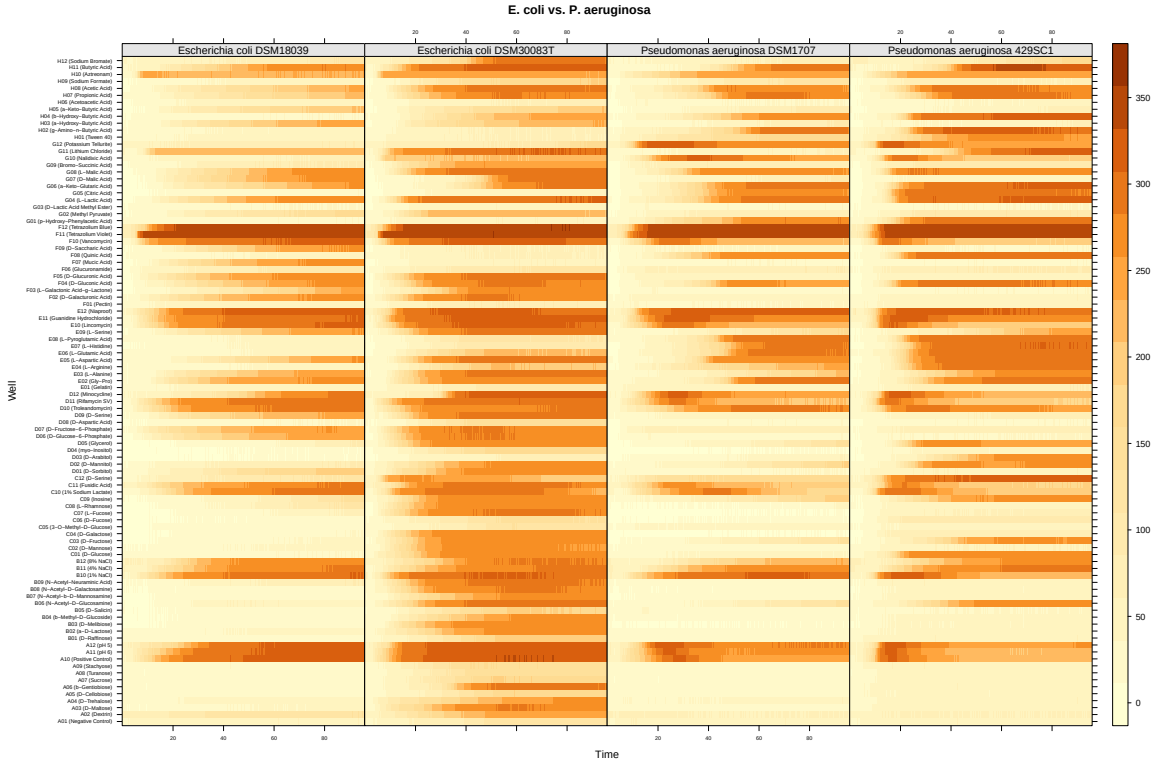

Figure 8: Visualization of PM curves using the function `level_plot()`. Each respiration curve is displayed as a thin horizontal line, in which the curve height as measured in colour-value units is represented by color intensity (darker parts indicate higher curves). The x-axes correspond to the measurement time in hours.

for annotating the plates. In the example the combination of species and strain is used. The default colour palette used can by modified with `opm_opt(color.borders = ...)`.

3.8. Plotting the aggregated data

The function `heat_map()` (see Figure 9) provides false-colour level plots in which both axes are rearranged according to clustering results. In the context of PM data, it makes most sense to apply it to the estimated curve parameters. This **opm** function is a wrapper for `heatmap()` from the **stats** and `heatmap.2()` from the **gplots** package with some adaptations to PM data. For instance, row groups can be automatically constructed from the metadata.

The function `heat_map()` could be applied to matrices or data frames constructed using the helper function `extract()`, but it is more convenient to apply it directly to OPMS objects:

```
R> vaas.1.6.A <- heat_map(vaas.1.6, as.labels = "Strain",
     as.groups = "Species")
```

The function `radial_plot()` is able to plot numeric values as distances from the center of a circular field in directions defined by angles in radians. Some selection of wells should usually be applied beforehand for these plots to be useful. Figure 10 provides an example of such a visualization. The parameter maximum height (A) is plotted for the wells A01 to A05 and A10 from dataset `vaas_4`. Note also that the values for positioning the upper-left corner of the legend are oriented according to the axes of the plot. Hence, if the legend should be placed in the lower left part of the figure, negative values for x and y would be necessary. The code is as follows:

```
R> radial_plot(vaas_4[, , c(1:5, 10)], as.labels = list("Species", "Strain"),
     x = 150, y = 200)
```

The function `ci_plot()` is able to visualize point estimates and corresponding 95% confidence intervals for the parameters, derived *via* bootstrapping during aggregation of raw kinetic data into curve parameters, or, in conjunction with `extract()`, from plate groups defined by the metadata. Thereby the bracket operator as described above (see Section 3.6) facilitates the selection of subsets of interest.

Figure 11 provides an example of such a visualization. The parameter maximum height (A) is plotted for the three wells A01 (Negative Control), A02 (Dextrin) and A03 (D-Maltose) from one plate (the sixth plate of the first biological repetition from dataset `vaas_et_al`). The code is as follows:

```
R> ci_plot.legend <- ci_plot(vaas.1.6[, , c("A01", "A02", "A03")],
     as.labels = list("Species", "Strain"), subset = "A",
     legend.field = NULL, x = 170, y = 3)
```

Furthermore, the helper function `extract()` (more specifically, its data-frame method) can group curve parameters from OPMS objects according to selected metadata and calculate according point estimates (means) and confidence intervals. It can also apply normalisation beforehand, which might frequently be necessary to more easily recognize biological differences; see Section 2.8.

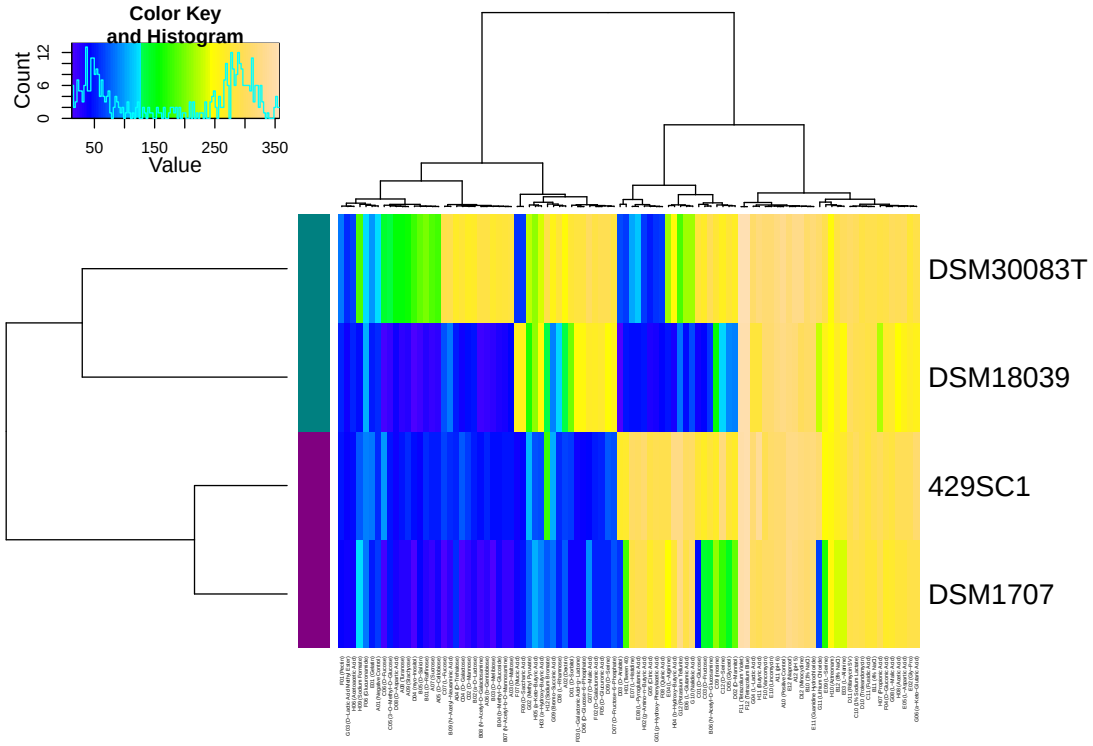

Figure 9: Visualization of the clustered results from the curve parameter maximum height (A) for each substrate using the function `heat_map()`. The x-axis corresponds to the substrates clustered according to the similarity of their values over all plates; the y-axis corresponds to the plates clustered to the similarity of their values over all substrates. As row labels, the strain names were selected (argument `as.labels`), whereas the species affiliations was used to assign row group colours (bars at the left side, argument `as.groups`). The central rectangle is a substrate × plate matrix in which the colours represent the classes of values. The default colour setting uses topological colours, with deep violet and blue indicating the lowest values and light brown indicating the highest values, but another colour palette could also be chosen by the user. The default can by set with `opm_opt(hm.colors = ...)`.

