respR – R Package Documentation

Januar Harianto *and* ***Nicholas Carey***

Contents

Chapter 1

Introduction

The R package `respR` provides a structural, reproducible workflow for the processing and analysis of respirometry data. Although the focus of the package is on aquatic respirometry, Functions in `respR` are largely unitless and can be used for the analysis of linear relationships in any time-series data. All analytical methods used in the package have been peer-reviewed and rigorously tested.

Use `respR` to:

- automatically **import** raw data from various oxygen sensing equipment;
- rapidly **test** data for issues before analysis;
- **explore** and visualise timeseries data;
- perform **regression** statistics on linear segments of data manually or automatically;
- **convert** units of oxygen consumption; and
- **export** results quickly for reporting.

The goal of this guide is to walk you through the `respR` package to import, analyse and convert your data. To begin, you will need to install the package.

Chapter 2

Installation

Use the `devtools` package to install a stable version of `respR`:

```
# install devtools
install.packages("devtools")
# use devtools to install respR from GitHub
devtools::install_github("januarharianto/respR")
```

The developmental version is mostly stable, and contains some features that are undergoing peer-review. You may install this version using the code:

```
# install dev version
devtools::install_github("januarharianto/respR", ref = "develop")
```

Once `respR` has been installed, load the package into your workspace:

```
library(respR)
```

Check out our [Quick start][quick start] guide if you are using `respR` for the first time. The Getting started vignette provides a detailed introduction to respirometry and information about other vignettes in this document.

VIGNETTES

Chapter 3

Getting started

3.1 Aquatic Respirometry

There are four broad methodological approaches in aquatic respirometry: *closed-chamber*, *intermittent-flow*, *flow-through* and *open-tank*.

In **closed-chamber** respirometry, O_2 decrease is measured within a hermetically sealed chamber of known volume, sometimes set within a closed loop to allow mixing of the environment within the chamber. Oxygen recordings may be continuous through use of an oxygen probe, periodic through withdrawing water or gas samples at set intervals, or a two-point measurement consisting of the initial and final concentrations. Metabolic rates are estimated from the O_2 timeseries by assuming a linear relationship between variables, and estimates of metabolic rate are straightforward in constant volume respirometry using the equation:

$$VO_2 = \dot{O}_2 V$$

where $\dot{O}_2$ is the slope of the regression that describes the rate of change in O_2 concentration over time, or in the case of a two-point measurement, the difference in O_2 concentration divided by time elapsed, and V is the volume of fluid in the container (Lighton 2008).

In **intermittent-flow** respirometry, O_2 concentration is measured as described above, but periodically the chamber is flushed with new water or air, returning it to initial conditions, resealed, and the experiment repeated (Svendsen et al. 2016). This technique is essentially the same as closed respirometry, but with the ability to conduct replicates easily. Depending on the metabolic rate metric being investigated, final respiration rate can be calculated as the mean of the measures (e.g. Carey et al. 2016), or the lowest or highest rates recorded in any trial (e.g. Stoffels 2015).

Flow-through respirometry involves a closed chamber, but with a regulated flow of air or water through it at a precisely determined rate. After equilibrium has been achieved, the oxygen concentration differential between the incurrent and excurrent channels, along with the flow rate, allows calculation of the oxygen extracted from the flow volume per unit time:

$$\dot{V}O_2 = (C_iO_2 - C_eO_2)FR$$

where $\dot{V}O_2$ is the rate of O_2 consumption over time, C_iO_2 and C_eO_2 are the incurrent and excurrent O_2 concentrations, and FR is the flow rate through the system (Lighton 2008).

A final method is **open-tank** respirometry, in which a tank or semi-enclosed area open to the atmosphere is used, but the input or mixing rate of oxygen from the surroundings has been quantified or found to be negligible relative to oxygen consumption of the specimens (Leclercq et al. 1999). It is seldom used, but for some applications it is a sufficient and practical methodology (Gamble et al. 2014). The common equation used for open respirometry is:

$$\dot{V}O_2 = \dot{O}_2 V + \phi_d$$

where $\dot{O}_2V$ is the slope of the regression that relates O_2 concentration to time, V is the volume of the arena and ϕ_d is the oxygen flux as determined by Fick's Law (Leclercq et al. 1999).

3.2 The respR R package

respR is a package designed to process the data from all of these types of respirometry experiment. It is designed primarily for aquatic respirometry, although because many of the main functions are unitless it is adaptable for use with gaseous respirometry, and indeed analysis of other data where a parameter may change over time.

When working with respirometry data, you will often need to:

1. Ensure that the data, or at least a **subset** of the data, is representative of the research question of interest.
2. Perform an initial analysis of the data to **estimate** the rate of change in oxygen concentration or amount.
3. Depending on the experimental setup, **correct** for background usage of oxygen by micro-organisms, or correct for oxygen flux from the air.
4. **Convert** the resulting usage rate to the volumetric and mass-specific rates in the appropriate units.

The respR package allows determination of common respirometry metrics and contains several functions to make this process straightforward.

- It provides visual feedback and diagnostic plots to help you explore, subset and analyse your data.
- It uses computational techniques such as *rolling regressions* and *kernel density estimates* to determine **maximum**, **minimum** or **most linear** rates within time-series data.
- The package takes an object-oriented approach, with all functions outputting objects which can be read by subsequent functions.
- By separating the workflow into a series of connected functions, you can "mix and match" functions to help you achieve your result.
- Output objects can also be saved or exported, and contain all raw data, parameters used in calculations, and results, allowing for a fully documented and reproducibile analysis of respirometry data.

