Online Analytical Chemistry notes: data manipulation

Sasha D. Hafner

10:38 29 February, 2024

Contents

Overview **1**

Data types **1**

Software for data analysis **2**

General steps **2**

Operations and tools **3**
- Data frames: the fundamental data object . . . 3

Data checking and visualization **4**
- Summaries . . . 4
- Simple plots . . . 7
- New variables (adding columns) . . . 13
- Subsetting . . . 15
- Merging . . . 16
- Interpolation . . . 19
- Integration . . . 23
- Grouped operations . . . 25
- Dates and times . . . 28
- Reshaping . . . 32
- Logs, reports, and exported data . . . 33

Overview

These notes are on the steps needed to get measurement data ready for statistical modeling, or more generally, data analysis. R and Python are used to demonstrate the basic operations commonly used.

Data types

First, let's discuss data a bit. A central feature of data from "online" measurements is *repetition*. Typically we have *multiple measurements on individual experimental units*. This has implications for data processing and analysis. Data that include multiple measurements on individual experimental units may be called different things. These names tend to be associated with particular research fields and purposes. Here we will discuss some, simply to better understand what is meant when these terms are used.

Time series data

"Time series" is typically used to describe repeated measurements of a single variable at a fixed frequency, for example monthly air temperature in Aarhus. Another common example would be economic data, e.g., monthly median price of all houses sold. Typically there is some seasonal component in time series data, and perhaps an underlying trend as well, and an objective of data analysis is to separate and quantify these.

Longitudinal data

"Longitudinal data" describes repeated measurements made on multiple subjects over time. I deliberately used the term "subjects" because "longitudinal data" or "longitudinal studies" are terms commonly applied in medical or epidemiology research, where each subject is a human.

Repeated measures

The term "repeated measures" is usually used for measurements made on the same experimental units at different times, typically under different conditions or after different treatments. An example study could include 10 people, each given 3 different blood pressure medicines, with blood pressure measured 30 times in total. "Repeated measures" is also used to refer to a type of statistical method used for analyzing such data: "repeated measures ANOVA".

Online measurements

My understanding of the term "online measurement" is that some variable is measured repeatedly and automatically, perhaps nearly in real-time.

Software for data analysis

Data analysis could be done using either spreadsheet programs like Microsoft Excel or programming languages like R or Python. For various reasons spreadsheets are a bad choice for all but the simplest cases. In this course I will work with R and Python. You should use one of these, and if you want my opinion, based on more than a decade of R use and maybe a year of Python, data manipulation and analysis is much easier in R. If you want, you could probably get through with Matlab, Octave, or simlar software, but I cannot provide much support. You probably cannot successfully complete this course with only Excel or another spreadsheet program. For more information on limits of spreadsheets and advantages of script-based software see the CCPDA guide (also under reading materials through Brightspace site).

Both R and Python are open-source and extensible and there are many add-on packages (the term for R) or modules (for Python) available. For better or worse, this means there are different ways to carry out even basic operations. This siutation has the potential to create a lot of confusion for new users and conflict when it comes to collaboration. Here I have made some choices about which approaches to show, and I guess I should apologize because I haven't made a major effort to include all the different approaches or to even try to reflect what is most popular. For example, I will use the data.table package in R quite a bit in course material. I'll try to show how to do the same thing with "base" R. In Python, I'll use data frames from the pandas package. I don't think there is an alternative.

General steps

I think you are taking this course because you want to understand better how to go from online measurements to some kind of result, such as an insight into how some process works or an estimate of the effect of some treatment. Getting there requires **data analysis**, but data analysis is typically the final step, and much more time and effort is usually spent getting data ready. We can divide these preparation tasks into three steps carried out before any proper "data analysis" is done:

1. Data collection and data entry

2. Data manipulation
3. Data checking and visualization

Here I will summarize these steps and then we will jump into the most important opertaions and tools.

Data collection and data entry

With online instruments data collection is typically automated. At some point there must be manual interaction to set up automatic export of measurement data or to extract relevant results. This may be done with all sorts of software tools including programs that are provided along with the instruments. For example, I have recently learned that PTR-MS results may be saved in a format called HDF5 (for hierarchical data format, version 5), which requires some data extraction steps prior to any of the work we'll cover here. I won't cover these steps.

Even with online measurements some manual data entry may be required, e.g., the values for some variables that were manually manipulated or the time of some intervention. Spreadsheets are convenient for this type of data entry.

Data manipulation

I like to use "data processing" for the steps taken to get "raw" data to some kind of measurements. This might include application of a calibration curve, for example. In contrast "data manipulation" is used here for handling the resulting measurements. The distinction is arbitrary and unimportant; I only describe it because many examples online completely ignore any kind of "data processing" and often treat measurement data as static, which is not exactly appropriate for this course. Anyway, it is the same software tools and operations that are used in both, and we won't typically distinguish between them here.

Data checking and visulalization

This set of operations should be carried out at multiple stages.

Operations and tools

This is the main part of these notes. We'll go through the most important fundamental data manipulation operations and actual tools in R and Python. Let's start with the typical data object we use in both computing environments.

Data frames: the fundamental data object

The R and Python analog of a spreadsheet worksheet with data is a *data frame*. In Matlab these are called *tables*.

Here is one in R:

```r
dat <- read.csv('../data/slurry_emis_small.csv')
dat
```

```
##   reactor     ch4   co2 day gas temp    flow
## 1      R1  11.374 338.3   5 co2   20 0.08200
## 2      R1  45.500 230.0  18 co2   20 0.08400
## 3      R1  22.170 210.0  32 co2   20 0.07400
## 4      R5  16.000 371.5   5 co2   30 0.07475
## 5      R5 124.800 440.0  18 co2   30 0.06900
## 6      R5  81.290 415.0  32 co2   30 0.07360
```

Important characteristics are:

- Multiple rows and columns

- Each column can have a different type of data
- Each column has a name
- Data are ordered in both dimensions

If you are used to working in spreadsheets instead of R or Python, the idea of working using symbolic variables like `dat` to represent (and work with) an entire dataset may seem strange. Try to become comfortable with the concept–it is much more efficient than dealing with individual cells in a spreadsheet.

Note that while rows and columns are ordered, the exact *order* itself is typically not important. You should get in the habit of referring to columns by name and not position.

Data checking and visualization

Summaries

It is important to check data for mistakes that occurred before or during data analysis. One way to do this is by looking at data frame summaries. In R there is a `summary()` function that does this.

```
voc <- read.csv('../data/VOC_reaction.csv', skip = 2)
head(voc)
```

```
##       time_string time_number  C1H3O2  C3H7O1  C2H5O2    C7H11O2    C10H17  C9H15O1     C8H15O2  C9H1
## 1 12/7/2023 10:34    45267.44 0.26983 0.23759 0.19983 0.00494390 0.0046413 0.032477  0.00222460 0.017
## 2 12/7/2023 10:34    45267.44 0.26303 0.25205 0.18137 0.00074215 0.0012158 0.035656  0.00027488 0.017
## 3 12/7/2023 10:34    45267.44 0.27097 0.22796 0.19361 0.00399780 0.0021266 0.038066 -0.00016937 0.019
## 4 12/7/2023 10:34    45267.44 0.24479 0.19712 0.17835 0.00546340 0.0026745 0.038752  0.00192130 0.024
## 5 12/7/2023 10:34    45267.44 0.28258 0.23840 0.18143 0.00438240 0.0055961 0.036147 -0.00021588 0.016
## 6 12/7/2023 10:34    45267.44 0.18797 0.22651 0.18668 0.00426480 0.0053425 0.032794 -0.00059051 0.017
##      C10H17O4
## 1  0.00101170
## 2 -0.00087358
## 3 -0.00040130
## 4  0.00076622
## 5  0.00073954
## 6 -0.00051198
```

```
summary(voc)
```

```
##  time_string         time_number        C1H3O2             C3H7O1             C2H5O2            C7H1
##  Length:12737       Min.   :45267   Min.   :-0.04658   Min.   :-0.02714   Min.   :-0.02579   Min.
##  Class :character   1st Qu.:45268   1st Qu.: 0.03517   1st Qu.: 0.02615   1st Qu.: 0.01155   1st Qu.
##  Mode  :character   Median :45268   Median : 0.92146   Median : 1.06835   Median : 0.30210   Median
##                     Mean   :45268   Mean   : 0.71118   Mean   : 1.18624   Mean   : 0.27231   Mean
##                     3rd Qu.:45268   3rd Qu.: 1.22900   3rd Qu.: 2.07972   3rd Qu.: 0.41262   3rd Qu.
##                     Max.   :45268   Max.   : 1.64360   Max.   : 6.18590   Max.   : 4.77970   Max.
##                                     NA's   :189        NA's   :189        NA's   :189        NA's
##     C8H15O2            C9H15O2            C8H13O3            C9H15O3            C8H13O4            C9
##  Min.   :-0.00333   Min.   :-0.00361   Min.   :-0.00309   Min.   :-0.00259   Min.   :-0.00211   Min.
##  1st Qu.: 0.00136   1st Qu.: 0.00679   1st Qu.: 0.00104   1st Qu.: 0.00117   1st Qu.: 0.00038   1st Q
##  Median : 0.01112   Median : 1.11245   Median : 0.02538   Median : 0.04290   Median : 0.00423   Media
##  Mean   : 0.00983   Mean   : 0.92017   Mean   : 0.02044   Mean   : 0.03942   Mean   : 0.00694   Mean
##  3rd Qu.: 0.01758   3rd Qu.: 1.78940   3rd Qu.: 0.03766   3rd Qu.: 0.07405   3rd Qu.: 0.01384   3rd Q
##  Max.   : 0.10247   Max.   : 1.90220   Max.   : 0.05152   Max.   : 0.09259   Max.   : 0.02475   Max.
##  NA's   :189        NA's   :189        NA's   :189        NA's   :189        NA's   :189        NA's
```

```
library(data.table)
voc <- fread('../data/VOC_reaction.csv', skip = 2)
```

```r
voc
```

```
##           time_string time_number     C1H3O2     C3H7O1     C2H5O2    C7H11O2       C10H17   C9H15O1
##     1: 12/7/2023 10:34    45267.44 0.26983000 0.23759000 0.19983000 0.00494390 4.641300e-03 0.0324770
##     2: 12/7/2023 10:34    45267.44 0.26303000 0.25205000 0.18137000 0.00074215 1.215800e-03 0.0356560
##     3: 12/7/2023 10:34    45267.44 0.27097000 0.22796000 0.19361000 0.00399780 2.126600e-03 0.0380660
##     4: 12/7/2023 10:34    45267.44 0.24479000 0.19712000 0.17835000 0.00546340 2.674500e-03 0.0387520
##     5: 12/7/2023 10:34    45267.44 0.28258000 0.23840000 0.18143000 0.00438240 5.596100e-03 0.0361470
##    ---
## 12733: 12/7/2023 17:38    45267.73 0.03405808 0.02747731 0.01098923 0.01535654 1.477731e-03 0.1282731
## 12734: 12/7/2023 17:38    45267.73 0.03853077 0.02504192 0.01067731 0.01657769 6.119231e-04 0.1302038
## 12735: 12/7/2023 17:38    45267.73 0.03404269 0.02224269 0.01141692 0.01477038 1.053115e-03 0.1296077
## 12736: 12/7/2023 17:38    45267.73 0.03497692 0.02501538 0.01088462 0.01620885 8.473462e-04 0.1285192
## 12737: 12/7/2023 17:38    45267.73 0.03452077 0.02439692 0.01039846 0.01530308 1.123654e-05 0.1264692
##             C8H13O4      C9H15O4      C10H17O4
##     1: -0.000329700  0.000333540  0.0010117000
##     2:  0.000026400  0.000302220 -0.0008735800
##     3:  0.000372220  0.000087300 -0.0004013000
##     4: -0.000260980 -0.000734150  0.0007662200
##     5: -0.000741000  0.000602880  0.0007395400
##    ---
## 12733:  0.001797462  0.001629615  0.0013644231
## 12734:  0.002281692  0.001849769  0.0009983462
## 12735:  0.001946538  0.001774385  0.0013291923
## 12736:  0.002222885  0.001807192  0.0008379615
## 12737:  0.001943769  0.002313885  0.0008159615
```

```r
summary(voc)
```

```
##  time_string         time_number       C1H3O2            C3H7O1            C2H5O2            C7H1
##  Length:12737       Min.   :45267   Min.   :-0.04658   Min.   :-0.02714   Min.   :-0.02579   Min.
##  Class :character   1st Qu.:45268   1st Qu.: 0.03517   1st Qu.: 0.02615   1st Qu.: 0.01155   1st Qu.
##  Mode  :character   Median :45268   Median : 0.92146   Median : 1.06835   Median : 0.30210   Median
##                     Mean   :45268   Mean   : 0.71118   Mean   : 1.18624   Mean   : 0.27231   Mean
##                     3rd Qu.:45268   3rd Qu.: 1.22900   3rd Qu.: 2.07972   3rd Qu.: 0.41262   3rd Qu.
##                     Max.   :45268   Max.   : 1.64360   Max.   : 6.18590   Max.   : 4.77970   Max.
##                                     NA's   :189        NA's   :189        NA's   :189        NA's
##     C8H15O2           C9H15O2           C8H13O3           C9H15O3           C8H13O4           C9
##  Min.   :-0.00333   Min.   :-0.00361   Min.   :-0.00309   Min.   :-0.00259   Min.   :-0.00211   Min.
##  1st Qu.: 0.00136   1st Qu.: 0.00679   1st Qu.: 0.00104   1st Qu.: 0.00117   1st Qu.: 0.00038   1st
##  Median : 0.01112   Median : 1.11245   Median : 0.02538   Median : 0.04290   Median : 0.00423   Media
##  Mean   : 0.00983   Mean   : 0.92017   Mean   : 0.02044   Mean   : 0.03942   Mean   : 0.00694   Mean
##  3rd Qu.: 0.01758   3rd Qu.: 1.78940   3rd Qu.: 0.03766   3rd Qu.: 0.07405   3rd Qu.: 0.01384   3rd
##  Max.   : 0.10247   Max.   : 1.90220   Max.   : 0.05152   Max.   : 0.09259   Max.   : 0.02475   Max.
##  NA's   :189        NA's   :189        NA's   :189        NA's   :189        NA's   :189        NA's
```