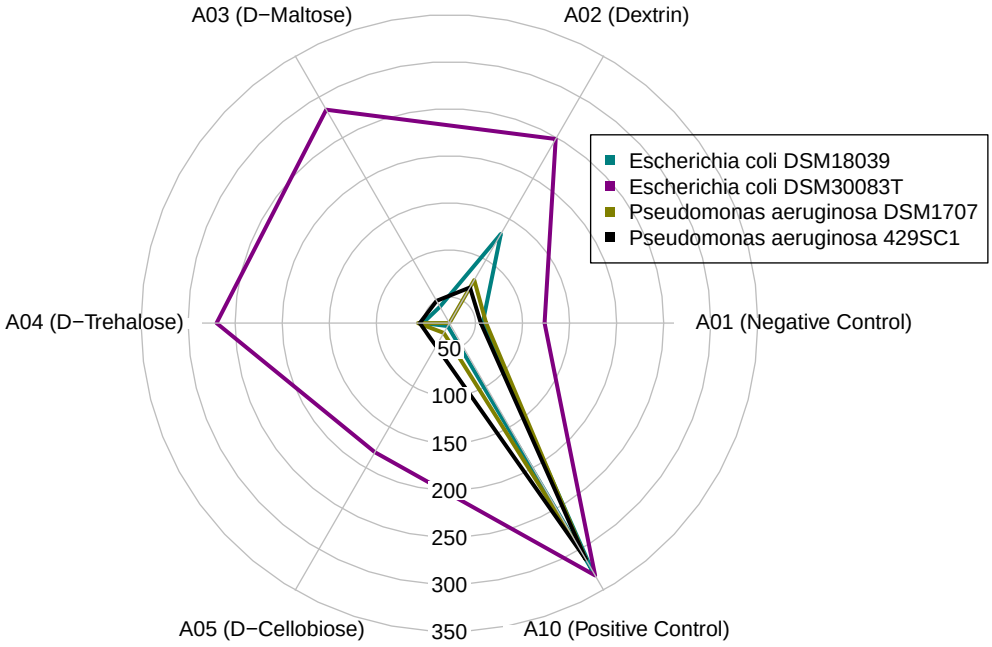

Figure 10: Comparison of point estimates for the parameter maximum height (A) observed from four strains, using `radial_plot()`. Shown are the results for estimating the maximum height of the single curves from wells A01 to A05 and A10.

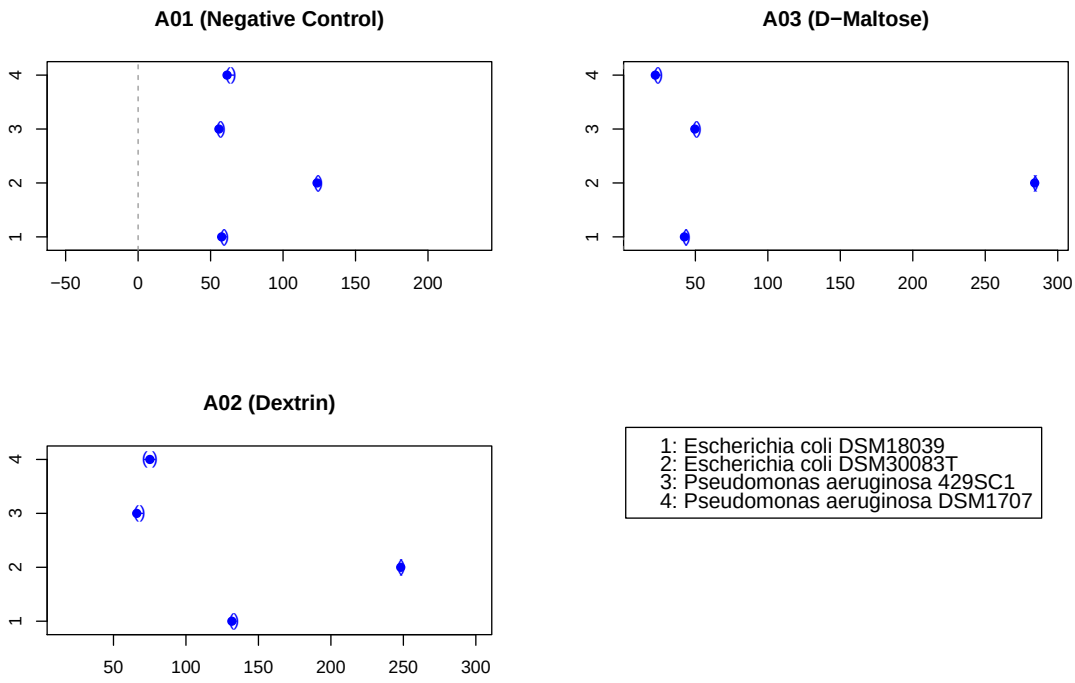

Figure 11: Comparison of point estimates and their 95% confidence intervals for the parameter maximum height (A) observed from four strains, using `ci_plot()`. Shown are the results for estimating the maximum height of the single curves on the three wells A01 (Negative Control), A02 (Dextrin) and A03 (D-Maltose) as indicated by the sub-plot titles. Point estimates that have no overlapping confidence intervals are regarded to be significantly different. But note that here the confidence intervals only indicate the uncertainty in parameter estimation from single curves.

After the extraction of the values together with necessary metadata (argument `as.labels`) in a first call to `extract()`, the resulting data frame can be treated by `extract()` again for generating another data frame with numeric values grouped according to the `as.groups` argument and optionally normalisation applied, as triggered *via* the argument `norm.per`. The first data frame would be created as follows:

```
R> x <- extract(vaas_et_al, as.labels = list("Species", "Strain"),
     dataframe = TRUE)
```

For a better understanding of the following secondary applications of `extract()` it is highly recommended to take a look at the results from plotting the data with `ci_plot()` and also at structure of the created data frames.

Using `norm.per = "none"` causes normalisation to be omitted. If `as.groups = TRUE` is used, all metadata that have been included in the first data frame are used to determine the groups. The result is shown in Fig 12, after a further selection of columns from the second data frame to be passed to `ci_plot()`.

```
R> # without normalisation
R> ci_plot(extract(x, as.groups = TRUE,  norm.per = "none")[, c(1:7, 13)],
     legend.field = NULL, x = 350, y = 0)
```

Normalisation can be applied by subtracting plate means (`norm.per = "row"`). Per default, this would subtract the mean of each plate from each of its values (over all wells of that plate). Alternatively, well means can be subtracted (`norm.per = "column"`). Per default, this would subtract the mean of each well from each of its values (over all plates in which this well is present). Division instead of subtraction is also possible (`subtract = FALSE`). The following code would first normalize with the plate means, then with the well means:

```
R> ci_plot(extract(x, as.groups = TRUE,  norm.per = "row")[, c(1:7, 13)],
     legend.field = NULL, x = 150, y = 0)
R> ci_plot(extract(x, as.groups = TRUE,  norm.per = "column")[, c(1:7, 13)],
     legend.field = NULL, x = 150, y = 0)
```

Moreover, *via* `norm.by` it is possible to use one to several selected wells or plates for the calculation of the means used for normalisation. With `direct = TRUE` even directly entered numeric values can be used for normalisation purposes. See Figure 13 for an example of plotted confidence intervals obtained from data normalized by subtracting the value of well A10 ("Positive Control"). Note that due to the structure of the data frame `norm.per = "row"` in combination with the `norm.by` argument has to be used. One could normalize by subtracting the means of well A10 only as follows:

```
R> ci_plot(extract(x, as.groups = TRUE,  norm.per = "row",
     norm.by = 10, subtract = TRUE)[, c(1:7, 13)],
     legend.field = NULL, x = 0, y = 0)
```

Additional example code on on visualizing curve parameters is available *via* `opm_files("examples")`. It is indeed easy to conduct principal-component analysis (PCA) with matrices created with `extract()`. Examples are given based on the **BiodiversityR** package (Kindt and Coe 2005).

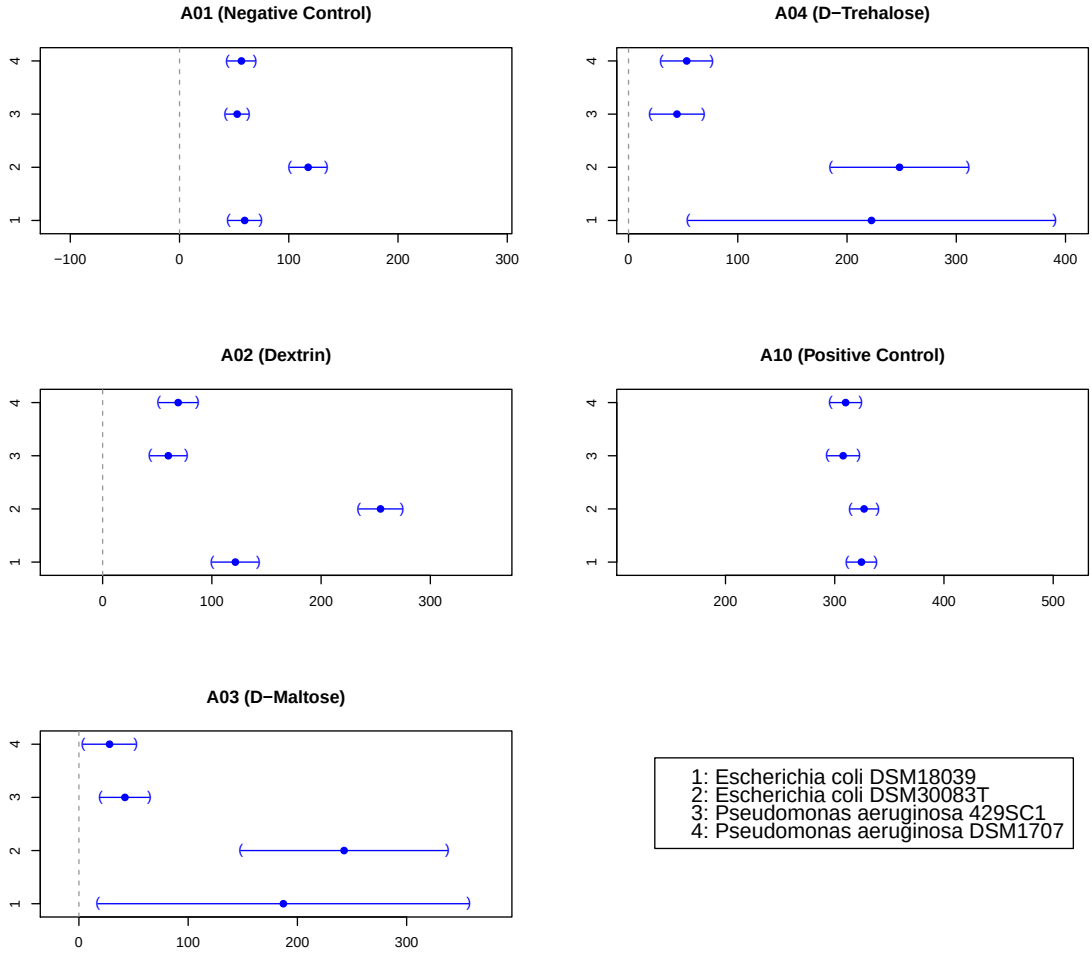

Figure 12: Comparison of mean point estimates and their 95% confidence intervals, computed with `extract()` over groups defined by the "Species" and "Strain" metadata entries, for the parameter maximum height (A) observed from four strains, using `ci_plot()`. Shown are the results on the three wells A01 (Negative Control), A02 (Dextrin), A03 (D-Maltose), A04 (D-Trehalose) and A10 (Positive Control) as indicated by the sub-plot titles. Normalisation was not used for this plot. Point estimates that have no overlapping confidence intervals are regarded to be significantly different. Compare this with Figure 13.