3.3 Example Data

We have provided example data that can be used immediately once respR is loaded (`urchins.rd()`, `intermittent.rd()`, `zeb_intermittent.rd()`, `sardine.rd()`, `squid.rd()`, `flowthrough.rd()`).

```
data(package = "respR")
```

These data were obtained from actual experiments and more information can be obtained by invoking the `?` command in the R console, for instance, `?urchins.rd`.

3.4 Vignettes and documentation

We have prepared several vignettes describing typical analysis workflows and how respR works:

1. **Importing your data**
   respR has functions to make it easy to bring in and prepare your data, even from the raw data files output by various respirometry systems.

2. **Closed-chamber respirometry**
   This is a good place to start to understand the full functionality of `respR`. It describes an entire workflow to process and analyse a closed-chamber respirometry dataset.

3. **auto_rate: Automatic detection of metabolic rates**
   Here we introduce the function `auto_rate()` for the automatic detection and estimation of **maximum**, **minimum** and **most linear** rates.

4. **Performance of auto_rate in detecting linear regions**
   We use simulated data to test the accuracy of `auto_rate()`, and discuss the function's performance with focus on linear data detection.

5. **Comparative performance of auto_rate and LoLinR**
   We make comparisons to another R package, `LoLinR`, using data from their package and simulated data.

6. **Intermittent-flow respirometry: Simple example**
   Intermittent-flow respirometry: Complex example
   How to analyse simple and more complex intermittent-flow respirometry experiments.

7. **Flowthrough respirometry**
   Analysis of flowthrough respirometry data.

8. **PCrit**
   Determine P_{crit} in long-term, closed chamber respirometry experiments.

9. **Two-point analyses**
   Determine oxygen use rate using only two datapoints.

10. **Reproducibility**
    How `respR` has been designed to allow reporting of reproducible analyses.

11. **respR and the Tidyverse**
    Information about how `respR` integrates with `tidyverse` practices to streamline analytic workflows.

12. **A comparison of respR with other R packages**
    When to use respR
    The functionality and workflow of `respR` in comparison to other options, and when you should use it.

3.5 References

Carey, Nicholas, Januar Harianto, and Maria Byrne. "Sea Urchins in a High-CO 2 World: Partitioned Effects of Body Size, Ocean Warming and Acidification on Metabolic Rate." The Journal of Experimental Biology 219, no. 8 (April 15, 2016): 1178–86. https://doi.org/10.1242/jeb.136101.

Gamble, S., A.G. Carton, and I. Pirozzi. "Open-Top Static Respirometry Is a Reliable Method to Determine the Routine Metabolic Rate of Barramundi, Lates Calcarifer." Marine and Freshwater Behaviour and Physiology 47, no. 1 (January 2, 2014): 19–28. https://doi.org/10.1080/10236244.2013.874119.

Leclercq, N, Jean-Pierre Gattuso, and J Jaubert. "Measurement of Oxygen Metabolism in Open-Top Aquatic Mesocosms:Application to a Coral Reef Community." Marine Ecology Progress Series 177 (1999): 299–304. https://doi.org/10.3354/meps177299.

Lighton, John R. B. Measuring Metabolic Rates. Oxford University Press, 2008. https://doi.org/10.1093/acprof:oso/9780195310610.001.0001.

Stoffels, Rick J. "Physiological Trade-Offs Along a Fast-Slow Lifestyle Continuum in Fishes: What Do They Tell Us about Resistance and Resilience to Hypoxia?" Edited by Jodie L. Rummer. PLOS ONE 10, no. 6 (June 12, 2015): e0130303. https://doi.org/10.1371/journal.pone.0130303.

Chapter 4

Importing data

We designed `respR` to be a universal, end-to-end solution for analysing data and reporting analyses from any and all aquatic respirometry experiments, regardless of the equipment used. Therefore, it is system-agnostic; the data need only be put into a simple structure for a full analysis to be conducted. Indeed, the entire package (with the exception of the final conversion step in `convert_rate`) considers data to be unitless, so non-aquatic respirometry data, or any time-series data can be explored and analysed using `respR`.

Generic R data frame type objects, including `vectors` and objects of class `data.frame`, `data.table` and `tibble`, are recognised. The only structural data requirement is that time~O_2 data be in a specific form; paired values of numeric time-elapsed (in s, m or h) and oxygen amount (in any common unit). Every respirometry system, to our knowledge, allows data to be exported in such a format, or at least in a structure from which it is easy to parse it to this format. Two functions are provided to assist with bringing in and formatting your data correctly.

4.1 `import_file()`

Most systems allow data to be exported in easily readable formats (e.g. .csv, .txt) which contain the numeric time and O2 data `respR` requires. These files are usually easily imported into R using generic functions such as `read.csv()` and the relevant columns specified when used in `respR` functions, or extracted into separate data frames.

Many systems however have raw output files with redundant information, or a structure that confuses importing functions. For example Loligo Systems AutoResp and Witrox files have several rows of metadata above the columns of raw data, which causes importing problems in `read.csv()`. These files can be altered in Excel or other spreadsheet software to fix these issues, however `respR` allows importing of many raw data files from various systems without modification.

The `import_file()` function uses pattern recognititon to identify the originating system of the file. It also automatically recognises the format of any date-time data and uses it to create a new time-elapsed column, if one does not already exist.