It can tell us if there is a problem with missing values or gross mistakes in values, e.g., large negative concentration values. Here we can see at least one small negative value in the concentration of the compound of interest in these data, in the `C10H17` column. The `dfsumm()` function does a bit more.

```r
source('../R-functions/dfsumm.R')
dfsumm(voc)
```

```
##
##  12737 rows and 15 columns
##  12556 unique rows
##                        time_string time_number  C1H3O2  C3H7O1  C2H5O2  C7H11O2   C10H17  C9H15O1 C8
## Class                    character     numeric numeric numeric numeric  numeric  numeric  numeric nu
## Minimum            12/7/2023 10:34       45300 -0.0466 -0.0271 -0.0258 -0.00457 -0.00264 -0.00357 -0
## Maximum            12/7/2023 17:38       45300    1.64    6.19    4.78    0.154     17.6     6.09
## Mean                          <NA>       45300   0.711    1.19   0.272   0.0668      3.7     2.68 0
## Unique (excld. NA)             425          76    9084   11454   11215     9363    12213    11293
## Missing values                   0           0     189     189     189      189      189      189
## Sorted                        TRUE        TRUE   FALSE   FALSE   FALSE    FALSE    FALSE    FALSE
##
##                    C10H17O4
## Class               numeric
## Minimum            -0.00554
## Maximum               0.012
## Mean                0.00279
## Unique (excld. NA)    12142
## Missing values          189
## Sorted                FALSE
##
```

We might think about:

- Is the size correct?
- Do we expect any missing values?
- Do we see unique values where expected?
- Are the column types right?

Other R functions that are helpful include:

- `dim()`
- `unique()`
- `length()`

And for summary statistics, try these functions:

- `min()` and `max()`
- `range()`
- `mean()`
- `sd()`
- `quantile()`

Try them.

In Python, we can use the `describe()` function.

```
import pandas as pd

voc = pd.read_csv('../data/VOC_reaction.csv', skiprows = 2)
print(voc)
```

```
##            time_string  time_number    C1H3O2    C3H7O1    C2H5O2   C7H11O2    C10H17  ...    C8H15O2
## 0      12/7/2023 10:34   45267.4414  0.269830  0.237590  0.199830  0.004944  0.004641  ...   0.002225
## 1      12/7/2023 10:34   45267.4414  0.263030  0.252050  0.181370  0.000742  0.001216  ...   0.000275
## 2      12/7/2023 10:34   45267.4414  0.270970  0.227960  0.193610  0.003998  0.002127  ...  -0.000169
## 3      12/7/2023 10:34   45267.4414  0.244790  0.197120  0.178350  0.005463  0.002675  ...   0.001921
## 4      12/7/2023 10:34   45267.4414  0.282580  0.238400  0.181430  0.004382  0.005596  ...  -0.000216
## ...                ...          ...       ...       ...       ...       ...       ...  ...        ...
## 12732  12/7/2023 17:38   45267.7344  0.034058  0.027477  0.010989  0.015357  0.001478  ...   0.002592
## 12733  12/7/2023 17:38   45267.7344  0.038531  0.025042  0.010677  0.016578  0.000612  ...   0.002446
## 12734  12/7/2023 17:38   45267.7344  0.034043  0.022243  0.011417  0.014770  0.001053  ...   0.002467
## 12735  12/7/2023 17:38   45267.7344  0.034977  0.025015  0.010885  0.016209  0.000847  ...  0.002529
## 12736  12/7/2023 17:38   45267.7344  0.034521  0.024397  0.010398  0.015303  0.000011  ...  0.002808
##
## [12737 rows x 15 columns]
```

```
print(voc.describe())
```

```
##         time_number        C1H302        C3H701        C2H502        C7H1102  ...       C8H1303
## count  12737.000000  12548.000000  12548.000000  12548.000000  12548.000000  ...  12548.000000  1254
## mean   45267.587726      0.711176      1.186239      0.272310      0.066836  ...      0.020440      0
## std        0.085145      0.535461      1.005795      0.190732      0.057731  ...      0.017790      0
## min    45267.441400     -0.046582     -0.027145     -0.025795     -0.004575  ...     -0.003087     -0
## 25%    45267.515600      0.035175      0.026150      0.011551      0.003132  ...      0.001042      0
## 50%    45267.585900      0.921455      1.068350      0.302095      0.079514  ...      0.025378      0
## 75%    45267.660200      1.229000      2.079725      0.412623      0.124940  ...      0.037664      0
## max    45267.734400      1.643600      6.185900      4.779700      0.154180  ...      0.051525      0
##
## [8 rows x 14 columns]
```

It can be helpful to turn it sideways.

```
print(voc.describe().transpose())
```

```
##                count          mean       std           min           25%           50%           75%
## time_number  12737.0  45267.587726  0.085145  45267.441400  45267.515600  45267.585900  45267.660200
## C1H302       12548.0      0.711176  0.535461     -0.046582      0.035175      0.921455      1.229000
## C3H701       12548.0      1.186239  1.005795     -0.027145      0.026150      1.068350      2.079725
## C2H502       12548.0      0.272310  0.190732     -0.025795      0.011551      0.302095      0.412623
## C7H1102      12548.0      0.066836  0.057731     -0.004575      0.003132      0.079514      0.124940
## C10H17       12548.0      3.696581  4.411238     -0.002645      0.001202      2.428200      5.783100
## C9H1501      12548.0      2.682968  2.489005     -0.003570      0.024107      2.842500      5.298050
## C8H1502      12548.0      0.009833  0.008442     -0.003327      0.001360      0.011121      0.017579
## C9H1502      12548.0      0.920174  0.824448     -0.003615      0.006790      1.112450      1.789400
## C8H1303      12548.0      0.020440  0.017790     -0.003087      0.001042      0.025378      0.037664
## C9H1503      12548.0      0.039420  0.033893     -0.002587      0.001165      0.042897      0.074049
## C8H1304      12548.0      0.006944  0.006970     -0.002106      0.000377      0.004230      0.013842
## C9H1504      12548.0      0.004631  0.004636     -0.002358      0.000377      0.003088      0.008722
## C10H1704     12548.0      0.002787  0.002814     -0.005539      0.000358      0.002078      0.005239
```

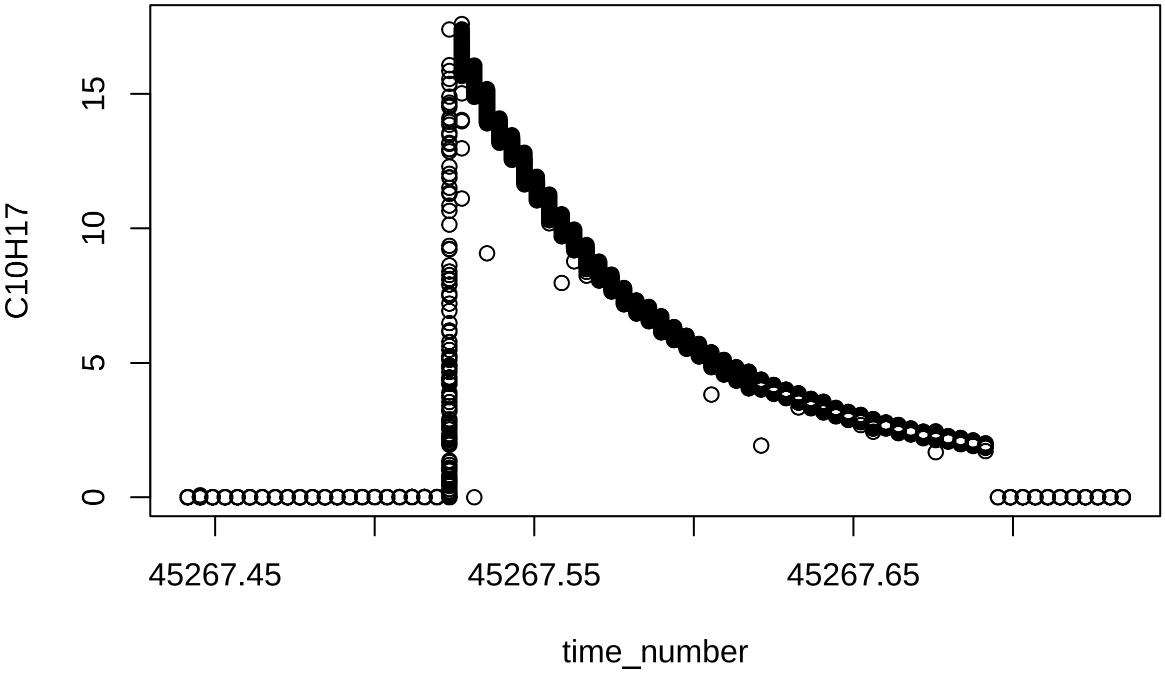
Simple plots

Always plot your data. No kind of numerical summary or anything else compares to visualization of data. There are a lot of different options for generating plots. Here let's look at some simple approaches for checking data (not producing publication- or presentation-ready graphics).

```
head(voc)
```