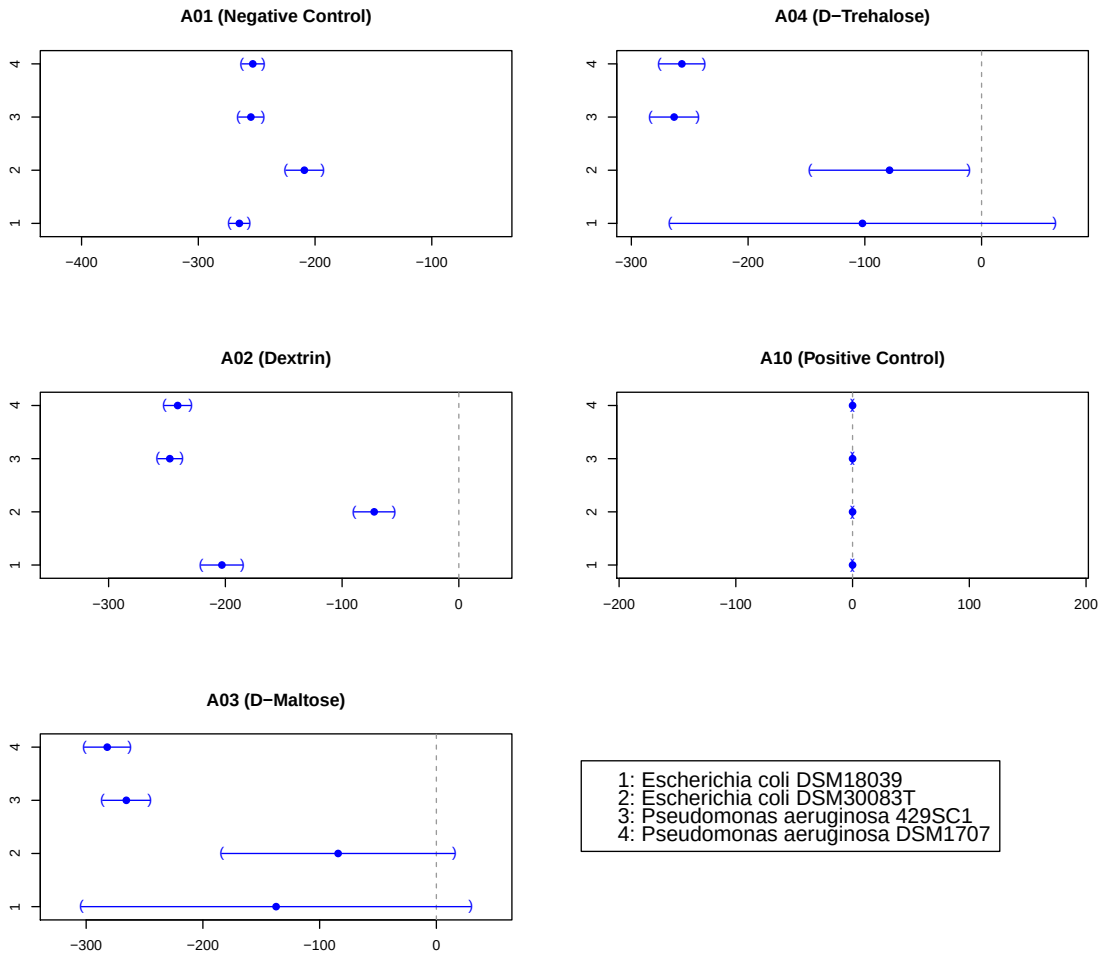

Figure 13: Comparison of mean point estimates and their 95% confidence intervals, computed with `extract()` over groups defined by the "Species" and "Strain" metadata entries, for the parameter maximum height (A) observed from four strains, using `ci_plot()`. Shown are the results on the three wells A01 (Negative Control), A02 (Dextrin), A03 (D-Maltose), A04 (D-Trehalose) and A10 (Positive Control) as indicated by the sub-plot titles. Normalisation was done by subtracting the overall well means of well A10 ("Positive Control"). Point estimates that have no overlapping confidence intervals are regarded to be significantly different. Compare this with Figure 12.

3.9. Statistical comparisons of group means

The `opm_mcp()` function allows the user to test for differences in the means of multiple groups directly on OPMS objects, obtaining the factors that determine the grouping structure from the stored metadata or the wells. In the following, the general concept and the application of the function is explained using several examples for groups defined within wells, across wells, or across metadata-based groups. Detailed explanations on how the graphical and numerical output of the results has to be interpreted are provided.

3.9.1. Tukey type of comparison: all-against-all

This paragraph addresses the comparison of a single well type across different plates organised into multiple groups. We compare four distinct strains, each of which being represented by ten replicates of GEN-III microplate measurements. The experimental question to be addressed relates to a single well: "Do these four bacteria differ in the mean value of curve parameter A on well G06?" (see Figure 14). This type of comparison is termed "Tukey"-type contrasts (all-against-all) because each strain is compared to each other.

The example data is taken from the first biological replicate included in `vaas_et_al`:

```
R> vaas.G06 <- subset(vaas_et_al[, , "G06"],
     list(Experiment = "First replicate"))
```

The resulting dataset of four strains, each represented by the ten replicates, is shown in Figure 14.

To solve the statistical question, we now perform a multiple comparison of group means using `opm_mcp()`.

As explained in section 2.9, the initial step is the, in the `opm_mcp()` methods internally executed, reshape of the data into a data frame containing one column for the chosen parameter, one column for the well (substrate) name, another column for the values itself and optionally additional columns for the selected metadata. By using the argument `output = "data"` the data frame created by `opm_mcp()` can be shown. Accordingly, the code below gives the head of the respective data frame for the example data containing the "A" values (maximum height) of the well G06 (α-Keto-Glutaric Acid) from four strains and 10 plates, respectively:

```
R> head(x <- opm_mcp(vaas.G06, model = ~ Strain, m.type = "aov",
     linfct = c(Tukey = 1), output = "data", full = FALSE))

    Strain Parameter Well    Value
1 DSM18039         A  G06 265.4947
2 DSM18039         A  G06 262.8439
3 DSM18039         A  G06 252.1659
4 DSM18039         A  G06 267.9070
5 DSM18039         A  G06 258.1758
6 DSM18039         A  G06 261.9873
```

For performing the testing procedure, a model composition has to be stated that specifies the factor levels that determine the grouping. The groups to be compared (and here to be selected from the metadata beforehand) are defined by the argument `model`.

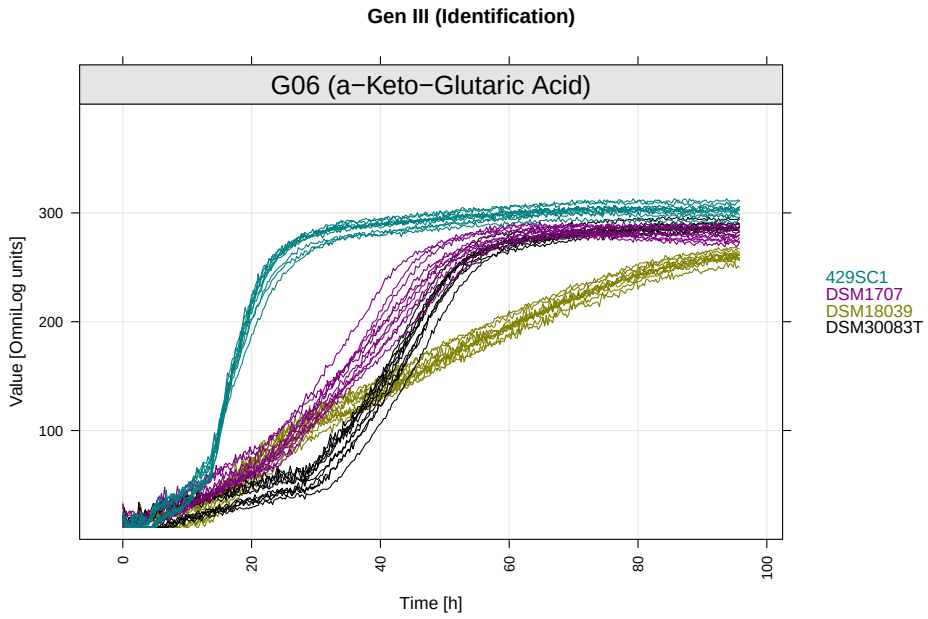

Figure 14: PM curves from the ten technical replicates of the first biological repetition plotted using `xy_plot()`. The respective curves from all four strains are superimposed; the affiliation to each strain is indicated by colour (see the legend). The x axis shows the measurement time in hours, the y axis the measured colour intensities in OmniLog® units.

The argument `m.type` specifies the type of model to be used for fitting, either a linear model (`lm`), a generalized linear model (`glm`), or an analysis-of-variance model (`aov`).

Via stating the name of the desired contrast type in the argument `linfct`, the user can define the set of comparisons to be computed in the multiple comparison. The contrast matrix determines from which `model`-defined groups the means should be compared and how.