Here's an example of importing a Witrox raw data file (from the current working directory, otherwise any external file can be specified with a filepath):

```
import_file("assets/Witrox_eg.txt")
```

```
## Loligo AutoResp/Witrox file detected
```

```
##       Date_Time_DDMMYYYY_HHMMSS Time_stamp_code Barometric_pressure_hPa
##    1:      5/11/2017 9:24:04 AM      3577364644                    1013
```

```
##    2:      5/11/2017 9:24:05 AM      3577364645                    1013
##    3:      5/11/2017 9:24:06 AM      3577364646                    1013
##    4:      5/11/2017 9:24:07 AM      3577364647                    1013
##    5:      5/11/2017 9:24:08 AM      3577364648                    1013
##   ---
## 6812:     5/11/2017 11:17:35 AM      3577371455                    1013
## 6813:     5/11/2017 11:17:36 AM      3577371456                    1013
## 6814:     5/11/2017 11:17:37 AM      3577371457                    1013
## 6815:     5/11/2017 11:17:38 AM      3577371458                    1013
## 6816:     5/11/2017 11:17:39 AM      3577371459                    1013
##       SDWA0003000060_CH_1_phase_rU SDWA0003000060_CH_1_temp_C
##    1:                        29.58                      14.01
##    2:                        29.58                      14.11
##    3:                        29.57                      14.14
##    4:                        29.59                      14.06
##    5:                        29.58                      14.07
##   ---
## 6812:                        30.47                      13.58
## 6813:                        30.49                      13.67
## 6814:                        30.47                      13.47
## 6815:                        30.49                      13.50
## 6816:                        30.47                      13.53
##       SDWA0003000060_CH_1_O2_mg/L
##    1:                      10.056
##    2:                      10.015
##    3:                      10.012
##    4:                      10.027
##    5:                      10.032
##   ---
## 6812:                       9.454
## 6813:                       9.403
## 6814:                       9.497
## 6815:                       9.469
## 6816:                       9.474
```

As we can see, the function automatically recognises that this is a Witrox file, removes redundant information, and renames the relevant columns. It also uses the date-time columns to calculate a numeric time elapsed column (called `time` or `elapsed` depending on the original file).

This function requires only a single input, the path to the file (one other option, `export = TRUE` allows exporting of the imported data to a .csv file). Everything else is handled automatically. This contrasts with other packages where numerous options such as the delimiter character, originating hardware, and specific date format must be specifed, which we have found leads to substantial usability issues (see A comparison of respR with other R packages).

After importing and saving to an object, this can be passed to the rest of the `respR` functions for processing, all while leaving the raw data file unmodified.

This function supports several systems at present (Firesting | Pyro | PRESENS OXY10 | PRESENS (generic) | MiniDOT | Loligo Witrox). However, it is still in development; some files may fail to import because of structural or version differences we have not encountered. We would encourage users to send us sample files for testing, especially any they have problems with, or from systems we do not yet support.

4.2 format_time()

For files types that are not yet supported, or if you have already imported your data by other means, the `format_time()` function will parse date-time to numeric time-elapsed, in the event the imported file does not contain this.

Here's an example of a 2 column data frame with date-time data and oxygen.

```
data <- import_file("assets/Witrox_eg.txt") %>%
  select(1, 6)
```

```
## Loligo AutoResp/Witrox file detected
```

```
head(data, n = 6)
```

```
##    Date_Time_DDMMYYYY_HHMMSS SDWA0003000060_CH_1_O2_mg/L
## 1:      5/11/2017 9:24:04 AM                      10.056
## 2:      5/11/2017 9:24:05 AM                      10.015
## 3:      5/11/2017 9:24:06 AM                      10.012
## 4:      5/11/2017 9:24:07 AM                      10.027
## 5:      5/11/2017 9:24:08 AM                      10.032
## 6:      5/11/2017 9:24:09 AM                      10.073
```

We can use `format_time` to parse these data to numeric (internally, `format_time` uses functionality in the package `lubridate`). The date-times can either be passed as a `vector` (for example, so it can be appended to the original data), or as a `data frame`. By default, the function assumes the date-time data are in the first column (i.e. `time = 1`), but this can be overridden by changing the `time` operator to specify the column index where the date-time data occurs. The resulting data frame will be identical (including column names), except a new column with the converted numeric time called `time.num` is added as the **first column**. We also need to specify the `format` of the date-times (see `?format_time` for further info):

```
## as vector
data_2 <- format_time(data$Date_Time_DDMMYYYY_HHMMSS, format = "dmyHMS")
head(data_2)
```

```
## [1] 1 2 3 4 5 6
```

```
## as data frame
data_3 <- format_time(data, format = "dmyHMS")
head(data_3)
```

```
##    Date_Time_DDMMYYYY_HHMMSS SDWA0003000060_CH_1_O2_mg/L elapsed
## 1:      5/11/2017 9:24:04 AM                      10.056       1
## 2:      5/11/2017 9:24:05 AM                      10.015       2
## 3:      5/11/2017 9:24:06 AM                      10.012       3
## 4:      5/11/2017 9:24:07 AM                      10.027       4
## 5:      5/11/2017 9:24:08 AM                      10.032       5
## 6:      5/11/2017 9:24:09 AM                      10.073       6
```

By default, the new time-elapsed data will start at zero, but we can override this. This could be useful if data are split into separate files, and you want to append the start of one onto the end of another, or you simply want to link a specific time elapsed value to the start of the experiment.

```
## as data frame
data_4 <- format_time(data$Date_Time_DDMMYYYY_HHMMSS, format = "dmyHMS", start = 1000)
head(data_4)
```

```
## [1] 1000 1001 1002 1003 1004 1005
```

Note, elapsed time data will always output in *seconds* regardless of the input format.

4.3 Dealing with timed events or notes

What if there are important notes or events associated with specific times in your experiment? For example, flushing of chambers, imposing a new swimming speed, changing the temperature, noting a response, etc. Resetting the times via formatting the time data may make these difficult to associate to certain stages of the analysis. This is easily dealt with by formatting the times of the events in the same way you formatted the data. You only need to make sure at least one event is associated with the same start time you used for experimental data.