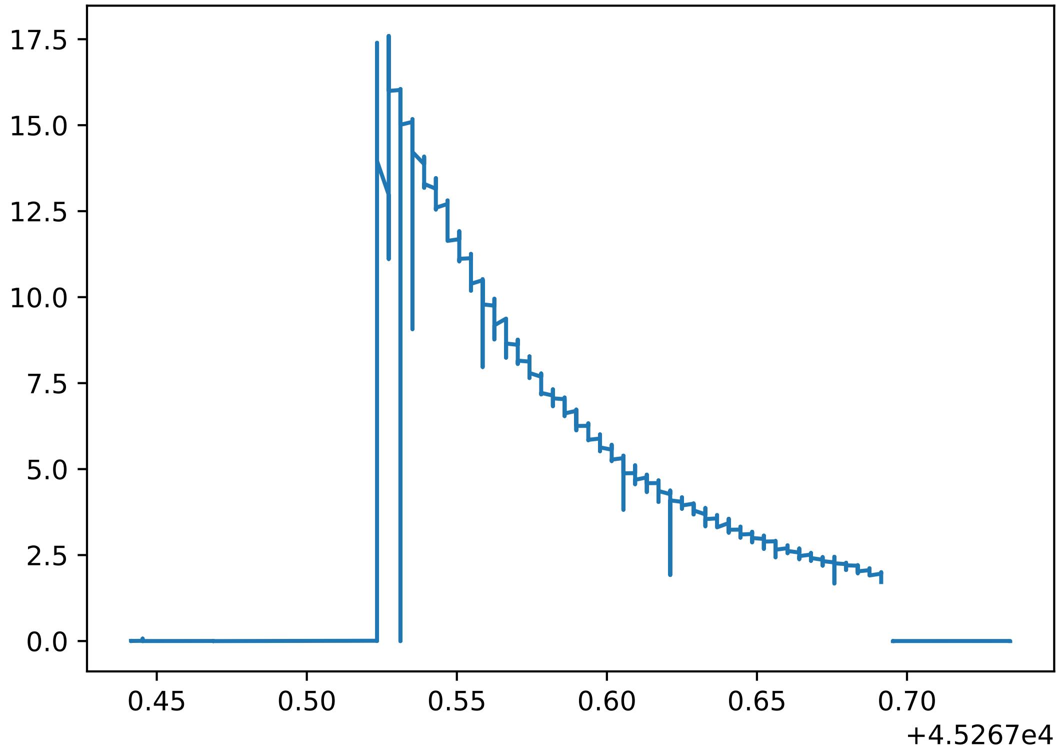
```
##        time_string time_number  C1H302  C3H701  C2H502    C7H1102    C10H17  C9H1501     C8H1502  C9
## 1: 12/7/2023 10:34    45267.44 0.26983 0.23759 0.19983 0.00494390 0.0046413 0.032477  0.00222460 0.0
## 2: 12/7/2023 10:34    45267.44 0.26303 0.25205 0.18137 0.00074215 0.0012158 0.035656  0.00027488 0.0
## 3: 12/7/2023 10:34    45267.44 0.27097 0.22796 0.19361 0.00399780 0.0021266 0.038066 -0.00016937 0.0
## 4: 12/7/2023 10:34    45267.44 0.24479 0.19712 0.17835 0.00546340 0.0026745 0.038752  0.00192130 0.0
## 5: 12/7/2023 10:34    45267.44 0.28258 0.23840 0.18143 0.00438240 0.0055961 0.036147 -0.00021588 0.0
## 6: 12/7/2023 10:34    45267.44 0.18797 0.22651 0.18668 0.00426480 0.0053425 0.032794 -0.00059051 0.0
##       C10H1704
## 1:  0.00101170
## 2: -0.00087358
## 3: -0.00040130
## 4:  0.00076622
## 5:  0.00073954
## 6: -0.00051198
```

```
plot(C10H17 ~ time_number, data = voc)
```



This shows a lot!

```
import matplotlib.pyplot as plt
plt.plot(voc['time_number'], voc['C10H17'])
plt.show()
```



For grouped data, the ggplot2 package in R can be efficient.

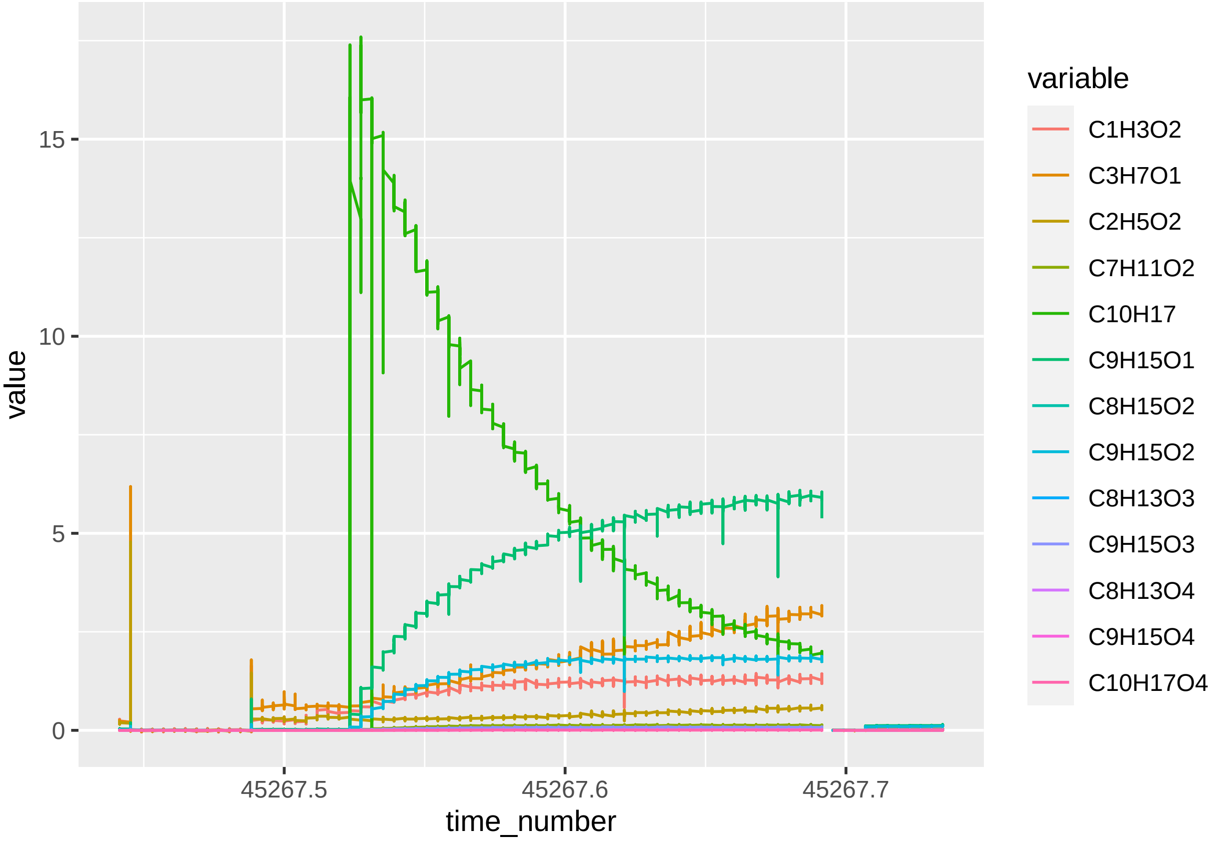
```
head(voc)
```

```
##        time_string time_number  C1H3O2  C3H7O1  C2H5O2    C7H11O2    C10H17  C9H15O1     C8H15O2  C9
## 1: 12/7/2023 10:34    45267.44 0.26983 0.23759 0.19983 0.00494390 0.0046413 0.032477  0.00222460 0.0
## 2: 12/7/2023 10:34    45267.44 0.26303 0.25205 0.18137 0.00074215 0.0012158 0.035656  0.00027488 0.0
## 3: 12/7/2023 10:34    45267.44 0.27097 0.22796 0.19361 0.00399780 0.0021266 0.038066 -0.00016937 0.0
## 4: 12/7/2023 10:34    45267.44 0.24479 0.19712 0.17835 0.00546340 0.0026745 0.038752  0.00192130 0.0
## 5: 12/7/2023 10:34    45267.44 0.28258 0.23840 0.18143 0.00438240 0.0055961 0.036147 -0.00021588 0.0
## 6: 12/7/2023 10:34    45267.44 0.18797 0.22651 0.18668 0.00426480 0.0053425 0.032794 -0.00059051 0.0
##       C10H17O4
## 1:  0.00101170
## 2: -0.00087358
## 3: -0.00040130
## 4:  0.00076622
## 5:  0.00073954
## 6: -0.00051198
```

```
vocl <- melt(voc, id.vars = c('time_string', 'time_number'))
vocl
```

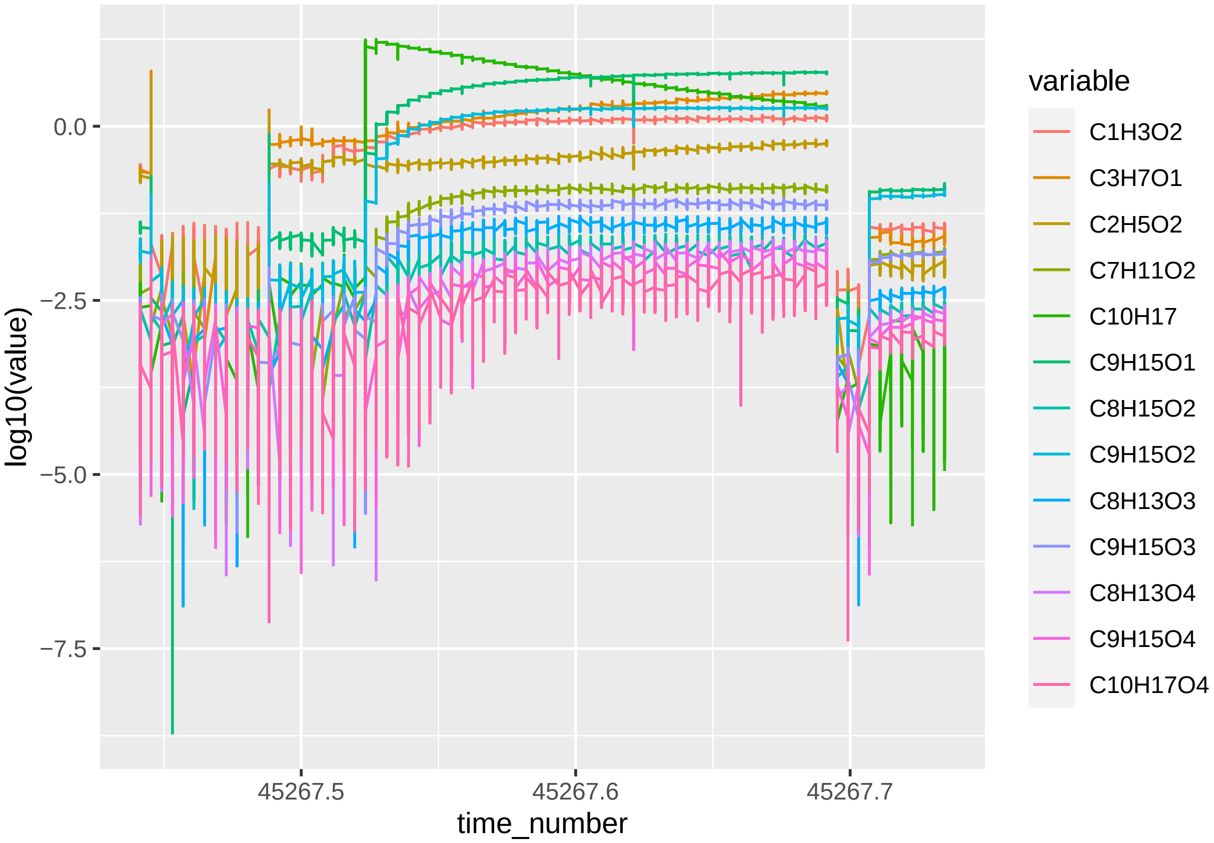
```
##             time_string time_number variable        value
##      1: 12/7/2023 10:34    45267.44   C1H3O2 0.2698300000
##      2: 12/7/2023 10:34    45267.44   C1H3O2 0.2630300000
##      3: 12/7/2023 10:34    45267.44   C1H3O2 0.2709700000
##      4: 12/7/2023 10:34    45267.44   C1H3O2 0.2447900000
##      5: 12/7/2023 10:34    45267.44   C1H3O2 0.2825800000
##     ---
## 165577: 12/7/2023 17:38    45267.73 C10H17O4 0.0013644231
## 165578: 12/7/2023 17:38    45267.73 C10H17O4 0.0009983462
## 165579: 12/7/2023 17:38    45267.73 C10H17O4 0.0013291923
## 165580: 12/7/2023 17:38    45267.73 C10H17O4 0.0008379615
## 165581: 12/7/2023 17:38    45267.73 C10H17O4 0.0008159615
```

```
library(ggplot2)
ggplot(vocl, aes(time_number, value, colour = variable)) +
  geom_line()
```



```
library(ggplot2)
ggplot(vocl, aes(time_number, log10(value), colour = variable)) +
  geom_line()
```