In our example, a Tukey-type contrast matrix is used (a more detailed explanation of predefined contrast matrices is given in the help page of `multcomp::contrMat()`).`opm_opt("contrast.type")` would be inserted if names were missing. The test results in a set of six two-sided pairwise comparisons between all four strain means (all possible pairs). The results of the comparison can be written into an object, in our example it is termed `vaas.G06.mcp`:

```
R> vaas.G06.mcp <- opm_mcp(vaas.G06, model = ~ Strain, m.type = "aov",
     linfct = c(Tukey = 1))
```

Since the model statement can be arbitrarily complex, the argument `linfct` offers flexibility to address specific variables for the performance of the testing procedure. The `linfct` argument given as a numeric vector simply refers to the positions of the variable within `model` to be used for the testing procedure. Accordingly, by using 1 the first (and in this example, only term) “Strain” is selected.

Note that the structure of the arguments set by `model` and by `linfct` may become more complex if several metadata entries are involved in the testing. The user might therefore wish to recheck the way how `model` and by `linfct` will actually be transformed during the

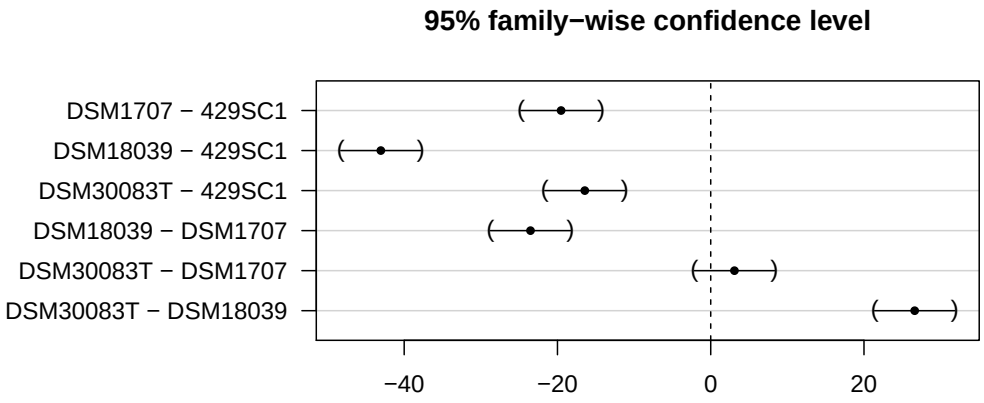

Figure 15: Comparisons of group means from well G06 between the four exemplar strains calculated with `opm_mcp()` and the plotting method for the resulting object. On the y axis the performed comparisons are indicated as differences of the groups, determining which differences of means are computed. All pairwise comparisons are shown. The filled black circle indicates the point estimator of difference between the mean of groups. 95% confidence intervals for are indicated by horizontal bars and parentheses. Note the differences between interpretation of this figure and the Figures obtained with `ci_plot()` in Section 3.8, as explained in the main text.

execution of the statistical test. Usage of the argument `model` can be checked by outputting just the converted argument:

```
R> opm_mcp(vaas.G06, model = ~ Strain, m.type = "aov",
     linfct = c(Tukey = 1), output = "model")
```

Similarly, the usage of the argument `linfct` can be checked as follows:

```
R> opm_mcp(vaas.G06, model = ~ Strain, m.type = "aov",
     linfct = c(Tukey = 1), output = "linfct")
```

For the computation of the tests itself, `opm_mcp()` uses `glht()` from the package **multcomp** and returns an object of class `opm_glht` (which inherits from `glht`). As shown in Figure 15, the results of the performed statistical test can be plotted using the methods available for objects of that class (see `?multcomp::glht` for details).

```
R> library(multcomp) # now needed
R> old.mar <- par(mar = c(3, 15, 3, 2)) # adapt margins in the plot
R> plot(vaas.G06.mcp)
R> par(old.mar) # reset to default plotting settings
```

A summary of the numerical results can be obtained as follows:

```
R> mcp.summary <- summary(vaas.G06.mcp)
R> mcp.summary$model$call <- NULL # avoid some unnecessary output
R> mcp.summary
```

```
         Simultaneous Tests for General Linear Hypotheses

Multiple Comparisons of Means: Tukey Contrasts


Linear Hypotheses:
                           Estimate Std. Error t value Pr(>|t|)
DSM1707 - 429SC1 == 0       -19.527      1.938 -10.076   <1e-04 ***
DSM18039 - 429SC1 == 0      -43.047      1.938 -22.213   <1e-04 ***
DSM30083T - 429SC1 == 0     -16.432      1.938  -8.479   <1e-04 ***
DSM18039 - DSM1707 == 0     -23.520      1.938 -12.136   <1e-04 ***
DSM30083T - DSM1707 == 0      3.095      1.938   1.597    0.393
DSM30083T - DSM18039 == 0    26.615      1.938  13.734   <1e-04 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Adjusted p values reported -- single-step method)
```

The interpretation of confidence intervals for differences of means is somewhat distinct from the interpretation of the confidence intervals for the point estimators for curve parameters as discussed in Section 3.8. The point estimator for differences of means represents the computed difference of the considered group means and, analogously, the size of the CI indicates the reliability of this difference. If the 95% CI for differences of means includes zero (dashed vertical lines in Figure 15) there is no significant difference between the group means. Conversely, if zero is not included, a statistically significant difference is indicated. Furthermore, the more distant the 95% CI is from zero, the larger the biological effect size, i.e. the real difference between the group means.

In the here shown example all group means of the curve parameter "A" are statistically significant different from each other ($p < 0.001$), except for the comparison of strains DSM 30083^T and DSM 1707 (Figure 15).

For an explanation of the graphical representation of the CIs, consider the last comparison in Figure 15. The comparison of the mean A value from strain DSM 30083^T minus the mean "A" value of strain DSM 18039 results in 26.615 units as the point estimator of this difference which is plotted accordingly on the x-axis. That is, on average the "A" values from well G06 and strain DSM 30083^T are 26.615 units larger than the "A" values of strain DSM 18039. The detailed numeric outcome would be obtained using `summary()` for the test results or `confint(vaas.G06.mcp)` for numerics of both the point estimator and confidence intervals. Additionally, each point estimator for the difference of means comes with a 95% CI providing information about the statistical significance of the test, the effect size and the variability of the mean differences. They are plotted as usual.

In our example, the results from the statistical calculations indicate that all significant differences are even *highly* significant ($p < 0.001$). However, the size of p-values does not say anything about the size of the differences of means and thus not anything about the biological relevance of the statistical significance. By computation of the CI around the point estimator for a difference of means, the user gets the information about how large the difference between two considered groups in mean is. Thus an assessment about the biological relevance of a statistical significance is facilitated by considering an information about the difference on the original scale of measurement. For a meaningful biological interpretation of the results it is therefore highly recommended to also consider the effect size rather than taking only the p

values into account.

For illustration purposes consider the position of the 95% CI from the difference between strains DSM 18039 and 429SC1 (i.e, the effect size in the second comparison in Figure 15), which is much larger than between strains DSM 30083^{T} and 429SC1 (smaller effect size in the third comparison).

3.9.2. Dunnett-type comparison: one-against-all

This paragraph describes another type of comparison of the means of multiple groups, which is the comparison of a single, selected well against all other wells available in the dataset. This type of comparison is termed "Dunnett"-type contrasts (one-against-all). In the example below, we compare the wells among themselves. Accordingly, the groups are defined by the wells rather than the measured organisms or others. The reference well can be either the negative or positive control but also one of the substrates, for example, serving as a standard in a specific chemical group. The following data example is again taken from the first biological replicate included in `vaas_et_al`, but this time only the type strain of *Escherichia coli*, measured in ten technical replicates, is selected.

```
R> vaas.e.coli <- subset(vaas_et_al,
     list(Experiment = "First replicate", Strain = "DSM30083T"))
```

For convenience, we perform the tests only for the first ten wells. The comparison of all wells against the negative control in A01 can now be performed by calling:

```
R> opm_mcp(vaas.e.coli[, , 1:10], output = "mcp", model = ~ Well,
     linfct = c(`Dunnett_A01 (Negative Control)` = 1))
```

Please note a special feature substantially simplifying the choice of the reference group: The value for the `linfct` argument can be constructed by typing `Dunnett`, plus, separated by any sign, e.g. underscore ("_"), the level name which should serve as the reference group in the contrast set. The next example shows the Dunnett-type comparison with well A03 chosen as the reference group.

```
R> mcp.A03 <- opm_mcp(vaas.e.coli[, , 1:10], output = "mcp", model = ~ Well,
     linfct = c(Dunnett_A03 = 1), full = FALSE)
R> mcp.summary <- summary(mcp.A03)
R> mcp.summary$model$call <- NULL # avoid some unnecessary output
R> mcp.summary

         Simultaneous Tests for General Linear Hypotheses

Multiple Comparisons of Means: User-defined Contrasts


Linear Hypotheses:
               Estimate Std. Error z value Pr(>|z|)
A01 - A03 == 0 -165.358      3.213 -51.468   <0.001 ***
A02 - A03 == 0  -39.304      3.213 -12.233   <0.001 ***
A04 - A03 == 0  -10.187      3.213  -3.171   0.0117 *
```

```
A05 - A03 == 0 -110.982      3.213 -34.543   <0.001 ***
A06 - A03 == 0   -1.376      3.213  -0.428   0.9997
A07 - A03 == 0 -121.911      3.213 -37.945   <0.001 ***
A08 - A03 == 0 -146.956      3.213 -45.740   <0.001 ***
A09 - A03 == 0 -137.566      3.213 -42.817   <0.001 ***
A10 - A03 == 0   40.219      3.213  12.518   <0.001 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Adjusted p values reported -- single-step method)
```

3.9.3. Pairs-type comparison of groups: pairwise comparisons as defined by specific combinations of metadata entries

This paragraph describes a more specific and hence complex design of the group structure by making a more distinct use of the stored metadata and its potential combinations. Assume data from two species, *P. aeruginosa* and *E. coli* (data set `vaas_4`), each with two plates restricted to the wells A01, A02, A03, and H02.

A combination of two species × four well types would yield eight different groups, which are each represented by two plates. A Tukey-type comparison (all-against-all) would then result in 28 pairwise comparisons, whereas a Dunnett-type comparison (one-against-all) would result in seven pairwise comparisons.