Here's an example of some experimental notes (in some systems such notes can be entered in the software, and so may be included in output files, or they could be copied from a lab book into a .csv file and imported).

```
exp_notes
```

```
##                times                    events
## 1  8/17/2016 9:42:02          Experiment start
## 2  8/17/2016 9:52:02        Flush period start
## 3  8/17/2016 9:54:34          Flush period end
## 4 8/17/2016 10:19:02  Specimen acting normally
## 5 8/17/2016 12:04:54             Went to lunch
## 6 8/17/2016 14:31:22 Swim speed set to 20 cm/s
```

```
format_time(exp_notes, format = "mdyHMS")
```

```
##                times                    events elapsed
## 1  8/17/2016 9:42:02          Experiment start       1
## 2  8/17/2016 9:52:02        Flush period start     601
## 3  8/17/2016 9:54:34          Flush period end     753
## 4 8/17/2016 10:19:02  Specimen acting normally    2221
## 5 8/17/2016 12:04:54             Went to lunch    8573
## 6 8/17/2016 14:31:22 Swim speed set to 20 cm/s   17361
```

Such notes do not even have to be in the same date-time format, or even at the same precision, depending on how accurately you need to know when events occurred. The important factors are associating at least one event with the *same start time* used to format the experimental time data, and using the correct `format` setting.

```
exp_notes
```

```
##   times                    events
## 1  9:42          Experiment start
## 2  9:52        Flush period start
## 3  9:54          Flush period end
## 4 10:19  Specimen acting normally
## 5 12:04             Went to lunch
## 6 14:31 Swim speed set to 20 cm/s
```

```
format_time(exp_notes, format = "HM")
```

```
##   times                    events elapsed
## 1  9:42          Experiment start       1
## 2  9:52        Flush period start     601
## 3  9:54          Flush period end     721
## 4 10:19  Specimen acting normally    2221
## 5 12:04             Went to lunch    8521
## 6 14:31 Swim speed set to 20 cm/s   17341
```

4.4 Next steps

After your data is in this paired, numeric *time-elapsed~O2* form, it can be passed to `inspect()` or other functions for analysis.

Chapter 5

Closed-chamber respirometry

5.1 A typical respR workflow: Closed-chamber respirometry

Here we describe a typical workflow for a **closed-chamber** respirometry experiment. The example data used here is `urchins.rd`, where the first column of the data frame is *time* data, while the remaining 18 columns are dissolved O_2 data. Columns 18 and 19 contain background respiration recordings. The units are minutes and mg/L of O_2, however all analyses in `respR` are unitless, and we only consider units when we later come to convert rates.

```
head(urchins.rd)
```

```
## # A tibble: 6 x 19
##   time.min     a     b     c     d     e     f     g     h     i     j
##      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1      0    7.86  7.86  7.64  7.65  7.87  7.74  7.62  7.65  7.96  7.75
## 2      0.2  7.87  7.79  7.6   7.71  7.87  7.72  7.61  7.66  7.97  7.72
## 3      0.3  7.89  7.7   7.6   7.7   7.9   7.72  7.61  7.63  7.98  7.72
## 4      0.5  7.9   7.68  7.6   7.72  7.92  7.74  7.62  7.66  7.97  7.72
## 5      0.7  7.87  7.64  7.6   7.67  7.9   7.73  7.59  7.65  7.95  7.71
## 6      0.8  7.82  7.69  7.61  7.61  7.88  7.7   7.6   7.65  7.94  7.7
## # ... with 8 more variables: k <dbl>, l <dbl>, m <dbl>, n <dbl>, o <dbl>,
## #   p <dbl>, b1 <dbl>, b2 <dbl>
```