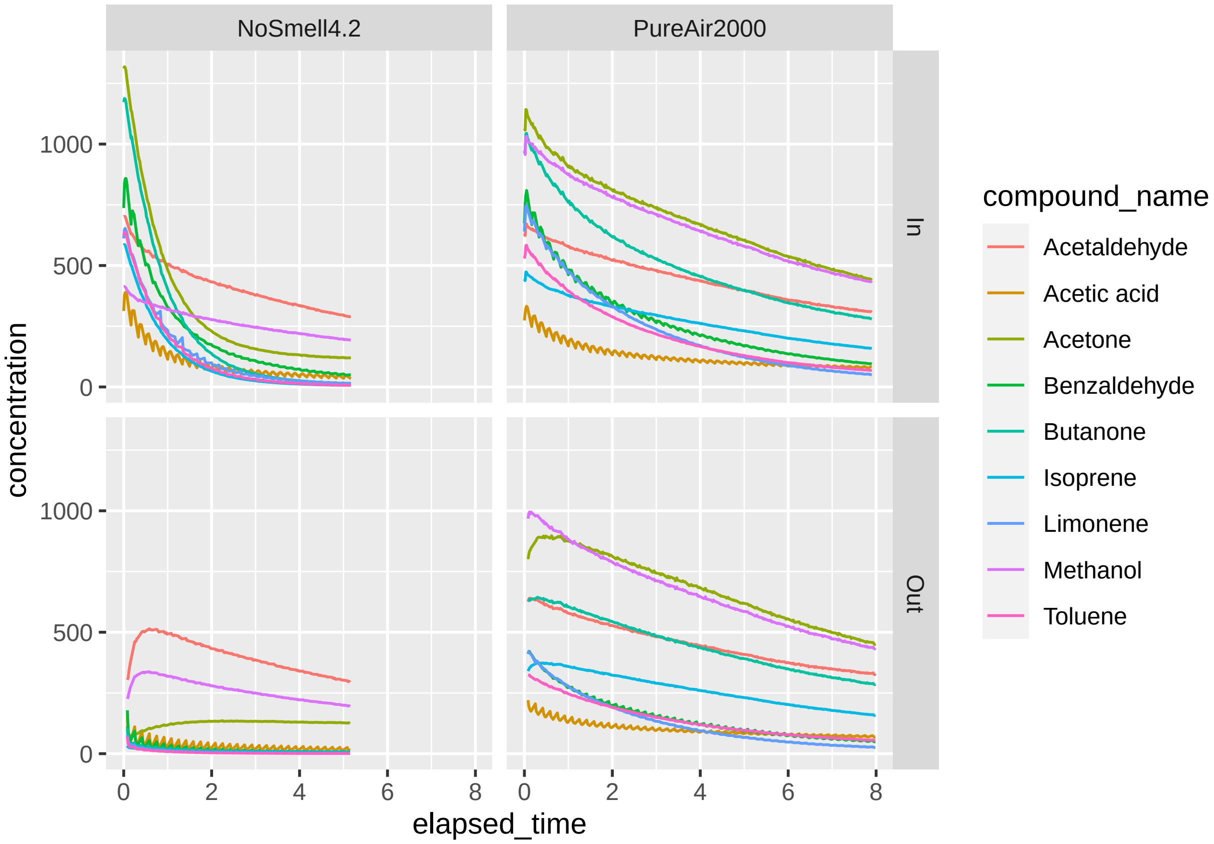
```
## Warning in FUN(X[[i]], ...): NaNs produced

## Warning in FUN(X[[i]], ...): NaNs produced

## Warning: Removed 1 row containing missing values (`geom_line()`).
```



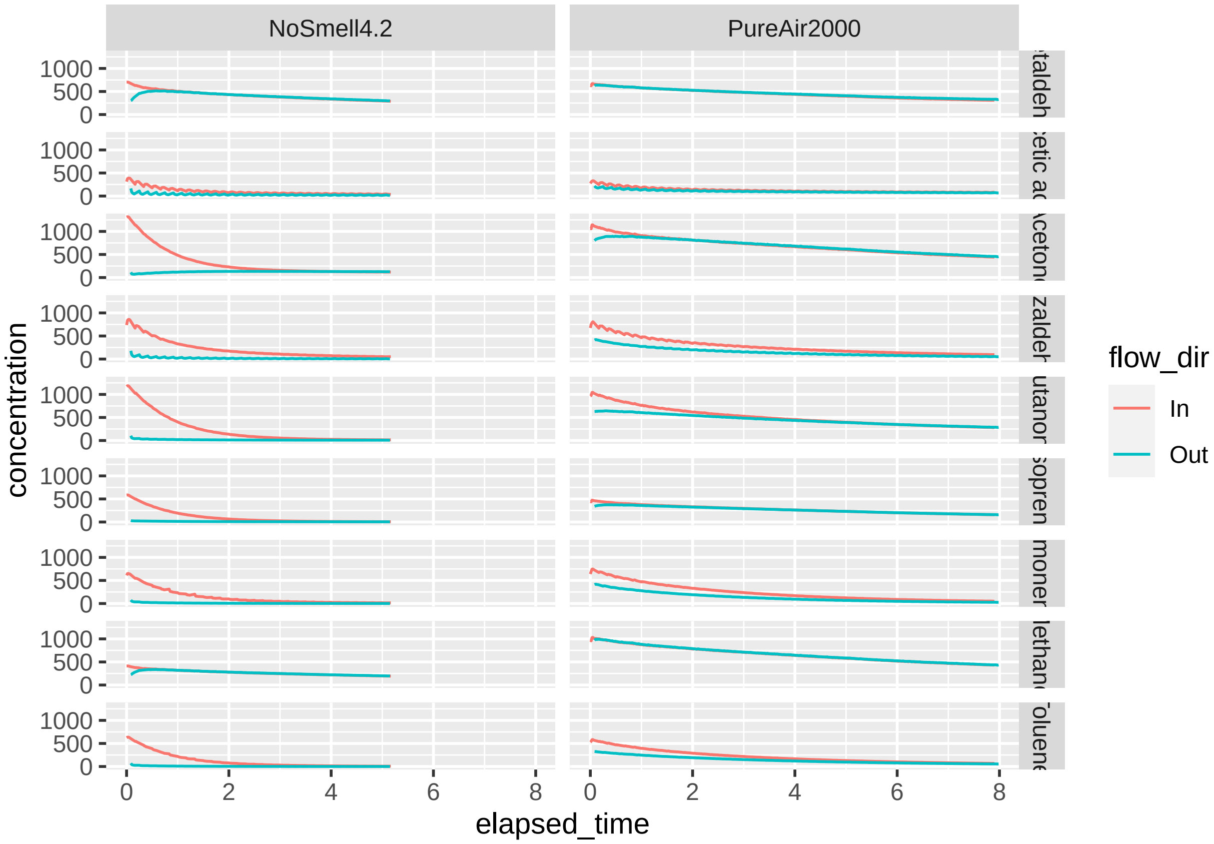
```r
air <- fread('../data/air_cleaners.csv')
air
```

```
##        aircleaner       timestamp elapsed_time   form     mtzr compound_name flow_dir concentration
##    1: PureAir2000 3/10/2022 13:45     0.000000   CH4O  32.0335      Methanol       In    971.682900
##    2: PureAir2000 3/10/2022 13:45     0.000000   C7H8  92.0699       Toluene       In    528.657950
##    3: PureAir2000 3/10/2022 13:45     0.000000   C5H8  68.0699      Isoprene       In    443.841800
##    4: PureAir2000 3/10/2022 13:45     0.000000  C7H6O 106.0491  Benzaldehyde       In    673.093200
##    5: PureAir2000 3/10/2022 13:45     0.000000  C3H6O  58.0491       Acetone       In   1063.301000
##   ---
## 7115:  NoSmell4.2 3/30/2022 18:05     5.166667   CH4O  32.0335      Methanol       In    193.325450
## 7116:  NoSmell4.2 3/30/2022 18:05     5.166667  C7H6O 106.0491  Benzaldehyde       In     50.615150
## 7117:  NoSmell4.2 3/30/2022 18:05     5.166667  C3H6O  58.0491       Acetone       In    120.169100
## 7118:  NoSmell4.2 3/30/2022 18:05     5.166667 C2H4O2  60.0284   Acetic acid       In     33.904700
## 7119:  NoSmell4.2 3/30/2022 18:05     5.166667   C7H8  92.0699       Toluene       In      7.061408
```

```r
ggplot(air, aes(elapsed_time, concentration, colour = compound_name)) +
  geom_line() +
  facet_grid(flow_dir ~ aircleaner)
```



```
ggplot(air, aes(elapsed_time, concentration, colour = flow_dir)) +
  geom_line() +
  facet_grid(compound_name ~ aircleaner)
```



New variables (adding columns)

Data processing typically requires the calculation of new variables. For example, to calculate the rate of methane production within a bottle from measured methane concentration and gas flow rate, we would multiply the two.

First, in R. For better or worse, there are a lot of different ways to do this. I'll start with some older approaches, which you can ignore or forget if you like.

```r
library(data.table)
dat <- fread('../data/slurry_emis_small.csv')
dat
```

```
##    reactor     ch4   co2 day gas temp    flow
## 1:      R1  11.374 338.3   5 co2   20 0.08200
## 2:      R1  45.500 230.0  18 co2   20 0.08400
## 3:      R1  22.170 210.0  32 co2   20 0.07400
## 4:      R5  16.000 371.5   5 co2   30 0.07475
## 5:      R5 124.800 440.0  18 co2   30 0.06900
## 6:      R5  81.290 415.0  32 co2   30 0.07360
```

```r
names(dat)
```

```
## [1] "reactor" "ch4"     "co2"     "day"     "gas"     "temp"    "flow"
```

```r
dat$qch4  <- dat$flow * dat$ch4
dat[, 'qch4.b']  <- dat[, 'flow'] * dat[, 'ch4']
dat
```

```
##    reactor     ch4   co2 day gas temp    flow     qch4   qch4.b
## 1:       R1  11.374 338.3   5 co2   20 0.08200 0.932668 0.932668
## 2:       R1  45.500 230.0  18 co2   20 0.08400 3.822000 3.822000
## 3:       R1  22.170 210.0  32 co2   20 0.07400 1.640580 1.640580
## 4:       R5  16.000 371.5   5 co2   30 0.07475 1.196000 1.196000
## 5:       R5 124.800 440.0  18 co2   30 0.06900 8.611200 8.611200
## 6:       R5  81.290 415.0  32 co2   30 0.07360 5.982944 5.982944
```

Here is a relatively new data table approach, which I have started using.

```
dat[, qch4.c := flow * ch4]
dat
```

```
##    reactor     ch4   co2 day gas temp    flow     qch4   qch4.b   qch4.c
## 1:       R1  11.374 338.3   5 co2   20 0.08200 0.932668 0.932668 0.932668
## 2:       R1  45.500 230.0  18 co2   20 0.08400 3.822000 3.822000 3.822000
## 3:       R1  22.170 210.0  32 co2   20 0.07400 1.640580 1.640580 1.640580
## 4:       R5  16.000 371.5   5 co2   30 0.07475 1.196000 1.196000 1.196000
## 5:       R5 124.800 440.0  18 co2   30 0.06900 8.611200 8.611200 8.611200
## 6:       R5  81.290 415.0  32 co2   30 0.07360 5.982944 5.982944 5.982944
```

And if you like tidyverse you can use the `mutate()` function from the dplyr package.