However, assume the user is only interested in a specific subset of pairwise comparisons defined by questions such as "For each well, is there a difference between the two species?" This experimental question resulted in testing four statistical hypotheses, as there are only four pairwise combinations that fit this question, since for each of the four wells, the two species should be compared.

This user-defined set of comparisons can easily be performed by applying the specially designed `linfct` argument "Pairs". The user needs to take care that "Well" is part of the model and is joined with at least one other factor extracted from the metadata, in this case with "Species". This can, be achieved on-the-fly with the `J()` pseudofunction as shown below. Note that the resulting model factor "Well.Species" contains eight levels, i.e. four groups per plate.

```
                 Species Parameter Well     Value              Well.Species
1        Escherichia coli         A  A01  57.66618       A01/Escherichia coli
2        Escherichia coli         A  A01 123.45581       A01/Escherichia coli
3  Pseudomonas aeruginosa         A  A01  61.35526 A01/Pseudomonas aeruginosa
4  Pseudomonas aeruginosa         A  A01  55.74738 A01/Pseudomonas aeruginosa
5        Escherichia coli         A  A02 131.67996       A02/Escherichia coli
6        Escherichia coli         A  A02 248.18087       A02/Escherichia coli
7  Pseudomonas aeruginosa         A  A02  75.10225 A02/Pseudomonas aeruginosa
8  Pseudomonas aeruginosa         A  A02  66.05093 A02/Pseudomonas aeruginosa
9        Escherichia coli         A  A03  42.45742       A03/Escherichia coli
10       Escherichia coli         A  A03 284.09938       A03/Escherichia coli
11 Pseudomonas aeruginosa         A  A03  22.37216 A03/Pseudomonas aeruginosa
12 Pseudomonas aeruginosa         A  A03  49.63049 A03/Pseudomonas aeruginosa
13       Escherichia coli         A  H02  48.75757       H02/Escherichia coli
14       Escherichia coli         A  H02  63.62915       H02/Escherichia coli
15 Pseudomonas aeruginosa         A  H02 294.68878 H02/Pseudomonas aeruginosa
16 Pseudomonas aeruginosa         A  H02 312.19430 H02/Pseudomonas aeruginosa
```

As explained above, the name of the `linfct` value indicates the type of contrast used for the testing procedure. Unless explicitly specified, "Pairs" selects the first subcomponent of the selected (joined) model component for the comparisons. The use of explicitly setting `linfct = c(Pairs.Well = 1)` ensures that for all levels present in the first (joined) component of the model, i.e, `Well`-wise, all pairwise comparisons are performed among the different groups present (here: the two species *P. aeruginosa* and *E. coli*).

The result of this analysis is shown below.

```
R> y <- opm_mcp(vaas_4[, ,c( 1:3, 86)], model = ~ J(Well, Species),
     m.type = "aov", linfct = c(Pairs.Well = 1), full = FALSE)

         Simultaneous Tests for General Linear Hypotheses

Multiple Comparisons of Means: User-defined Contrasts


Linear Hypotheses:
                                                        Estimate Std. Error t value
`A01/Pseudomonas aeruginosa` - `A01/Escherichia coli` == 0   -32.01      69.68  -0.459
`A02/Pseudomonas aeruginosa` - `A02/Escherichia coli` == 0  -119.35      69.68  -1.713
`A03/Pseudomonas aeruginosa` - `A03/Escherichia coli` == 0  -127.28      69.68  -1.827
`H02/Pseudomonas aeruginosa` - `H02/Escherichia coli` == 0   247.25      69.68   3.549
                                                        Pr(>|t|)
`A01/Pseudomonas aeruginosa` - `A01/Escherichia coli` == 0   0.9816
`A02/Pseudomonas aeruginosa` - `A02/Escherichia coli` == 0   0.3753
`A03/Pseudomonas aeruginosa` - `A03/Escherichia coli` == 0   0.3240
`H02/Pseudomonas aeruginosa` - `H02/Escherichia coli` == 0   0.0271 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Adjusted p values reported -- single-step method)
```

Especially, if models are more complex, e.g., if more than two metadata entries would be joined by the `J()` function, it is highly recommended to state the names of the metadata entries for which the pairwise comparisons should be performed by appending it directly to the `Pairs` argument. For instance in the example the metadata name `Species` can directly be addressed by using `linfct = c(Pairs.Species = 1)` and results in the pairwise all-against-all comparisons of the selected four wells *within each* of the two species *Escherichia coli* and *Pseudomonas aeruginosa*.

```
R> y <- opm_mcp(vaas_4[, , c(1:3, 86)], model = ~ J(Well, Species), m.type = "aov",
               linfct = c(Pairs.Species = 1), full = FALSE)
R> mcp.summary <- summary(y)
R> mcp.summary$model$call <- NULL # avoid some unnecessary output
R> mcp.summary

         Simultaneous Tests for General Linear Hypotheses

Multiple Comparisons of Means: User-defined Contrasts
```

```
Linear Hypotheses:
                                                                  Estimate Std. Error
`A02/Escherichia coli` - `A01/Escherichia coli` == 0               99.37      69.68
`A03/Escherichia coli` - `A01/Escherichia coli` == 0               72.72      69.68
`H02/Escherichia coli` - `A01/Escherichia coli` == 0              -34.37      69.68
`A03/Escherichia coli` - `A02/Escherichia coli` == 0              -26.65      69.68
`H02/Escherichia coli` - `A02/Escherichia coli` == 0             -133.74      69.68
`H02/Escherichia coli` - `A03/Escherichia coli` == 0             -107.09      69.68
`A02/Pseudomonas aeruginosa` - `A01/Pseudomonas aeruginosa` == 0    12.03      69.68
`A03/Pseudomonas aeruginosa` - `A01/Pseudomonas aeruginosa` == 0   -22.55      69.68
`H02/Pseudomonas aeruginosa` - `A01/Pseudomonas aeruginosa` == 0   244.89      69.68
`A03/Pseudomonas aeruginosa` - `A02/Pseudomonas aeruginosa` == 0   -34.58      69.68
`H02/Pseudomonas aeruginosa` - `A02/Pseudomonas aeruginosa` == 0   232.86      69.68
`H02/Pseudomonas aeruginosa` - `A03/Pseudomonas aeruginosa` == 0   267.44      69.68
                                                                  t value Pr(>|t|)
`A02/Escherichia coli` - `A01/Escherichia coli` == 0                1.426   0.7298
`A03/Escherichia coli` - `A01/Escherichia coli` == 0                1.044   0.9070
`H02/Escherichia coli` - `A01/Escherichia coli` == 0               -0.493   0.9974
`A03/Escherichia coli` - `A02/Escherichia coli` == 0               -0.383   0.9994
`H02/Escherichia coli` - `A02/Escherichia coli` == 0               -1.919   0.4604
`H02/Escherichia coli` - `A03/Escherichia coli` == 0               -1.537   0.6684
`A02/Pseudomonas aeruginosa` - `A01/Pseudomonas aeruginosa` == 0    0.173   1.0000
`A03/Pseudomonas aeruginosa` - `A01/Pseudomonas aeruginosa` == 0   -0.324   0.9998
`H02/Pseudomonas aeruginosa` - `A01/Pseudomonas aeruginosa` == 0    3.515   0.0581 .
`A03/Pseudomonas aeruginosa` - `A02/Pseudomonas aeruginosa` == 0   -0.496   0.9973
`H02/Pseudomonas aeruginosa` - `A02/Pseudomonas aeruginosa` == 0    3.342   0.0734 .
`H02/Pseudomonas aeruginosa` - `A03/Pseudomonas aeruginosa` == 0    3.838   0.0376 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Adjusted p values reported -- single-step method)
```

This yields two (for each species) x six (all-against-all among four wells = six) = 12 pairwise comparisons for which the adjustment of multiplicity has been undertaken.

When dealing with more complex models also keep in mind that the numeric vector in `linfct` can refer to the position of any variable, or set of variables obtained by joining, within `model`.

3.9.4. User-defined comparisons of interest

For performing even more specific comparisons of interest, the user can provide a contrast matrix directly. A contrast for multiple comparison procedures is defined as a linear combination of two or more factor level means (averages) whose coefficients add up to zero (Hochberg and Tamhane 1987). To demonstrate the principle of a contrast matrix, an all-against-all comparison (a "Tukey"-type contrast) of four groups is performed using a toy-example.

```
R> n <- c(10, 20, 30, 40)
R> names(n) <- paste0("group", 1:4)
R> contrMat(n, type = "Tukey")

         Multiple Comparisons of Means: Tukey Contrasts


               group1 group2 group3 group4
```

```
group2 - group1     -1      1      0      0
group3 - group1     -1      0      1      0
group4 - group1     -1      0      0      1
group3 - group2      0     -1      1      0
group4 - group2      0     -1      0      1
group4 - group3      0      0     -1      1
```

In each line, a pair of group-wise comparisons is defined by the locations of the non-zero values. For instance, in the first line, the 1 and -1 values indicate that the means of `group1` are subtracted from the means of `group2`. The function `contrMat()` from **multcomp** (see `?multcomp::contrMat`) provides an overview of the predefined contrast types that can be used in the `opm_mcp()` argument `linfct`.

In the example from above (see Figure 15), a "Tukey"-type contrast was used to trigger the comparison of all groups against all others in the data set. The underlying contrast matrix used to set up the contrasts can be viewed by entering

```
R> summary(vaas.G06.mcp)$linfct
```

```
                    (Intercept) StrainDSM1707 StrainDSM18039 StrainDSM30083T
DSM1707 - 429SC1              0             1              0               0
DSM18039 - 429SC1             0             0              1               0
DSM30083T - 429SC1            0             0              0               1
DSM18039 - DSM1707            0            -1              1               0
DSM30083T - DSM1707           0            -1              0               1
DSM30083T - DSM18039          0             0             -1               1
attr(,"type")
[1] "Tukey"
```

Accordingly, the user is free to set up contrast matrices for `opm_mcp()` that define the comparisons of interest. However, a `model` argument is necessary for definition of the factors that determine the groups and thus the possibilities for comparisons. As the next example, we compare the overall performance of the tested organisms in the four wells A01 to A04.