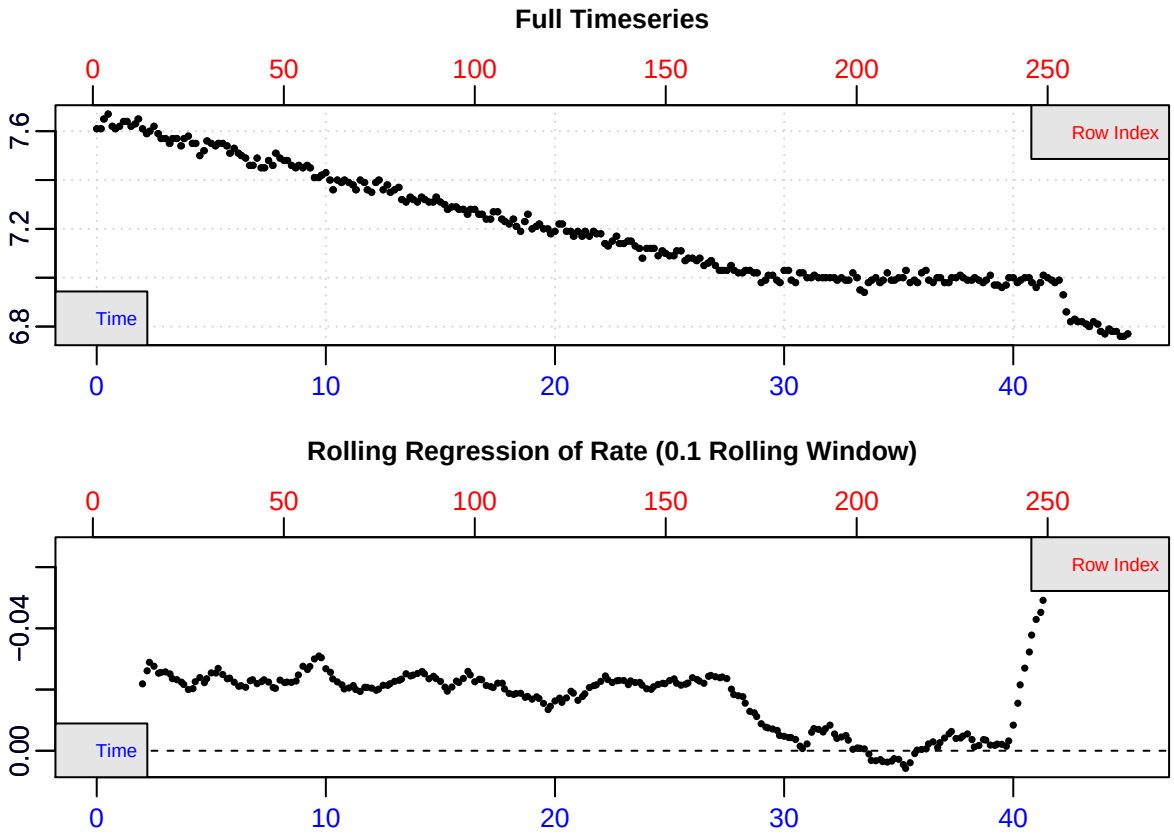
5.2 Step 1: Check for common errors

We first use `inspect()` to prepare the data and to scan for:

- Missing or non-numeric (`NA`/`NaN`) data
- Sequential time data
- Duplicate time data
- Evenly-spaced time data

By default, the function assumes the first column of the data frame is time data, while the second column is O_2 data. In the case of `urchins.rd` where a multi-column dataset is provided, the function defaults to using the first two columns. However, the `time =` and `oxygen =` arguments can modify that behaviour to select particular columns.

```
urchin <- inspect(urchins.rd, time =  1, oxygen = 15)
```




```
##
## # inspect # -----------------------------
##                time.min    n
## NA/NAN             pass pass
## sequential         pass    -
## duplicated         pass    -
## evenly-spaced      WARN    -
## Uneven time data locations  (first 20 shown) in column: time.min n
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

## Warning: Time values are not evenly-spaced.
```

From the plot, we can see irregularities in these data near the end of the timeseries (in this case the specimen had interfered with the oxygen sensor). A linear regression of the entire data series would therefore give an erroneous calculation of the true rate. However, the bottom output plot shows that over the initial stages of the experiment, oxygen uptake shows a consistent rate, and so in this experiment this section would be suitable for analysis.

The function also warns us that time data is not *numerically* evenly-spaced. However, *this does not mean the data cannot be processed*. Rather than make assumptions that rows represent evenly spaced datapoints, the functions in `respR` use actual time values for analyses and calculations, and so even irregularly spaced data are analysed correctly. This warning is for information purposes only: it is to make the user aware that if they use row numbers for manual operations such as subsetting, the same width may not represent the same time period. For now, the data frame is saved as an object, `urchin` which contains the original data columns we selected coerced into a data frame, and various other metadata.

It should be noted that using `inspect()` is optional - the main functions in `respR` will readily accept

regular R data structures (e.g. data frames, tibbles, vectors) as long as data are numeric and error-free. Running `inspect()` is a qualitative, exploratory step that highlights potential issues about the data before analysis. We use this particular example, with an obvious error towards the end, to illustrate the point that you should always visualise and explore your data before analysis. `respR` has been designed to make this easy.

Note there is an older version of this function called `inspect_data()` - this function has been deprecated, but kept in the package to maintain compatibility with older code. It will not be updated in the future, so users should use `inspect()`.

5.3 Step 2: Process background respiration

The presence of microorganisms may be a potential source of experimental bias, and we may want to account for background respiration rates during experiments. Since background rates typically account for a small percentage of experimental rates, these often-called "blank" experiments are routinely conducted alongside, or before and after main experiments, and often the rates are averaged across several datasets to obtain a more accurate estimate of the correction.

The function `calc_rate.bg()` can be used to simultaneously process multiple background rate measurements as long as they share the **same units of time and oxygen data as the experiments they will be used to correct**. In `urchins.rd`, background respiration was recorded and saved in columns 18 and 19. We analyse the data using the specialised function `calc_rate.bg()` and save the output as an object. Note the function allows us to subset the data, here by `time` from 5 to 40 minutes, to remove potentially erroneous sections at the start (or end) before the system had reached stability.

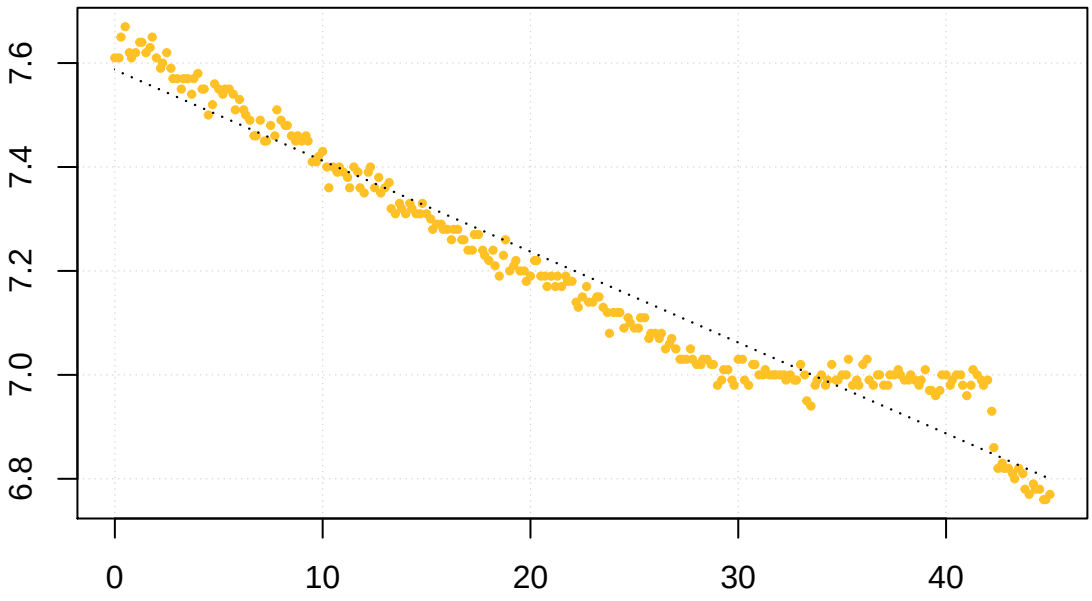
```
# bg <- calc_rate.bg(urchins.rd, xcol = 1, ycol = 18:19, from = 5, to = 40,
#   by = "time")
# print(bg)
```