They all give the same result.

```
head(dat)
```

```
##    reactor     ch4   co2 day gas temp    flow     qch4   qch4.b   qch4.c
## 1:       R1  11.374 338.3   5 co2   20 0.08200 0.932668 0.932668 0.932668
## 2:       R1  45.500 230.0  18 co2   20 0.08400 3.822000 3.822000 3.822000
## 3:       R1  22.170 210.0  32 co2   20 0.07400 1.640580 1.640580 1.640580
## 4:       R5  16.000 371.5   5 co2   30 0.07475 1.196000 1.196000 1.196000
## 5:       R5 124.800 440.0  18 co2   30 0.06900 8.611200 8.611200 8.611200
## 6:       R5  81.290 415.0  32 co2   30 0.07360 5.982944 5.982944 5.982944
```

In Python.

```
dat = pd.read_csv('../data/slurry_emis_small.csv')
print(dat)
```

```
##   reactor      ch4    co2  day  gas  temp     flow
## 0      R1   11.374  338.3    5  co2    20  0.08200
## 1      R1   45.500  230.0   18  co2    20  0.08400
## 2      R1   22.170  210.0   32  co2    20  0.07400
## 3      R5   16.000  371.5    5  co2    30  0.07475
## 4      R5  124.800  440.0   18  co2    30  0.06900
## 5      R5   81.290  415.0   32  co2    30  0.07360
```

```
dat['qch4'] = dat['flow'] * dat['ch4']
print(dat)
```

```
##   reactor      ch4    co2  day  gas  temp     flow      qch4
## 0      R1   11.374  338.3    5  co2    20  0.08200  0.932668
## 1      R1   45.500  230.0   18  co2    20  0.08400  3.822000
## 2      R1   22.170  210.0   32  co2    20  0.07400  1.640580
## 3      R5   16.000  371.5    5  co2    30  0.07475  1.196000
## 4      R5  124.800  440.0   18  co2    30  0.06900  8.611200
## 5      R5   81.290  415.0   32  co2    30  0.07360  5.982944
```

And here is an alternative that uses a dot to extract columns. But it cannot be used for column creation.

```
dat['qch4b'] = dat.flow * dat.ch4
print(dat)
```

```
##    reactor      ch4    co2  day  gas  temp     flow      qch4     qch4b
## 0      R1   11.374  338.3    5  co2    20  0.08200  0.932668  0.932668
## 1      R1   45.500  230.0   18  co2    20  0.08400  3.822000  3.822000
## 2      R1   22.170  210.0   32  co2    20  0.07400  1.640580  1.640580
## 3      R5   16.000  371.5    5  co2    30  0.07475  1.196000  1.196000
## 4      R5  124.800  440.0   18  co2    30  0.06900  8.611200  8.611200
## 5      R5   81.290  415.0   32  co2    30  0.07360  5.982944  5.982944
```

Subsetting

Subsetting means *extracting* part of a dataset. Perhaps early measurements need to be excluded because sample gas had not reached the sensor. Or maybe data analysis needs to be applied separately to “before” and “after” samples, which therefore need to be separated. Here I will demonstrate it in R and Python.

First R. Let’s get the data (again, slightly differently this time).

```
library(data.table)
dat <- fread('../data/slurry_emis_small.csv')
dat
```

```
##     reactor     ch4   co2 day gas temp    flow
## 1:       R1  11.374 338.3   5 co2   20 0.08200
## 2:       R1  45.500 230.0  18 co2   20 0.08400
## 3:       R1  22.170 210.0  32 co2   20 0.07400
## 4:       R5  16.000 371.5   5 co2   30 0.07475
## 5:       R5 124.800 440.0  18 co2   30 0.06900
## 6:       R5  81.290 415.0  32 co2   30 0.07360
```

```
summary(dat)
```

```
##    reactor               ch4              co2             day              gas                 temp
##  Length:6           Min.   : 11.37   Min.   :210.0   Min.   : 5.00   Length:6           Min.   :20
##  Class :character   1st Qu.: 17.54   1st Qu.:257.1   1st Qu.: 8.25   Class :character   1st Qu.:20
##  Mode  :character   Median : 33.84   Median :354.9   Median :18.00   Mode  :character   Median :25
##                     Mean   : 50.19   Mean   :334.1   Mean   :18.33                      Mean   :25
##                     3rd Qu.: 72.34   3rd Qu.:404.1   3rd Qu.:28.50                      3rd Qu.:30
##                     Max.   :124.80   Max.   :440.0   Max.   :32.00                      Max.   :30
```

If we want only measurements made between 5 and 30 days:

```
dat
```

```
##     reactor     ch4   co2 day gas temp    flow
## 1:       R1  11.374 338.3   5 co2   20 0.08200
## 2:       R1  45.500 230.0  18 co2   20 0.08400
## 3:       R1  22.170 210.0  32 co2   20 0.07400
## 4:       R5  16.000 371.5   5 co2   30 0.07475
## 5:       R5 124.800 440.0  18 co2   30 0.06900
## 6:       R5  81.290 415.0  32 co2   30 0.07360
```

```
sub1 <- dat[day >= 5 & day <= 30, ]
sub1
```

```
##     reactor     ch4   co2 day gas temp    flow
## 1:       R1  11.374 338.3   5 co2   20 0.08200
## 2:      R1  45.500 230.0  18 co2   20 0.08400
## 3:      R5  16.000 371.5   5 co2   30 0.07475
## 4:      R5 124.800 440.0  18 co2   30 0.06900
```

Check the values of `gas` and `temp`.

```
table(dat[, .(gas, temp)])
```

```
##      temp
## gas   20 30
##   co2  3  3
```

We could take all observations with `gas = 'n2'` and `temp = 10` with this:

```
sub2 <- dat[gas == 'n2' & temp == 10, ]
sub2
```

```
## Empty data.table (0 rows and 7 cols): reactor,ch4,co2,day,gas,temp...
```

Python is not so different. Note that the data frame data structure only comes in an add-on package or "module" called pandas.

```
import pandas as pd

dat = pd.read_csv('../data/slurry_emis_small.csv')
print(dat)
```

```
##   reactor      ch4    co2  day  gas  temp     flow
## 0      R1   11.374  338.3    5  co2    20  0.08200
## 1      R1   45.500  230.0   18  co2    20  0.08400
## 2      R1   22.170  210.0   32  co2    20  0.07400
## 3      R5   16.000  371.5    5  co2    30  0.07475
## 4      R5  124.800  440.0   18  co2    30  0.06900
## 5      R5   81.290  415.0   32  co2    30  0.07360
```

```
sub1 = dat[(dat['day'] >= 5) & (dat['day'] <= 30)]
print(sub1)
```

```
##   reactor      ch4    co2  day  gas  temp     flow
## 0      R1   11.374  338.3    5  co2    20  0.08200
## 1      R1   45.500  230.0   18  co2    20  0.08400
## 3      R5   16.000  371.5    5  co2    30  0.07475
## 4      R5  124.800  440.0   18  co2    30  0.06900
```

```
sub2 = dat[(dat['gas'] == 'co2') & (dat['temp'] == 10)]
print(sub2)
```

```
## Empty DataFrame
## Columns: [reactor, ch4, co2, day, gas, temp, flow]
## Index: []
```

Merging

There are several different ways that data frames can be combined, thinking about both *concepts* and *functions*. A type of combining called *merging* means aligning by row using some key in R and Python. Here, for example, are some results from an experiment on ammonia volatilization from field-applied animal slurry, organized into two different files.

```
amm_int <- fread('../data/NH3_emis_acid_interval.csv')
amm_int
```

```
##       pmid     ct      cta   dt             t_start               t_end     j_NH3
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350
```

```
amm_plot <- fread('../data/NH3_emis_acid_plot.csv')
amm_plot
```

```
##     pmid treat   app_date tan_app e_cum_final e_rel_final date_int
##  1: 1947  tank 2020-11-18   97.30      3.9108    0.040193        1
##  2: 1948  tank 2020-11-18   97.30      4.9536    0.050910        1
##  3: 1949 field 2020-11-18  103.60     13.6860    0.132110        1
##  4: 1950 field 2020-11-18  103.60     12.3270    0.118980        1
##  5: 1951  none 2020-11-18   95.20     20.0020    0.210100        1
##  6: 1952 field 2020-11-18  103.60     14.6960    0.141860        1
##  7: 1953  none 2020-11-18   95.20     19.9610    0.209670        1
##  8: 1954  tank 2020-11-18   97.30      5.3328    0.054808        1
##  9: 1955  none 2020-11-18   95.20     17.1320    0.179960        1
## 10: 1956  none 2020-11-25   71.75     25.1850    0.351020        2
## 11: 1957 field 2020-11-25   72.45     26.9790    0.372390        2
## 12: 1958  tank 2020-11-25   67.55      1.3104    0.019399        2
## 13: 1959 field 2020-11-25   72.45     20.7570    0.286510        2
## 14: 1960  tank 2020-11-25   67.55      1.8739    0.027741        2
## 15: 1961  none 2020-11-25   71.75     25.3840    0.353780        2
## 16: 1962  tank 2020-11-25   67.55      2.3160    0.034286        2
## 17: 1963 field 2020-11-25   72.45     23.5660    0.325270        2
## 18: 1964  none 2020-11-25   71.75     26.8990    0.374900        2
## 19: 1965  none 2020-02-12  151.20     20.4720    0.135400        3
## 20: 1966  tank 2020-02-12  118.30      3.3581    0.028386        3
## 21: 1967 field 2020-02-12  149.10     17.5260    0.117540        3
## 22: 1968 field 2020-02-12  149.10     17.5560    0.117750        3
## 23: 1969  tank 2020-02-12  118.30      3.1914    0.026977        3
## 24: 1970 field 2020-02-12  149.10     17.2320    0.115580        3
## 25: 1971  none 2020-02-12  151.20     25.9790    0.171820        3
## 26: 1972  tank 2020-02-12  118.30      3.1087    0.026278        3
## 27: 1973  none 2020-02-12  151.20     24.6010    0.162700        3
## 28: 1974  tank 2020-09-12   71.40      8.6166    0.120680        3
## 29: 1975  tank 2020-12-09   71.40      8.8196    0.123520        4
## 30: 1976 field 2020-12-09   65.10     15.6990    0.241150        4
## 31: 1977  none 2020-09-12   66.50     17.2490    0.259380        4
## 32: 1978 field 2020-09-12   65.10     14.6140    0.224490        4
## 33: 1979  none 2020-12-09   66.50     18.9850    0.285480        4
## 34: 1980  tank 2020-12-09   71.40      9.3760    0.131320        4
## 35: 1981 field 2020-12-09   65.10     14.6650    0.225270        4
## 36: 1982  none 2020-12-09   66.50     18.4340    0.277210        4
##     pmid treat   app_date tan_app e_cum_final e_rel_final date_int
```

```
dim(amm_int)
```

```
## [1] 3489    7
```

```
dim(amm_plot)
```

```
## [1] 36  7
```

The plot-level data frame is smaller, with only a single observation for each field plot. And each field plot has a unique *key* or *ID* in the `pmid` column. We can use the key to merge.

```
amm_comb <- merge(amm_plot, amm_int, by = 'pmid')
amm_comb
```

```
##       pmid treat   app_date tan_app e_cum_final e_rel_final date_int     ct      cta   dt
##    1: 1947  tank 2020-11-18    97.3      3.9108    0.040193        1   1.73   1.7333 1.73 2020-11-18
##    2: 1947  tank 2020-11-18    97.3      3.9108    0.040193        1   3.46   3.4667 1.73 2020-11-18
##    3: 1947  tank 2020-11-18    97.3      3.9108    0.040193        1   5.19   5.2000 1.73 2020-11-18
##    4: 1947  tank 2020-11-18    97.3      3.9108    0.040193        1   6.92   6.9333 1.73 2020-11-18
##    5: 1947  tank 2020-11-18    97.3      3.9108    0.040193        1   8.65   8.6667 1.73 2020-11-18
##   ---
## 3485: 1982  none 2020-12-09    66.5     18.4340    0.277210        4 178.19 178.5300 1.73 2020-12-16
## 3486: 1982  none 2020-12-09    66.5     18.4340    0.277210        4 179.92 180.2700 1.73 2020-12-17
## 3487: 1982  none 2020-12-09    66.5     18.4340    0.277210        4 181.65 182.0000 1.73 2020-12-17
## 3488: 1982  none 2020-12-09    66.5     18.4340    0.277210        4 183.38 183.7300 1.73 2020-12-17
## 3489: 1982  none 2020-12-09    66.5     18.4340    0.277210        4 185.11 185.4700 1.73 2020-12-17
```

And now we have all the plot-level data combined with the interval-level data (and duplicated, because of the difference in data frame size).