Although the user typically expects these wells to be in order in an OPMS object, this actually may have been changed by a previous well selection. Moreover, details of the the implementation of the conversion of OPMS objects to data frames, and of reshaping these data frames, can be subject to change, which might also affect the order of factor levels within the final data passed to `glht()`. Hence, it should be avoided to set up a contrast matrix fully by hand. Instead, `opm_mcp(output = "contrast")` yields one to several template contrast matrices, which are guaranteed to match the used OPMS object. We highly recommend to generate those template matrices and modify them according to specific user needs.

For instance, the following output contains a contrast matrix with all possible comparisons (because "Tukey" is used) for `Well` as factor variable in the correct order for the first four wells of `vaas_4`:

```
R> contr <- opm_mcp(vaas_4[, , 1:4], model = ~ Well, linfct = c(Tukey = 1),
     output = "contrast", full = FALSE)
R> contr
```

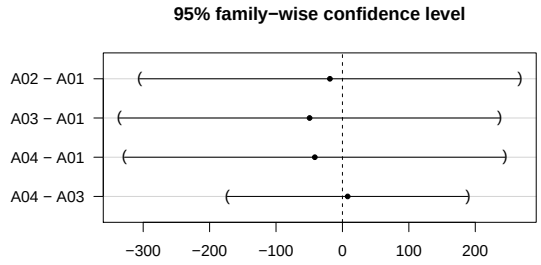

Figure 16: Point estimates and 95% confidence intervals in a manually defined comparison of group means for a specifically selected set of wells (A01 to A04) from the `vaas_4` exemplar object. The picture was obtained by running `opm_mcp()` and then the plotting function for the resulting object. Compare with Figure 15, where details on the axis annotation are given.

The `contr` object is a named list of contrast matrices, with one matrix per factor selected. An according call of the `opm_mcp()` function including the selecting of some comparisons of interest is:

```
R> vaas4.mcp <- opm_mcp(vaas_4[, , 1:4], model = ~ Well, m.type = "lm",
     linfct = contr$Well[c(1:3, 6), ], full = FALSE)
```

Since `output = "contrast"` does not work in this situtation, the correct set-up of the contrast matrix can be controlled by:

```
R> summary(vaas4.mcp)$linfct
```

```
          A01 A02 A03 A04
A02 - A01  -1   1   0   0
A03 - A01  -1   0   1   0
A04 - A01  -1   0   0   1
A04 - A03   0   0  -1   1
```

As mentioned above, the outcome can be visualized using the `plot()` method for `glht` objects (see Figure 16).

Note that the `model` argument defines the group means available for comparisons. In the following example "Species" contains only two levels ("*Pseudomonas aeruginosa*" and "*Escherichia coli*"). Thus, irrespective of the stated contrast type, only one comparison is possible.

```
R> vaas4.mcp <- opm_mcp(vaas_4, model = ~ Species, m.type = "lm",
     linfct = mcp(Species = "Dunnett"))
```

Finally, besides the multiple comparison of single group means as described above, it is also possible to compare averages from several subgroups with a single other subgroup or averages from several other subgroups. For example, the user may be interested in comparing the data shown in Figure 17 at the level of groups that may contain different data sets as subgroups.

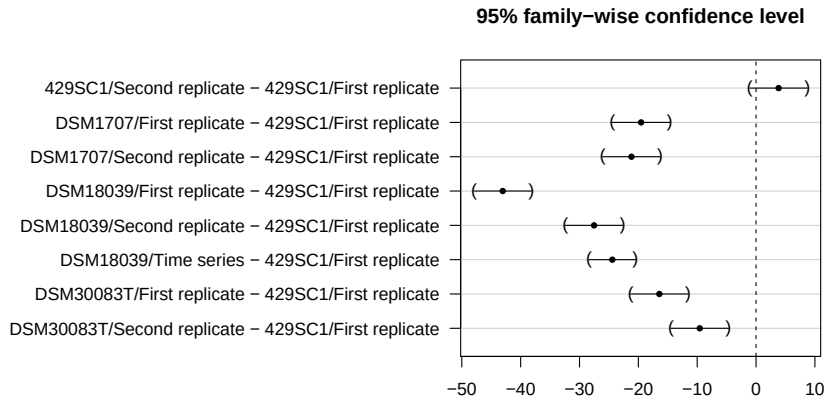

Figure 17: Point estimates and 95% confidence intervals in a Dunnett-type comparison of group means for a *cell-means model* for the `vaas.G06` exemplar object. In analogy to Figure 16, the picture was obtained by running `opm_mcp()` and then the plotting function for the resulting object. Compare also with Figure 15, where details on the axis annotation are given.

```
R> vaas.G06 <- vaas_et_al[, , "G06"]
R> vaas.G06.mcp <- opm_mcp(vaas.G06, model = ~ J(Strain, Experiment),
     linfct = c(Dunnett = 1))
```

The result is shown in Figure 17, visualized using `plot()` as described above.

When building a contrast matrix, keep in mind that the levels of the model-defining factor needs to match the columns of the contrast matrix, in order. For this reason, it is advantageous to work with a template contrast matrix generated with `opm_mcp()` from the object under study and check the positioning of its column names prior to any modification:

```
R> contr <- opm_mcp(vaas.G06, model = ~ J(Strain, Experiment),
     linfct = c(Dunnett = 1), output = "contrast")$Strain.Experiment
R> colnames(contr)
```

The user is then free to choose other values than just 0 and 1 for the coefficients, provided that each contrast sums up to zero. In the example below, the contrast matrix is reduced to three contrasts of interest, in which the values 0, -1/4 , 1/4, and 1 are used. The reader might have noted that the "First replicate" entries are in columns 1, 3, 5 and 8, whereas the "Second replicate" entries are in columns 2, 4, 6 and 9 and the "Time series" entries are in column 7 of the object `contr`. This information is sufficient to set up a correct contrast matrix for the following three contrasts of interest:

```
R> contr <- contr[1:3, ] # keeps the column names
R> rownames(contr) <- c(
     "First repl. - Second repl.",
     "First repl. - Time series",
     "DSM 1707 #1 - Second repl."
   )
R> contr[1, ] <- c(1/4, -1/4, 1/4, -1/4, 1/4, -1/4, 0, 1/4, -1/4)
```

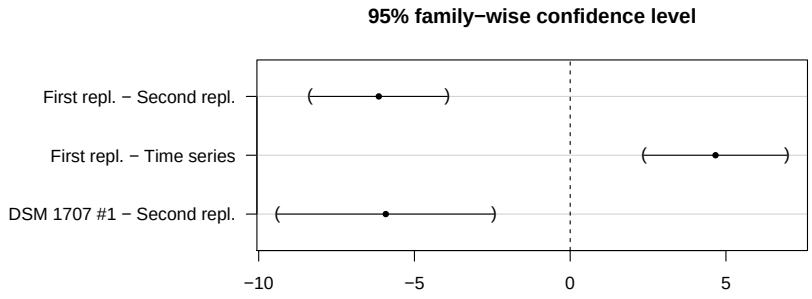

Figure 18: Point estimates and 95% confidence intervals in a user defined comparison of group means for a *cell-means model* for the `vaas.G06` exemplar object. Like Figure 16, the picture was obtained by running `opm_mcp()` and then the plotting function for the resulting object. Compare also with Figure 15, where details on the axis annotation are given.

```
R> contr[2, ] <- c(1/4, 0, 1/4, 0, 1/4, 0, -1, 1/4, 0)
R> contr[3, ] <- c(0, -1/4, 1, -1/4, 0, -1/4, 0, 0, -1/4)
R> contr
R> vaas6.mcp <- opm_mcp(vaas.G06, model = ~ J(Strain, Experiment), m.type = "lm",
     linfct = mcp(Strain.Experiment = contr))
```

The resulting visualisation of this entirely user-defined contrast matrix is shown in Figure 18.

3.10. Discretization

After calculating curve parameters and optionally generating a suitable subset, data can be discretized and optionally also exported for analysis with external phylogeny software or for inclusion into a scientific manuscript as text or table. In the **opm** manual and help pages, the functions relevant for either task are contained in the families "discretization-functions", "phylogeny-functions" and partially also in "naming-functions", with according cross-references. Much like `do_aggr` for aggregation, `do_disc` should be preferred for discretization. By default it works on the A parameter (see Figure 4) but this can be modified.

3.10.1. Discretization and phylogenetic data export

Restricting the `vaas_et_al` example dataset to the two biological replicates yields an orthogonal dataset with 2×10 replicates for each of the four strains for which we can calculate discretized parameters:

```
R> vaas.repl <- subset(vaas_et_al,
     query = list(Experiment = c("First replicate", "Second replicate")))
R> vaas.repl <- do_disc(vaas.repl)
```

Note that the resulting objects is an OPMS object with OPMD objects as elements. Such objects contain discretized values, available *via* `discretized()`, and the discretization settings used, which can be obtained using `disc_settings()`. This works much like `aggregated()` and `aggr_settings()` explained above. `disc_settings()` also yields the computed discretization cutoffs. The `subset()` function has a `positive` argument that allows one to

create a subset containing only the wells that were positive in at least one plate or in all plates, as well as a corresponding `negative` argument. The effect of either could be modified with `subset(invert = TRUE)`. For example, the command `xy_plot(subset(vaas_4, positive = "all"), neg.ctrl = NULL)` would plot only those wells in which all curves have been classified by k-means partitioning to yield a positive reaction. See the manual for further details, using `help(subset, package = "opm")`.

3.10.2. Discretization and export of text

The `listing()` methods of the OPMD and OPMD classes create textual descriptions of the discretization results suitable for the direct inclusion in scientific manuscripts.

```
R> listing(vaas.repl, as.groups = NULL)
R> listing(vaas.repl, as.groups = list("Species"))
```

As usual, the results can be grouped according to specified metadata entries using the "as.groups" argument. If this yields ambiguities (such as a negative reaction of the same well on one plate and a positive reaction on another plate), the result is accordingly renamed. The "cutoff" argument can be used to define filters, keeping only those values that occur in a specified minimum proportion of wells. See the manual for details, using `help(listing, package = "opm")`.