This object contains both individual background rates for each data column entered, and an averaged rate which, by default, will be used as the correction rate when this is applied later in `adjust_rate`.

5.4 Step 3: Calculate oxygen uptake rate

Calling the function `calc_rate()` on the `inspect()` object, with no additional arguments, will prompt the function to perform a linear regression on the entire data series.

```
calc_rate(urchin) # same as: calc_rate(urchin$df)
```




```
##
## # calc_rate # --------------------
## Rate(s):
## [1] -0.01749242
```

Note how the function recognises the `inspect()` object. Alternatively, you can specify a `data.frame` object containing raw data, in which case the function will automatically consider the first column as time data, and the second column as dissolved O_2 data.

In many cases, there is a need to truncate or subset the data before rate is determined. For example, we may want to determine rate over an exact period of time, or within a threshold of O_2 concentrations. Equipment interference or other factors may cause irregularities in the data. We can work around such errors by subsetting the regions that are not erroneous and still obtain valid results.

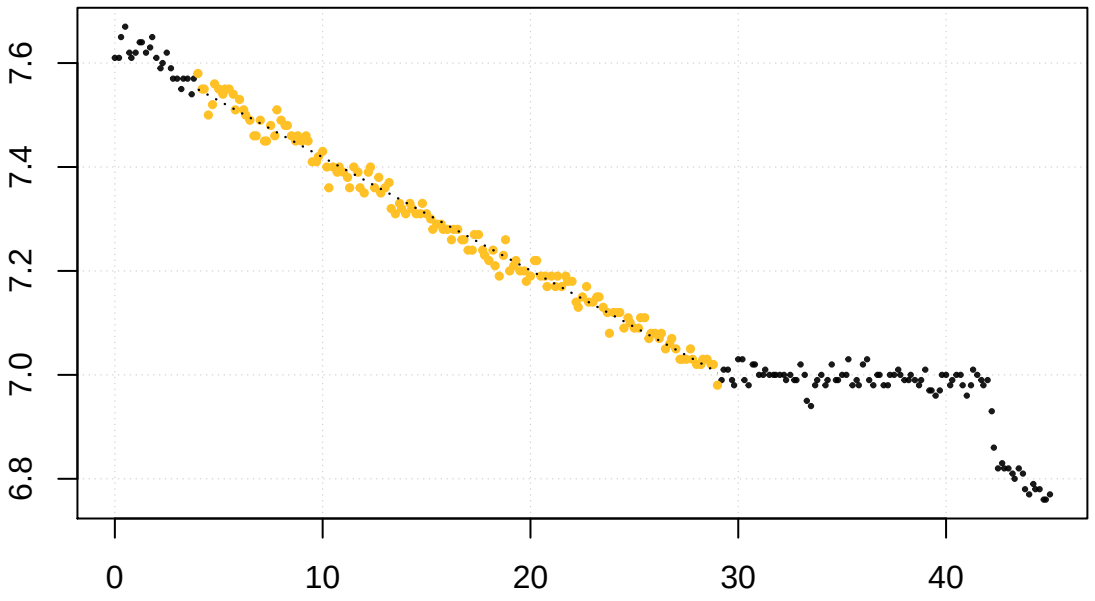
Based on the `from` and `to` arguments, a user may use `calc_rate()` to subset data in any of 4 ways:

1. **Time period** (`by = "time"`) - *"What is the rate over a specific 25 minute period?"*
2. **Total oxygen consumed** (`by = "o2"`) - *"At what rate is oxygen consumed between saturation points of 95% and 80%?"*
3. **Proportion based on total oxygen consumed** (`by = "proportion"`) - *"What is the rate from 4/5ths (0.8) to halfway (0.5) along the data?"*
4. **Precise subsetting by row** (`by = "row"`). - *"I'd like to determine rate between rows 11 and 273."*

We do not need to be overly precise; if input values of O_2 and time do not match exactly to a value in the data, the function will identify the closest matching values, rounded down, and use these for subsequent calculations.