In Python

```
amm_int = pd.read_csv('../data/NH3_emis_acid_interval.csv')
amm_plot = pd.read_csv('../data/NH3_emis_acid_plot.csv')
```

The `merge` function is in the Pandas module, and seems quite analogous to the R version (we actually used one from the data.table package above, but it is nearly identical in behavior to the version from the R base package). One difference is in the `on` argument instead of `by`.

```
amm_comb = pd.merge(amm_int, amm_plot, on = 'pmid')
print(amm_comb)
```

```
##       pmid      ct       cta    dt              t_start                t_end     j_NH3 treat    app_
## 0     1947    1.73    1.7333  1.73  2020-11-18 13:40:00  2020-11-18 15:24:00  0.008822  tank  2020-1
## 1     1947    3.46    3.4667  1.73  2020-11-18 15:24:00  2020-11-18 17:08:00  0.000000  tank  2020-1
## 2     1947    5.19    5.2000  1.73  2020-11-18 17:08:00  2020-11-18 18:52:00  0.006170  tank  2020-1
## 3     1947    6.92    6.9333  1.73  2020-11-18 18:52:00  2020-11-18 20:36:00  0.013609  tank  2020-1
## 4     1947    8.65    8.6667  1.73  2020-11-18 20:36:00  2020-11-18 22:20:00  0.015426  tank  2020-1
## ...    ...     ...       ...   ...                  ...                  ...       ...   ...
## 3484  1982  178.19  178.5300  1.73  2020-12-16 23:49:00  2020-12-17 01:33:00  0.010049  none  2020-1
## 3485  1982  179.92  180.2700  1.73  2020-12-17 01:33:00  2020-12-17 03:17:00  0.009846  none  2020-1
## 3486  1982  181.65  182.0000  1.73  2020-12-17 03:17:00  2020-12-17 05:01:00  0.009571  none  2020-1
## 3487  1982  183.38  183.7300  1.73  2020-12-17 05:01:00  2020-12-17 06:45:00  0.009954  none  2020-1
## 3488  1982  185.11  185.4700  1.73  2020-12-17 06:45:00  2020-12-17 08:29:00  0.011635  none  2020-1
##
## [3489 rows x 13 columns]
```

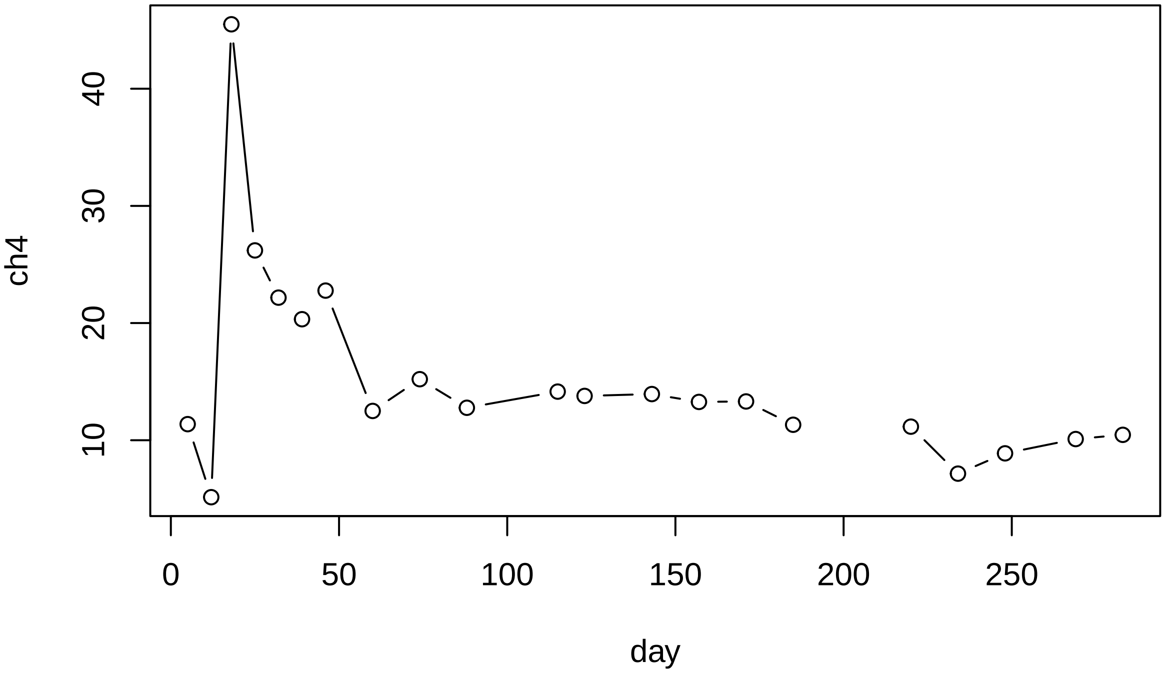
Both functions are flexible, and can merge on multiple columns, keep or drop unmatched rows, and add suffixes to columns as needed.

Interpolation

Interpolation is used to estimate a value based on values made under similar conditions. For the type of data we will be working with in this course, it commonly means estimating a value at a particular time based on neighboring values measured at a different time.

```
dat <- fread('../data/slurry_emis.csv')
datr1 <- dat[reactor == 'R1', ]
```

```
plot(ch4 ~ day, data = datr1, type = 'b')
```



If, for some reason, we need values for 10 and 20 d, interpolation is an obvious approach.

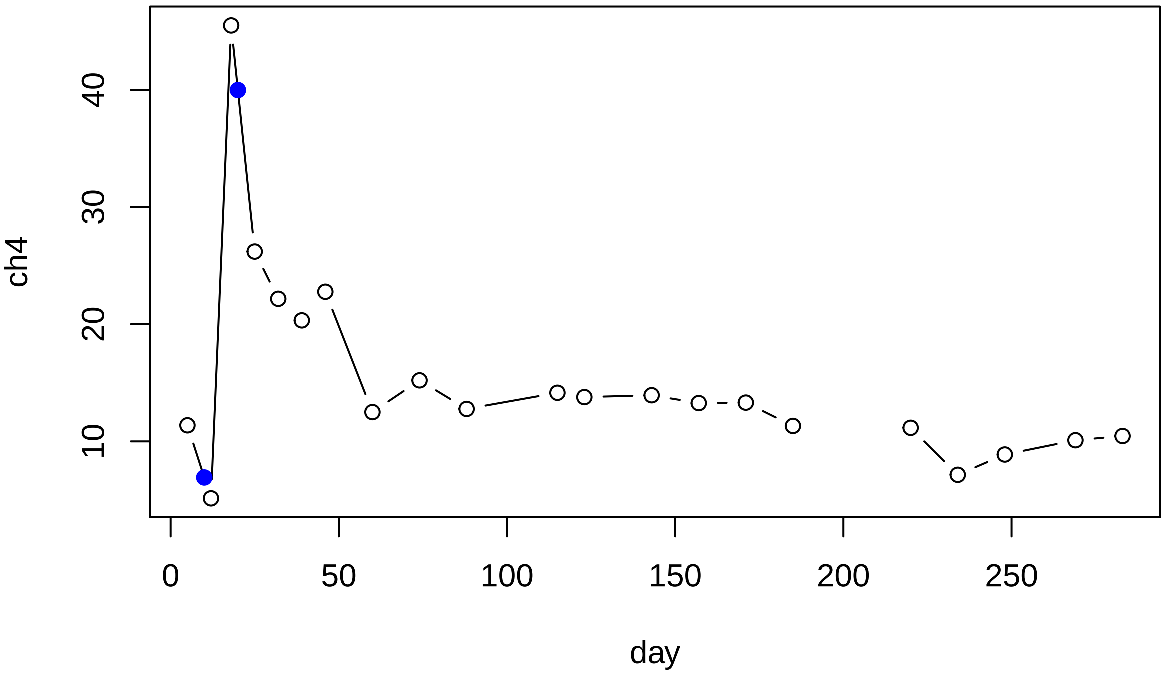
```
args(approx)
```

```
## function (x, y = NULL, xout, method = "linear", n = 50, yleft,
##     yright, rule = 1, f = 0, ties = mean, na.rm = TRUE)
## NULL
```

```
approx(datr1[, day], datr1[, ch4], xout = c(10, 20))$y
```

```
## [1]  6.921143 39.985714
```

```
yinterp <- approx(datr1[, day], datr1[, ch4], xout = c(10, 20))$y
plot(ch4 ~ day, data = datr1, type = 'b')
points(c(10, 20), yinterp, col = 'blue', pch = 19)
```

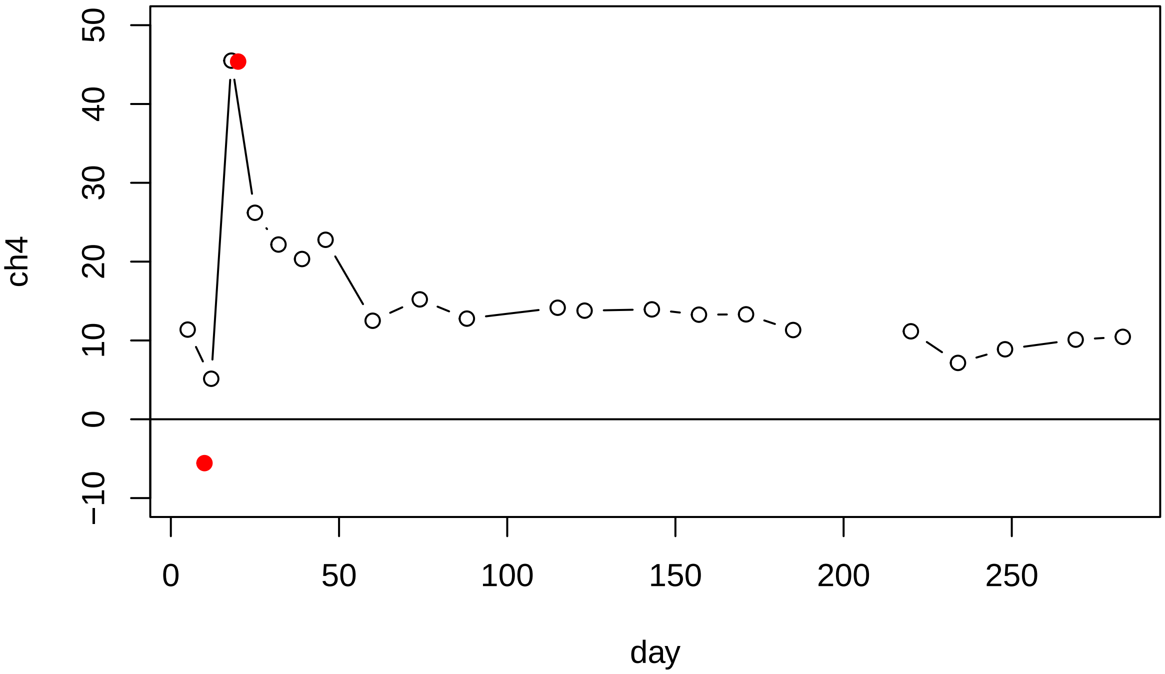


That function `approx` uses linear interpolation. There are more sophisticated methods that could be used in the `spline` function. But be sure the method is appropriate! As seen in this example, the default method is not always the most appropriate.