The `listing()` function returns a character vector or matrix with the S3 class "OPMD_listing" or "OPMS_listing", allowing for a special `phylo_data()` function that further formats these objects. Accordingly, the following code snippets

```
R> phylo_data(listing(vaas.repl, as.groups = NULL))
R> phylo_data(listing(vaas.repl, as.groups = list("Species")))
```

would yield character scalars better suitable for exporting into text files using `write`. It is also possible to generate HTML output, yielding formatted text. Try

```
R> phylo_data(listing(vaas.repl, as.groups = NULL, html = TRUE))
R> phylo_data(listing(vaas.repl, as.groups = list("Species"), html = TRUE))
```

and note that the `phylo_data()` function has a "html.args" argument. Textual HTML output supports most of the formatting instructions for the output of HTML tables described below (see 3.10.3). Note particularly how formatting *via* a CSS file works, as described in Section 3.10.3.

The default settings of `do_disc()` imply exact k-means partitioning into three groups ("negative", "ambiguous" and "positive"), treating all contained plates together, and using the maximum-height parameter for discretization. Let A_1 and A_2 be the maximum-height parameters from two curves C_1 and C_2, respectively, and let us assume that $A_1 \geq A_2$ holds. The algorithm then guarantees that if C_2 is judged as positive reaction then C_1 is also judged as positive; if C_2 is weak then C_1 is not negative; if C_1 is negative then C_2 is negative; and if C_1 is weak then C_2 is not positive. In this sense, the results will be consistent, but there are not many other things the algorithm guarantees. Note particularly that always three clusters result by default (one can omit the middle cluster, i.e. the "weak" reactions), irrespective of

the input data. This is usually unproblematic if the data contain both really negative and really positive reactions, but data that in reality are negative throughout, or uniformly positive, would nevertheless be split into three (or two) clusters. That is, additionally checking the curve heights and particularly the "cutoffs" entry obtained *via* `disc_settings()` should initially be mandatory.

It is also possible to make the reactions uniform within metadata-defined groups. This would be specified with the `unify` argument and would deliberately deviate from the kind of consistency described above. The unification approach replaces the primary discretization results with the most frequent value within the respective combination of group and well if this value is present in a given proportion of the original values and with `NA` otherwise. The according cutoff is set using `opm_opt(min.mode = ...)` or directly. Thus there are two distinct meanings of "ambiguous" reactions, as ambiguity either results from the clustering of the parameters, or by clustering results that deviate between distinct experimental replications. It is unnecessary and perhaps not preferable to use both approaches together, i.e. to cluster into three groups only and then also unify. Note that `listing()` and `phylo_data()` would use the same unification approach, if requested.

The manual describes the other discretization approaches available in **opm**, such as using `best_cutoff()` instead of k-means partitioning, and using subsets of the plates, specified using stored metainformation. See `?do_disc`.

3.10.3. Discretization and export of tables

The HTML created by **opm** deliberately contains no formatting instructions. Rather, it is possible (and recommended) to link it to a CSS file. CSS ("Cascading Style Sheets") is a style-sheet language used for defining the formatting of a document written in a markup language such as HTML.

As the generated HTML is richly annotated with "class" attributes, which not only provide information on the structure of the file but also on the depicted data, very specific formatting can be obtained just by modifying one to several associated CSS files. For the following example, we set the default CSS file to be linked from the generated HTML to the file that comes with **opm**.

```
R> opm_opt(css.file = grep("[.]css$", opm_files("auxiliary"), value = TRUE))
```

One could now easily create an HTML table from the discretized data and write it to a file:

```
R> vaas.html <- phylo_data(vaas.repl, format = "html",
     as.labels = list("Species", "Strain"), outfile = "vaas.html")
```

A practical problem is that the resulting HTML file is linked to its CSS file with a fixed path. The formatting would thus get lost once the HTML file was copied to another system, without a warning. So users might want to copy the predefined CSS file to the working directory and set it as default:

```
R> file.copy(grep("[.]css$", opm_files("auxiliary"), value = TRUE),
     "opm_styles.css", overwrite = TRUE)
R> opm_opt(css.file = "opm_styles.css")
```

The generated HTML would subsequently be linked to this file, and the two files could be distributed together. The same mechanism works for text generation using `listing()` (see 3.10.2). In addition to the default CSS file, a complete list of the settings that can be modified with this function is available *via* `?opm_opt`.

Users who want to define their own CSS files can start with modifying the file shipped with **opm**. Microsoft Windows users should consider that the path to the file must be provided in UNIX style, as obtained, e.g., using `normalizePath(x, winslash = "/")` if `x` is the path to the file. This is according to World Wide Web standards and not determined by **opm**.

By default columns with measurement repetitions as specified using `as.labels` are joined together. The `delete` argument specifies how to reduce the table: either not at all or keeping only the variable, parsimony-informative or non-ambiguous characters. The legend of the table is as used in taxonomic journals such as the *International Journal of Systematic and Evolutionary Microbiology* (http://ijs.sgmjournals.org/) but could also be adapted. Users can modify the headline, add sections before the table legend, or before or after the table. The title and the "meta" entries of the resulting HTML can also be modified. The `phylo_data()` methods have an auxiliary function, `html_args`, which assists in putting together the arguments that determine the shape and content of the HTML output. See the manual for further details, using `?html_args`.

3.10.4. Fine-tuning the discretization

One can also conduct discretization step-by-step by using the functions `best_cutoff()` or `discrete()` after extracting matrices from the OPMS object. This offers more flexibility (such as additional discretization approaches, e.g. the creation of multiple-state characters) but is also more tedious than using `do_disc()`.

```
R> vaas.repl <- subset(vaas_et_al,
     query = list(Experiment = c("First replicate", "Second replicate")))
R> vaas.repl <- extract(vaas.repl,
     as.labels = list("Species", "Strain", "Experiment", "Plate number"))
```

The A parameter (see Figure 4) can be discretized into (per default) 32 states using the theoretical range of 0 to 400 OmniLog® units (see Section 2.10):

```
R> vaas.repl.disc <- discrete(vaas.repl, range = c(0, 400))
```

This yields (at most) 32 distinct character states corresponding to the 32 equal-width intervals within 0 and 400. Exporting the data in *extended PHYLIP format* readable by RAxML (Stamatakis *et al.* 2005) would work as follows:

```
R> phylo_data(vaas.repl.disc, outfile = "example_replicates.epf")
```

The other supported formats are PHYLIP, NEXUS and TNT (Goloboff *et al.* 2008). For discretizing the data not in equally spaced intervals but into binary characters including missing data, or ternary characters with a third, intermediary state between "negative" and "positive" the gap mode of `discrete()` can be used:

```
R> vaas.repl.disc <- discrete(vaas.repl, range = c(120.2, 236.6), gap = TRUE)
```

Here the range argument provides not the overall boundaries of the data as before (at least as large as the real range), but the boundaries of a zone within the real range of the data corresponding to an area of ambiguous affiliation. That is, values below 120.2 are coded as "0", those above 236.6 as "1", and those in between as "?". The values used above were determined by k-means partitioning of the A values from the `vaas_et_al` dataset (Vaas *et al.* 2012); there is currently no conclusive evidence that they can generally be applied. The last command would result in the treatment of values within the given range as "missing data" (`NA` in R, "?" if exported). To treat them as a third, intermediary character state, set `middle.na` to `FALSE`:

```
R> vaas.repl.disc <- discrete(vaas.repl, range = c(120.2, 236.6),
     gap = TRUE, middle.na = FALSE)
```

The three resulting states, coded as "0", "1" and "2" (in contrast to "0", "?" and "1" above) would have to be interpreted as "negative", "weak" and "positive". Exporting the data in one of the supported phylogeny formats would work as described above. If the `do_disc()` function described above calls `discrete()`, then only in gap mode and with `middle.na` set to `TRUE`, yielding a vector or logical matrix.

4. Discussion and conclusion

The high-dimensional sets of longitudinal data collected by the OmniLog® PM system call for fast and easily applicable (and extendable) data organisation and analysis facilities. The here presented **opm** package for the free statistical software R (R Development Core Team 2011) features not only the calculation of aggregated values (curve parameters) including their (bootstrapped) confidence intervals, but also provides a rather complete infrastructure for the management of raw kinetic values and curve parameters together with any kind of meta-information of relevance for the user (Vaas *et al.* 2012, 2013a).

The spline estimation and parameter calculation in the data-aggregation step of has been optimized for the analysis of PM data. One main issue in the spline-fitting procedure is the selection of suitable smoothing parameters. The methods included in **opm** provide not only the basic framework (Vaas *et al.* 2012) based on methods from the **grofit** package (Kahm *et al.* 2010), but also specifically adapted applications of `smooth.spline` and the **mgcv** package (Wood 2003; Eilers and Marx 1996)

The analysis toolbox of the package includes the implementation of a fully automated estimation of whether respiration kinetics should be classified as either a "positive" or "negative" (absent) physiological reaction. This dichotomization is apparently of high interest to many users of the OmniLog® PM system but would apparently be extremely biased as long as thresholds are chosen ad hoc and by eye. Users should nevertheless be aware that loss of information is inherent to discretizing continuous data.

The **opm** package enables the user to produce highly informative and specialized graphical outputs from both the raw kinetic data as well as the curve-parameter estimates. Moreover, the package provides simultaneous multiple comparisons of group means (Hothorn *et al.* 2008; Bretz *et al.* 2010; Hsu 1996) with an interface specifically adapted to the typical PM data objects. In combination with the functionality for annotating the data with meta-information

and then selecting subsets of the data, straightforward analyses regarding specific analytical questions can be performed without the need to invoke other R packages.

But since the design of the **opm** objects is not intended to be limited to specific analysis frameworks, the **opm** package works as a data containment providing well organized and comprehensive PM data for further, more specialized analyses using methods from different R packages or other third-party software tools. Particularly the generation of S4 objects featuring a rich set of methods as containers for either single or multiple OmniLog® PM plates enables not only the transfer of raw kinetic data into R but also eases their further processing. The complex data bundles can also be exported in YAML format, which is a human-readable data serialization format that can be read by most common programming languages and facilitates fast and easy data exchange between laboratories. If a proper YAML parser was unavailable, its subset JSON could also be used. The package can also generate CSV output files, but due to the limitation of this format these files cannot be read back into **opm** in a meaningful way (but into R).