Here we'll select a 25 minute period before the interference occurred:

```
rate <- calc_rate(urchin, from = 4, to = 29, by = "time")
```



The saved object can be exported to a file, or explored using generic R commands.

```
# uncomment to run and save as file
# write.csv(summary(rate), file = "results.csv")

print(rate)
```

```
##
## # calc_rate # --------------------
## Rate(s):
## [1] -0.02177588
```

```
summary(rate)
```

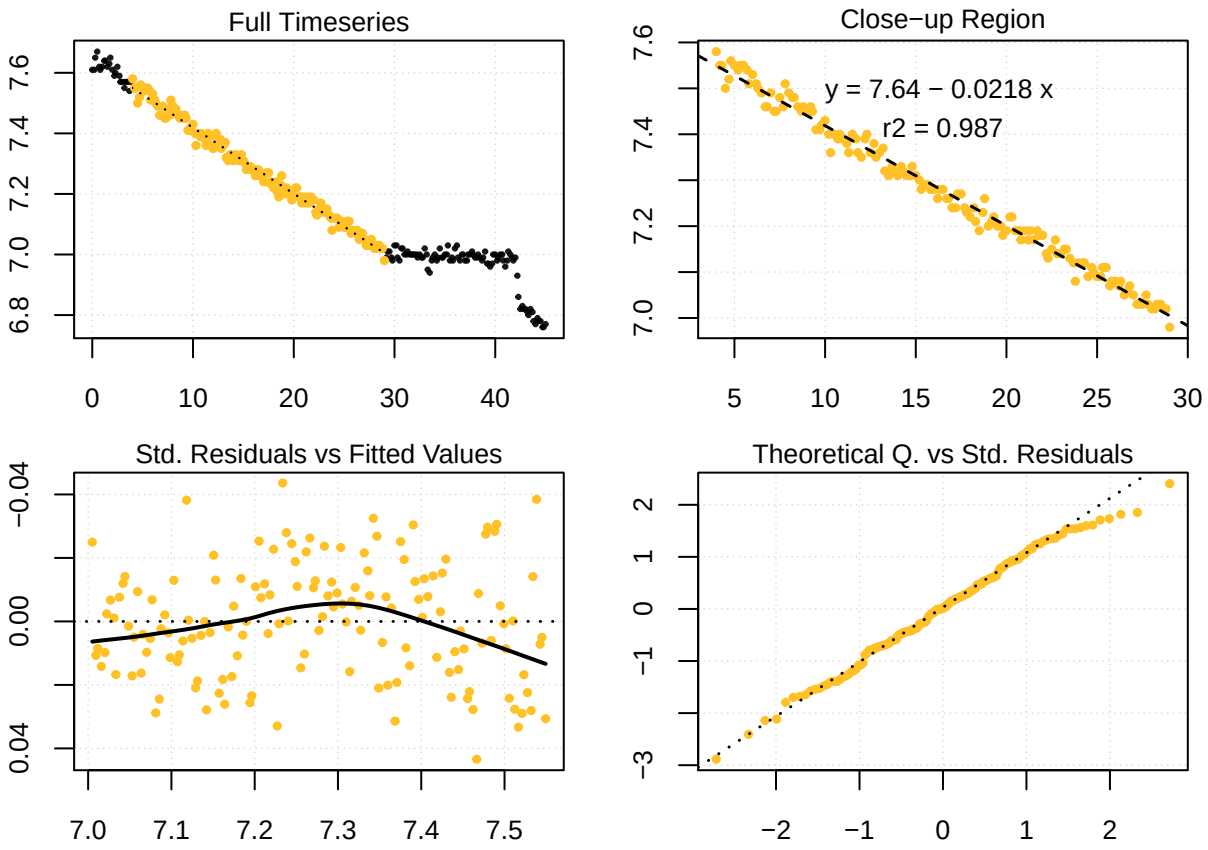
```
## Summary:
##    intercept_b0     rate_b1   rsq row endrow time endtime  oxy endoxy
## 1:     7.636454 -0.02177588 0.987  25    175    4      29 7.58   6.98
##    rowlength timelength rate_twopoint
## 1:       150         25        -0.024
```

The rate can be seen as the second entry `rate_b1`, and other summary data are saved in the object. The output also includes a `rate_2pt`. This is the rate determined by simple two-point calculation of difference in O_2 divided by difference in Time. For almost all analyses, the `rate_b1` should be used. See Two-point analyses for an explanation of this output and when it should be used.

Plotting the output provides a series of diagnostic plots of the data subset that was analysed.

```
plot(rate)
```

```
##
## # plot # ------------------------
## Plotting...this may take a while for large datasets.
```




```
## Done.
```

5.5 Step 4: Adjust for background respiration

Since background rate has been calculated in `calc_rate.bg()`, adjustment is straightforward using the function `adjust_rate()`. The `rate` input can be an object of class `calc_rate` or `auto_rate`, or any numeric value.

```
# a.rate <- adjust_rate(rate, bg)
# a.rate
```

A background correction can also be entered manually. Care should be taken to include the correct (typically negative) sign.

```
a.rate <- adjust_rate(rate, -0.00083)
```

```
##
## Rate adjustments applied. Use print() command for more info.
```

```
a.rate
```

```
##
## # adjust_rate # -------------------------
## Note: please consider the sign of the value while correcting the rate.
##
## Rank/position 1 result shown. To see all results use summary().
## Input rate: -0.02177588
## Adjustment: -0.00083
```

```
## Adj. rate: -0.02094588
```

For experiments where there is a quantified background **input** of oxygen, such as in *open-tank* respirometry, `adjust_rate()` can be used to correct rates using a *positive* background value.

```
a.rate <- adjust_rate(rate, 0.002)
```

```
##
## Rate adjustments applied. Use print() command for more info.
```

```
a.rate
```

```
##
## # adjust_rate # -------------------------
## Note: please consider the sign of the value while correcting the rate.
##
## Rank/position 1 result shown. To see all results use summary().
## Input rate: -0.02177588
## Adjustment: 0.002
## Adj. rate: -0.02377588
```

5.6 Step 5: Convert the results

Note, that until this point `respR` has **not required units of time or oxygen to be entered**. Here, we convert calculated, unitless rates to specified output units.

For example, we may want to calculate:

1. $\dot{V}O_2$ - Total change in O_2 per unit time within the chamber; or
2. $\dot{M}O_2$ - Mass-specific rate of change in O_2 per unit time of the specimen.