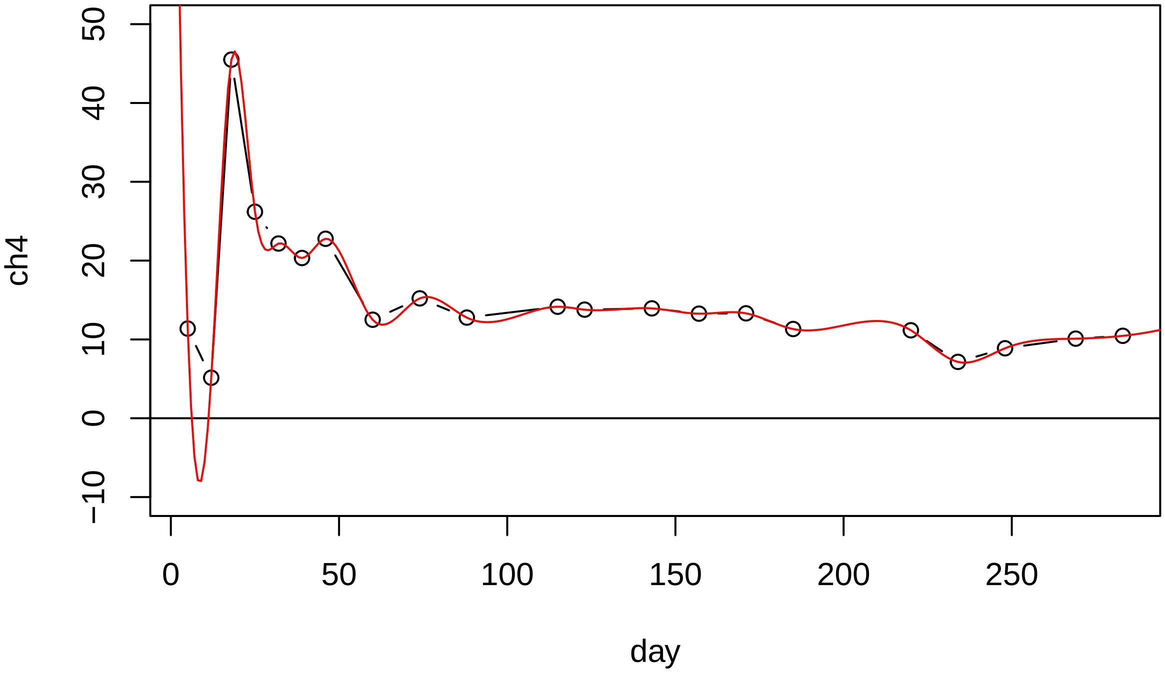
```
#?spline
args(spline)
```

```
## function (x, y = NULL, n = 3 * length(x), method = "fmm", xmin = min(x),
##     xmax = max(x), xout, ties = mean)
## NULL
```

```
yspline <- spline(datr1[, day], datr1[, ch4], xout = c(10, 20))$y
plot(ch4 ~ day, data = datr1, type = 'b', ylim = c(-10, 50))
abline(h = 0)
points(c(10, 20), yspline, col = 'red', pch = 19)
```



```r
xout <- 0:300
yspline2 <- spline(datr1[, day], datr1[, ch4], xout = xout)$y
plot(ch4 ~ day, data = datr1, type = 'b', ylim = c(-10, 50))
abline(h = 0)
lines(xout, yspline2, col = 'red')
```



Sometimes simple is best.

In Python:

```python
import numpy as np
import pandas as pd
```

```
#import matplotlib.pyplot as plt

dat = pd.read_csv('../data/slurry_emis.csv')
print(dat)
```

```
##     reactor      ch4       co2      flow  day  gas  temp
## 0        R1   11.374   338.300  0.063000    5  co2    20
## 1        R2    9.638   348.235  0.073000    5  co2    20
## 2        R3    5.221   320.180  0.082000    5  co2    20
## 3        R4    7.200   313.690  0.081000    5  co2    20
## 4        R5   16.000   371.500  0.084000    5  co2    30
## ..      ...      ...       ...       ...  ...  ...   ...
## 349     R12   59.150  1002.000  0.061214  283   ar    20
## 350     R13   48.320   858.300  0.067546  283   ar    30
## 351     R14   49.970   865.400  0.068602  283   ar    30
## 352     R15   45.260   837.200  0.068602  283   ar    30
## 353     R16  105.800   895.000  0.059103  283   ar    30
##
## [354 rows x 7 columns]
```

```
datr1 = dat[dat['reactor'] == 'R1']
print(datr1)
```

```
##     reactor     ch4     co2      flow  day  gas  temp
## 0        R1  11.374  338.30  0.063000    5  co2    20
## 17       R1   5.140  193.00  0.074750   12  co2    20
## 34       R1  45.500  230.00  0.079350   18  co2    20
## 51       R1  26.200  190.00  0.065550   25  co2    20
## 67       R1  22.170  210.00  0.060950   32  co2    20
## 83       R1  20.330  197.00  0.057500   39  co2    20
## 99       R1  22.770  205.00  0.058650   46  co2    20
## 115      R1  12.500  236.00  0.058650   60  co2    20
## 131      R1  15.210  160.00  0.058650   74  co2    20
## 147      R1  12.770  122.10  0.058650   88  co2    20
## 163      R1  14.150  110.00  0.073879  115  co2    20
## 179      R1  13.780  112.30  0.067546  123  co2    20
## 195      R1  13.940  102.00  0.067546  143  co2    20
## 211      R1  13.270   97.91  0.067546  157  co2    20
## 227      R1  13.310   97.80  0.067546  171  co2    20
## 243      R1  11.320   95.34  0.074934  185  co2    20
## 258      R1     NaN     NaN  0.071768  200  co2    20
## 274      R1  11.160   92.00  0.071768  220  co2    20
## 290      R1   7.150   83.00  0.071768  234  co2    20
## 306      R1   8.880   78.12  0.071768  248  co2    20
## 322      R1  10.100   92.20  0.068602  269  co2    20
## 338      R1  10.460  101.90  0.068602  283  co2    20
```

```
xout = [10, 20]
print(xout)
```

```
## [10, 20]
```

```
print(type(xout))
```

```
## <class 'list'>
```

```
yout = np.interp(xout, datr1['day'], datr1['ch4'])
print(yout)
```

```
## [ 6.92114286 39.98571429]
```

That's linear interpolation. (Note that the help file (you get it with `help(np.interp)`) states that this function is "…for monotonically increasing sample points" but that seems to apply to the x values.) See the scipy.interpolate "sub-package" for alternatives.

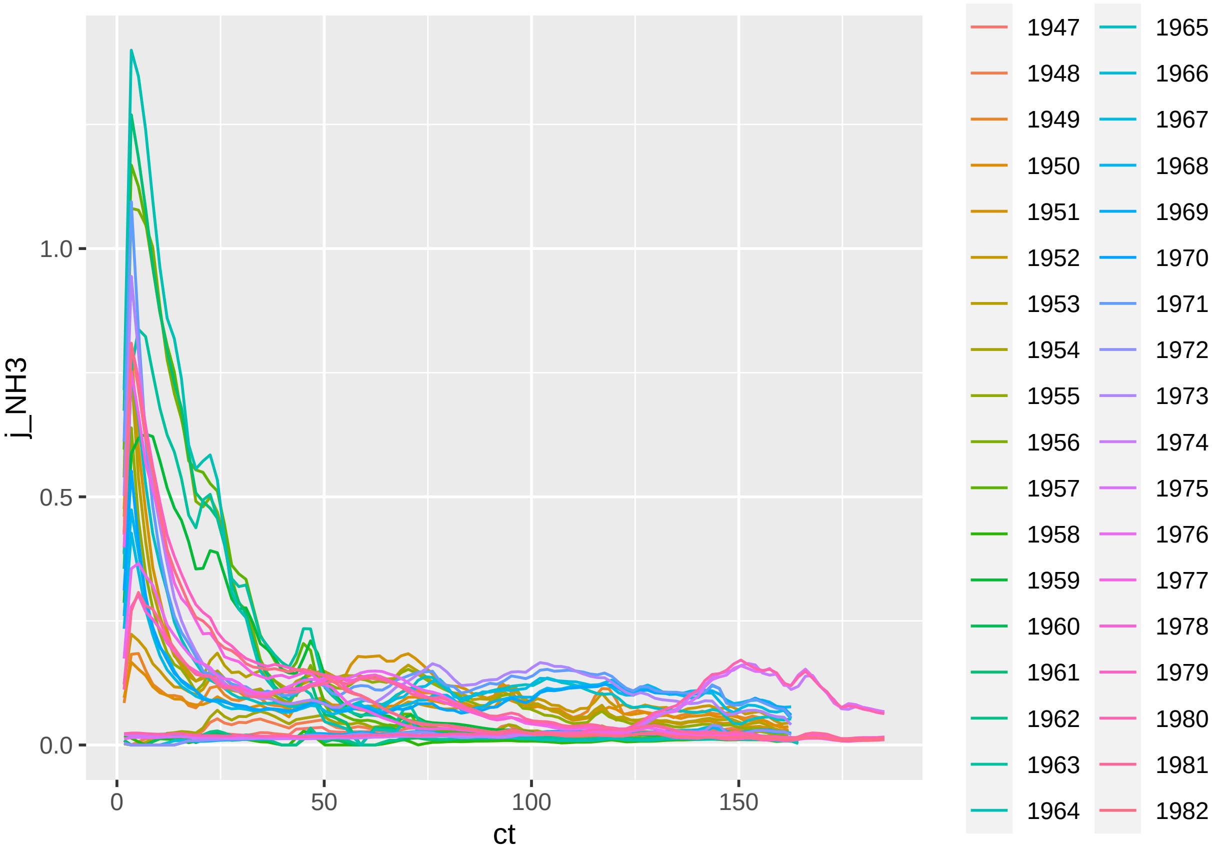
Integration

Integration is a common task in emission measurements. With older methods such as traps measurement of *cumulative* emission was common. But with an online measurement system it may be more common to measure emission rate at some points in time. So we need to be able to convert these to an estimate of total emission.

```
amm_int <- fread('../data/NH3_emis_acid_interval.csv')
amm_int
```

```
##       pmid     ct      cta   dt             t_start               t_end     j_NH3
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350
```

```
library(ggplot2)
amm_int[, pmid := factor(pmid)]
ggplot(amm_int, aes(ct, j_NH3, colour = pmid)) +
  geom_line()
```