These features render the **opm** package the first comprehensive toolbox for the management and a broad range of analyses of OmniLog® PM data. Its usage requires some familiarity with R, but is otherwise intuitive and straightforward also for biologists who are not used to command-line based software.

Power and limitations regarding usage of substrate information and their implementation for data arrangement and hypothesis testing are discussed in detail in the vignette "Working with substrate information in **opm**".

Last but not least, it might also be useful to provide functionality for a direct cross-talk between **opm** and database management systems. The current version is entirely file-based, and whereas powerful selection mechanisms for both input files and container objects for previously imported PM plates have already been implemented, future version could directly include database access. In the meantime, however, the output YAML format (or its subset, JSON) is likely to facilitate the quick establishment of third-party software for importing PM data into a database.

To summarize, we are convinced that the **opm** package already enables the users to analyse OmniLog® PM data in rather unlimited exploratory directions (Vaas *et al.* 2012, 2013a).

5. Acknowledgements

The help of Nora Buddruhs (DSMZ) and Anne Fiebig (DSMZ), who contributed a lot to the stored substrate information and to examples in earlier versions of this tutorial, is gratefully acknowledged. We owe very much to Hans-Peter Klenk (DSMZ), who brought the OmniLog® instrument to the DSMZ and supported this project in numerous ways. We thank Barry Bochner (BIOLOG Inc.), John Kirkish (BIOLOG Inc.), Andre Chouankam (BIOLOG Inc.), Jan Meier-Kolthoff (DSMZ), Pia Wüst (DSMZ), Stefan Ehrentraut (DSMZ) and Jörn Petersen (DSMZ) for helpful advice, as well as Victoria Michael (DSMZ) for technical support. We are also grateful to the maintainers of R-Forge and CRAN for providing the online resources used by this project. This work was supported by the German Research Foundation (DFG) SFB/TRR 51 and by the Microme project within the Framework 7 programme of the European Commission, which is gratefully acknowledged. JS gratefully acknowledges his support by DFG grant SI 1352/1-2.

References

Berger S, Stamatakis A (2010). "Accuracy of Morphology-Based Phylogenetic Fossil Placement under Maximum Likelihood." In *8th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-10)*. ACS/IEEE, Hammamet, Tunisia.

BiOLOG Inc (2009). *Converter, File Management Software, Parametric Software, Phenotype MicroArray, User Guide, Part 90333.* Biolog Inc., Hayward CA.

Bochner B (2009). "Global Phenotypic Characterization of Bacteria." *FEMS Microbiological Reviews*, **33**, 191–205.

Bochner B, Gadzinski P, Panomitros E (2001). "Phenotype MicroArrays for High Throughput Phenotypic Testing and Assay of Gene Function." *Genome Research*, **11**, 1246–1255.

Bochner B, Savageau M (1977). "Generalized Indicator Plate for Genetic, Metabolic, and Taxonomic Studies with Microorganisms." *Applied and Environmental Microbiology*, **33**, 434–444.

Bretz F, Hothorn T, Westfall P (2010). *Multiple Comparisons Using R.* CRC Press, Boca Raton.

Brisbin I, Collins C, White G, McCallum D (1987). "A New Paradigm for the Analysis and Interpretation of Growth Data: The Shape of Things to Come." *The Auk*, **104**, 552–553.

Broadbent J, Larsen R, Deibel V, Steele J (2010). "Physiological and Transcriptional Response of *Lactobacillus casei* ATCC 334 to Acid Stress." *Journal of Bacteriology*, **192**, 2445–2458.

Chambers J (1998). *Programming with Data.* Statistics and Computing. Springer-Verlag, New York.

Dougherty J, Kohavi R, Sahami M (1995). "Supervised and Unsupervised Discretization of Continuous Features." In A Prieditis, S Russell (eds.), *Machine Learning: Proceedings of the fifth international conference.*

Efron B (1979). "Bootstrap Methods: Another Look at the Jackknife." *The Annals of Statistics*, **7**, 1–26.

Eilers P, Marx B (1996). "Flexible Smoothing with B-splines and Penalties." *Statistical Sciences*, **11**, 89–121.

Farris J (1970). "Methods for Computing Wagner Trees." *Systematic Zoology*, **19**, 83–92.

Felsenstein J (2004). *Inferring Phylogenies.* Sinauer Associates, Inc., Sunderland, MA.

Fitch W (1971). "Towards Defining the Course of Evolution: Minimal Change for a Specified Tree Topology." *Systematic Zoology*, **20**, 406–416.

Goloboff P, Farris J, Nixon K (2008). "TNT, a Free Program for Phylogenetic Analysis." *Cladistics*, **24**, 774–786.

Hochberg A, Tamhane Y (1987). *Multiple Comparison Procedures.* Wiley.

Hothorn T, Bretz F, Westfall P (2008). "Simultaneous Inference in General Parametric Models." *Biometrical Journal*, **50**, 346–363. See vignette("generalsiminf", package = "multcomp").

Hsu J (1996). *Multiple Comparisons*. Chapman & Hall, London.

Kahm M, Hasenbrink G, Lichtenberg-Frate H, Ludwig J, Kschischo M (2010). "**grofit**: Fitting Biological Growth Curves with R." *Journal of Statistical Software*, **33**, 1–21. URL: http://cran.r-project.org/web/packages/grofit/.

Kindt R, Coe R (2005). *Tree diversity analysis. A manual and software for common statistical methods for ecological and biodiversity studies*. World Agroforestry Centre (ICRAF), Nairobi (Kenya). ISBN 92-9059-179-X, URL http://www.worldagroforestry.org/treesandmarkets/tree_diversity_analysis.asp.

Mahner M, Kary M (1997). "What Exactly are Genomes, Genotypes and Phenotypes? And what about Phenomes?" *Journal of Theoretical Biology*, **186**, 55–63.

Mayr E (1997). "The Objects of Selection." *Proceedings of the National Academy of Science USA*, **94**, 2091–2094.

Mithani A, Hein J, Preston G (2011). "Comparative Analysis of Metabolic Networks Provides Insight into the Evolution of Plant Pathogenic and Nonpathogenic Lifestyles in *Pseudomonas*." *Molecular Biology and Evolution*, **28**, 483–499.

Montero-Calasanz MdC, Göker M, Pötter G, Rohde M, Spröer C, Schumann P, Gorbushina AA, Klenk HP (2012). "*Geodermatophilus arenarius* sp. nov., a xerophilic actinomycete isolated from Saharan desert sand in Chad." *Extremophiles*, **16**, 903–909.

Montero-Calasanz MdC, Göker M, Rohde M, Schumann P, Pötter G, Spröer C, Gorbushina AA, Klenk HP (2013). "*Geodermatophilus siccatus* sp. nov., isolated from arid sand of the Saharan desert in Chad." *Antonie van Leeuwenhoek*, **103**, 449–456.

Quackenbush J (2002). "Microarray data normalization and transformation." *Nature Genetics*, **32**, 496–501.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL: http://www.R-project.org/.

Reinsch CH (1967). "Smoothing by spline functions." *Numerische Mathematik*, **10**, 177–183.

Schaarschmidt F, Vaas LA (2009). "Analysis of trials with complex treatment structure using multiple contrast tests." *HortScience*, **44**, 188–195.

Searle SR (1971). *Linear Models*. John Wiley & Sons, New York.

Selezska K, Kazmierczak M, Müsken M, Garbe J, Schobert M, Häussler S, Wiehlmann L, Rohde C, Sikorski J (2012). "*Pseudomonas aeruginosa* Population Structure Revisited under Environmental Focus: Impact of Water Quality and Phage Pressure." *Environmental Microbiology*, **14**, 1952–1967.

Stamatakis A, Ludwig T, Meier H (2005). "RAxML-III: A fast Program for Maximum Likelihood-Based Inference of Large Phylogenetic Trees." *Bioformatics*, **21**, 456–463.

Swofford D (2003). *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4.* Sinauer Associates, Sunderland, Massachusetts.

Tindall B, Kämpfer P, Euzéby J, Oren A (2006). "Valid publication of names of prokaryotes according to the rules of nomenclature: past history and current practice." *International Journal of Systematic and Evolutionary Microbiology*, **56**, 2715–2720.

Tukey J (1994). "The problem of multiple comparisons." *unpublished manuscript*. Reprinted in: Braun, H.I. (Ed.), The collected works of John W. Tukey. VIII Multiple Comparisons. Chapman & Hall, New York.

Vaas LA, Sikorski J, Hofner B, Fiebig A, Buddruhs N, Klenk HP, Göker M (2013a). "opm: An R Package for Analysing OmniLog®Phenotype MicroArray Data." *Bioinformatics*. doi:10.1093/bioinformatics/btt291. URL http://bioinformatics.oxfordjournals.org/content/early/2013/06/05/bioinformatics.btt291.full.pdf+html.

Vaas LA, Sikorski J, Michael V, Göker M, Klenk H (2012). "Visualization and Curve Parameter Estimation Strategies for Efficient Exploration of Phenotype MicroArray Kinetics." *PLoS ONE*, **7**, e34846. doi:10.1371/journal.pone.0034846.

Vaas LAI, Marheine M, Sikorski J, Göker M, Schumacher HM (2013b). "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.

Ventura D, Martinez T (1995). "An Empirical Comparison of Discretization Methods." In *Proceedings of the Tenth International Symposium on Computer and Information Sciences*, pp. 443–450. Morgan Kaufmann Publishers, San Francisco, CA.

Wang H, Song M (2011). "Ckmeans.1d.dp: optimal k-means clustering in one dimension by dynamic programming." *The R Journal*, **3**, 29–33.

Wood SN (2003). "Thin Plate Regression Splines." *Journal of the Royal Statistical Society. Series B*, **65**, 95–114.

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