The function `convert_rate()` can be used to convert rate values to chamber volume and/or specimen mass specific values. This requires the units of the original data (`o2.unit`, `time.unit`), and in SI units (L, kg) the `volume` of fluid in the chamber, and `mass` of the specimen.

Note: the `volume` is volume of water in the respirometer, **not the volume of the respirometer**. That is, it represents the effective volume; a specimen might displace a significant proportion of the water, depending on its size. Therefore the volume of water entered here should equal the volume of the respirometer **minus the volume of the specimen**. It depends on your experiment how you determine water volume. There are several approaches to calculate the volume of a specimen; geometrically, through displacement in a separate vessel, or calculated from the mass assuming a density value (e.g. for fish it is often assumed they have an equal density as water, that is ~1000 kg/m^3). Volume could also be determined directly by pouring out the water at the end of the experiment, or by a weighing approach.

For an example of VO_2, or absolute oxygen uptake rate, we may convert the output of `calc_rate()` to O_2 consumed per hour:

```
convert_rate(a.rate,
             o2.unit = "mg/L",
             time.unit = "min",
             output.unit = "mg/h",
             volume = 1.09)
```

```
##
## # convert_rate # ------------------------
## Rank/position 1 result shown. To see all results use summary().
## Input:
## [1] -0.02377588
```

```
## [1] "mg/L" "min"
## Converted:
## [1] -1.554943
## [1] "mg/hour"
```

We can also convert the rate to a volume-corrected, mass-specific rate:

```
convert_rate(a.rate,
             o2.unit = "mgl-1",
             time.unit = "m",
             output.unit = "mg/s/kg",
             volume = 1.09,
             mass = 0.19)
```

```
##
## # convert_rate # ------------------------
## Rank/position 1 result shown. To see all results use summary().
## Input:
## [1] -0.02377588
## [1] "mg/L" "min"
## Converted:
## [1] -0.002273308
## [1] "mg/sec/kg"
```

A "fuzzy" string matching algorithm is used to automatically recognise variations in base units, allowing natural, intuitive input of units. For example, `"ml/s"`, `"mL/sec"`, `"milliliter/s"`, and `"millilitre/second"` are all equally identified as `mL/s`. Unit delimiters can be any combination of a space, dot (`.`), forward-slash (`/`), or the "per" unit (`-1`). Thus, `"ml/kg"`, `"mL / kg"`, `"mL /kilogram"`, `"ml kg-1"` or `"ml.kg-1"` are equally recognised as `mL/kg`. For a reminder on what units are available to use, call `unit_args()`:

```
unit_args()
```

```
## Note: A string-matching algorithm is used to identify units.
## E.g. all of these are the same: mg/L; mg/l, mg L-1, mgL-1, mg per litre, mg.l-1, mg.L-1
##
## O2 Units - Do not require t, S and P
## [1] "mg/L"   "ug/L"   "mmol/L" "umol/L"
##
## O2 Units - Require t, S and P
##  [1] "mL/L"    "mg/kg"   "ug/kg"   "mmol/kg" "umol/kg" "mL/kg"   "%"
##  [8] "Torr"    "hPa"     "kPa"     "mmHg"    "inHg"
##
## Time units
## [1] "s" "m" "h"
##
## Output mass units
## [1] "ug" "mg" "g"  "kg"
```

5.7 Summary

This is an example of a straightforward analysis of a *closed-chamber* respirometry experiment. This entire analysis can be documented and shared in only a few lines of code, making it entirely reproducible if the original data file is included:

```
# # import and inspect
# urchin <- inspect(urchins.rd, time = 1, oxygen = 15)
#
# # Background
# bg <- calc_rate.bg(urchins.rd, xcol = 1, ycol = 18:19, from = 5, to = 40, by = "time")
#
# # Specimen rate
# rate <- calc_rate(urchin, from = 4, to = 29, by = "time")
#
# # Adjust rate
# a.rate <- adjust_rate(rate, bg)
#
# # Convert to final rate units
# urchin_MO2 <- convert_rate(a.rate,
#                            o2.unit = "mgl-1",
#                            time.unit = "m",
#                            output.unit = "mg/s/kg",
#                            volume = 1.09,
#                            mass = 0.19)
#
# # Alternatively, use dplyr pipes, adding a print() between functions:
#
# urchins.rd %>%                                          # With the data object,
#   inspect(1, 15) %>%                                    # inspect, then
#   calc_rate(from = 4, to = 29, by = "time") %>%         # calculate rate, then
#   print() %>%
#   adjust_rate(
#     calc_rate.bg(urchins.rd, xcol = 1, ycol = 18:19,    # adjust bg rate, then
#       from = 5, to = 40, by = "time")) %>%
#   print() %>%
#   convert_rate(o2.unit = "mgl-1", time.unit = "m",
#     output.unit = "mg/s/kg", volume = 1.09, mass = 0.19) # convert units.
```

Chapter 6

Using `auto_rate()`

-> -> -> ->
->
->
-> -> -> ->
->
->
-> -> ->

Chapter 7

Performance in detecting linear regions

Chapter 8

Comparative performance of `auto_rate()` and `LoLinR`

->
->
-> ->
->
->
-> ->

Chapter 9

Intermittent-flow respirometry: Simple example

Chapter 10

Intermittent-flow respirometry: Complex example

Chapter 11

Flowthrough respirometry

Chapter 12

P_{crit}

Chapter 13

Two-point analyses

Chapter 14

Open science and reproducibility using respR

Chapter 15

respR and the tidyverse

15.1 Working in the Tidyverse

Chapter 16

A comparison of respR with other R packages

Chapter 17

When to use respR

–> –>

–> –> –>

–>

–>