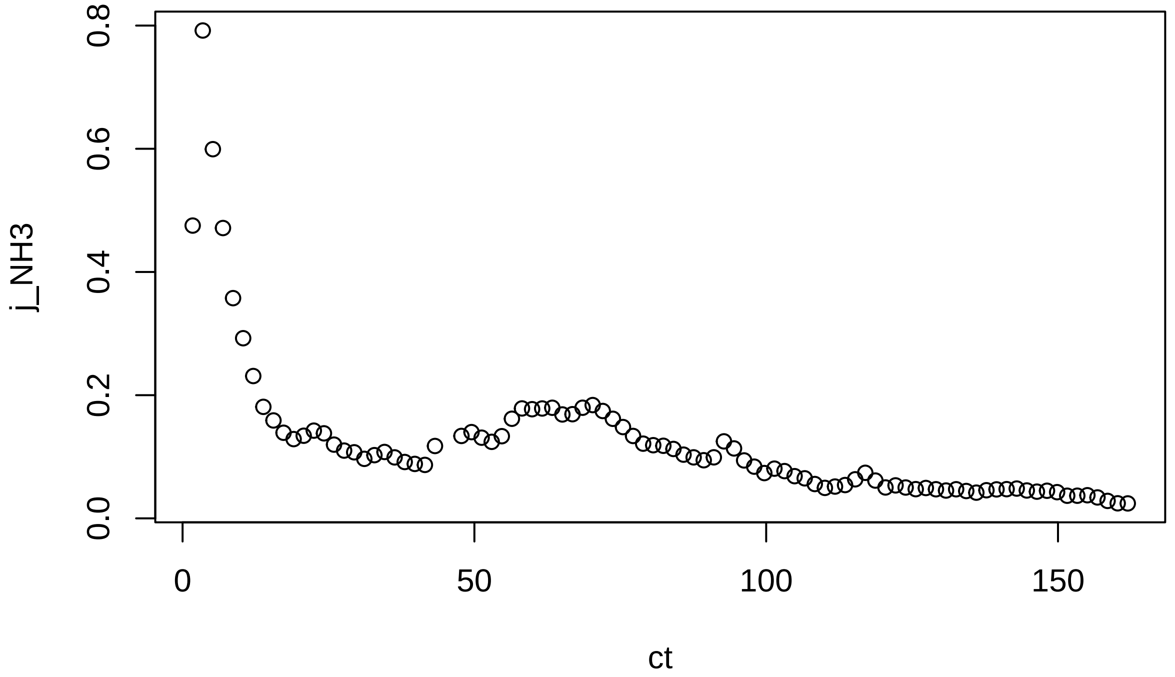
Here we have ammonia volatilization in mass of N as kg/h-ha and want kg/ha. There are some R packages with integration functions but I like a function I wrote called `mintegrate()` (for *m*easurement integration, as opposed to other functions focused on function integration).

```
source('../R-functions/mintegrate.R')
args(mintegrate)
```

```
## function (x, y, method = "midpoint", lwr = min(x), upr = max(x),
##     ylwr = y[which.min(x)], value = "all")
## NULL
```

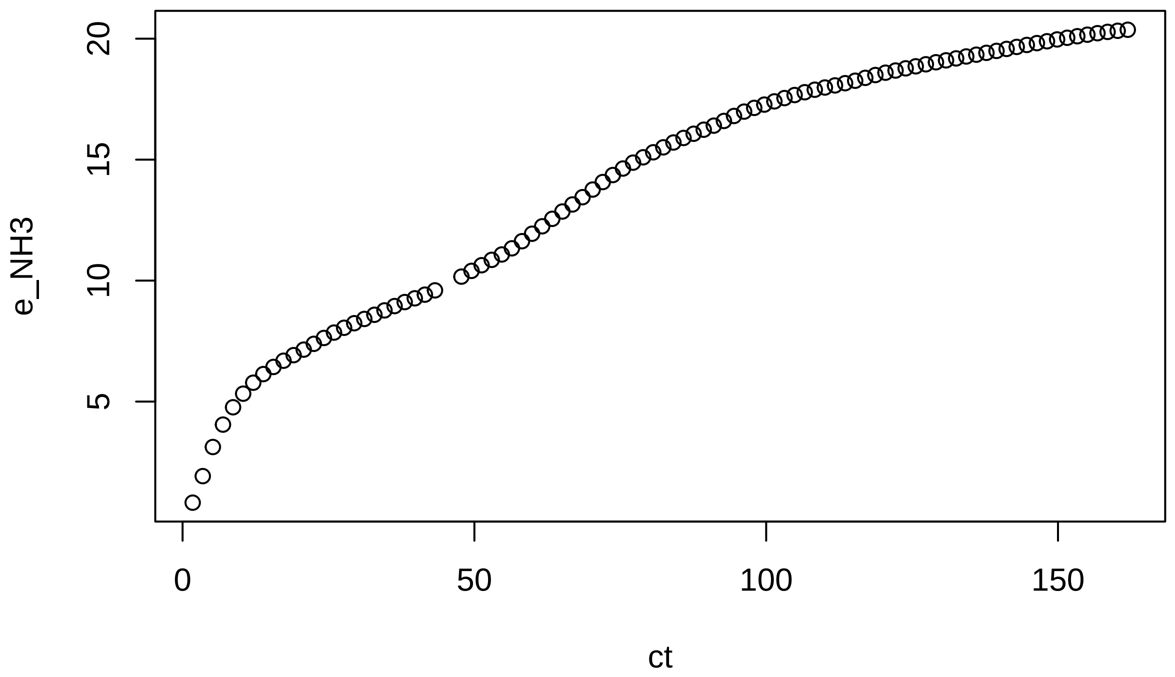
Let's apply it to a single flux curve.

```
amm1 <- amm_int[pmid == 1951]
plot(j_NH3 ~ ct, data = amm1)
```



```
amm1[, e_NH3 := mintegrate(ct, j_NH3, method = 'trap', lwr = 0)]
```

```
plot(e_NH3 ~ ct, data = amm1)
```



Grouped operations

Often we need to apply some kind of operation, for example any of the new column operations done above, *separately* to individual groups. Examples of groups include individual reactors, bottles, cows, or field plots. It is common to need some kind of a summary.

```
library(data.table)
dat <- fread('../data/slurry_emis_small.csv')
dat
```

```
##    reactor     ch4   co2 day gas temp    flow
## 1:      R1  11.374 338.3   5 co2   20 0.08200
## 2:      R1  45.500 230.0  18 co2   20 0.08400
## 3:      R1  22.170 210.0  32 co2   20 0.07400
## 4:      R5  16.000 371.5   5 co2   30 0.07475
## 5:      R5 124.800 440.0  18 co2   30 0.06900
## 6:      R5  81.290 415.0  32 co2   30 0.07360
```

Mean methane concentration by bottle.

```
dat[, .(ch4.mn = mean(ch4)), by = reactor]
```

```
##    reactor ch4.mn
## 1:      R1 26.348
## 2:      R5 74.030
```

For cumulative emission, we can integrate by bottle. Here we do not want 1 row per bottle in the output, but we want to add to the original data frame.

```
dat[, e_ch4 := mintegrate(day, flow * ch4, method = 'trap', lwr = 0), by = reactor]
dat
```

```
##    reactor     ch4   co2 day gas temp    flow     e_ch4
## 1:      R1  11.374 338.3   5 co2   20 0.08200   4.66334
## 2:      R1  45.500 230.0  18 co2   20 0.08400  35.56868
## 3:      R1  22.170 210.0  32 co2   20 0.07400  73.80674
## 4:      R5  16.000 371.5   5 co2   30 0.07475   5.98000
## 5:      R5 124.800 440.0  18 co2   30 0.06900  69.72680
## 6:      R5  81.290 415.0  32 co2   30 0.07360 171.88581
```

Note that use of `=` for a summary versus `:=` to add a column. These are data.table operators. Here we calculated emission rate as `flow * ch4` internally and did not save the result. If we want to add it as a column, do we need a grouped operation? No, because each value of the result depends only on a single row.

```
dat[, qch4 := flow * ch4]
dat
```

```
##    reactor     ch4   co2 day gas temp    flow     e_ch4     qch4
## 1:      R1  11.374 338.3   5 co2   20 0.08200   4.66334 0.932668
## 2:      R1  45.500 230.0  18 co2   20 0.08400  35.56868 3.822000
## 3:      R1  22.170 210.0  32 co2   20 0.07400  73.80674 1.640580
## 4:      R5  16.000 371.5   5 co2   30 0.07475   5.98000 1.196000
## 5:      R5 124.800 440.0  18 co2   30 0.06900  69.72680 8.611200
## 6:      R5  81.290 415.0  32 co2   30 0.07360 171.88581 5.982944
```

For better or worse, there are many different ways to carry out grouped operations in R. These include old base R functions like `by` and `aggregate` (which is still a good function). The dplyr package, part of the "tidyverse" set of packages, is aimed at grouped operations, but its prevalence in search results shouldn't be taken to mean it is the only or even best approach.

```
library(dplyr)
dat <- fread('../data/slurry_emis_small.csv')
dat <- dat %>% group_by(reactor) %>% mutate(ech4 = mintegrate(day, flow * ch4, method = 'trap', lwr = 0
dat
```

```
## # A tibble: 6 x 8
## # Groups:   reactor [2]
##   reactor   ch4   co2   day gas    temp   flow   ech4
##   <chr>   <dbl> <dbl> <int> <chr> <int>  <dbl>  <dbl>
## 1 R1       11.4  338.     5 co2      20 0.082    4.66
## 2 R1       45.5  230     18 co2      20 0.084   35.6
## 3 R1       22.2  210     32 co2      20 0.074   73.8
## 4 R5       16    372.     5 co2      30 0.0748   5.98
## 5 R5      125.   440     18 co2      30 0.069   69.7
## 6 R5       81.3  415     32 co2      30 0.0736 172.
```

I don't like tidyverse.

In Python

```
from mintegrate import mintegrate
import pandas as pd

dat = pd.read_csv('../data/slurry_emis_small.csv')
dat['qch4'] = dat['flow'] * dat['ch4']
print(dat)
```

```
##   reactor      ch4    co2  day  gas  temp     flow      qch4
## 0      R1   11.374  338.3    5  co2    20  0.08200  0.932668
## 1      R1   45.500  230.0   18  co2    20  0.08400  3.822000
## 2      R1   22.170  210.0   32  co2    20  0.07400  1.640580
## 3      R5   16.000  371.5    5  co2    30  0.07475  1.196000
## 4      R5  124.800  440.0   18  co2    30  0.06900  8.611200
## 5      R5   81.290  415.0   32  co2    30  0.07360  5.982944
```

Here is integration by bottle.

```
print(dat.groupby(['reactor']).apply(lambda x: mintegrate(x['day'], x['qch4'])))
```

```
## reactor
## R1       0      6.062342
##          1     57.659342
##          2     69.143402
## R5       3      7.774000
##          4    124.025200
##          5    165.905808
## Name: qch4, dtype: float64
```

Those are the values, but for some reason the Pandas developers have not made it so easy to get the results back in the original data frame. To do it, we need to drop the `reactor` index.

```
dat['ech4'] = dat.groupby(['reactor']).apply(lambda x: mintegrate(x['day'], x['qch4'], lwr = 0)).reset_

print(dat)
```

```
##   reactor      ch4    co2  day  gas  temp     flow      qch4        ech4
## 0      R1   11.374  338.3    5  co2    20  0.08200  0.932668    8.394012
## 1      R1   45.500  230.0   18  co2    20  0.08400  3.822000   59.991012
## 2      R1   22.170  210.0   32  co2    20  0.07400  1.640580   71.475072
## 3      R5   16.000  371.5    5  co2    30  0.07475  1.196000   10.764000
## 4      R5  124.800  440.0   18  co2    30  0.06900  8.611200  127.015200
## 5      R5   81.290  415.0   32  co2    30  0.07360  5.982944  168.895808
```

Dates and times

The challenge with date and time data is getting R or Python to correctly interprete your values. Once that is sorted out, manipulation is simple. Newer functions for reading in data from add-on packages fortunately make this quite easy, by recognizing date/time objects when data are read in.

```
amm_int <- fread('../data/NH3_emis_acid_interval.csv')
amm_int
```

```
##       pmid     ct      cta   dt             t_start               t_end      j_NH3
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350
```

The `t_start` and `t_end` columns sure *look* like date/time objects, but we can't trust their appearance.

```
source('../R-functions/dfsumm.R')
dfsumm(amm_int)
```

```
##
##  3489 rows and 7 columns
##  3489 unique rows
##                        pmid      ct     cta      dt             t_start               t_end    j_NH3
## Class               integer numeric numeric numeric     POSIXct, POSIXt     POSIXct, POSIXt  numeric
## Minimum                1950    1.73    1.73    1.73 2020-11-18 13:40:00 2020-11-18 15:24:00        0
## Maximum                1980     185    7220    4.53 2020-12-17 06:53:00 2020-12-17 08:37:00      1.4
## Mean                   1970    85.5    1990    1.74 2020-12-02 23:50:58 2020-12-03 01:35:24   0.0867
## Unique (excld. NA)       36     174     508       2                3489                3489     3343
## Missing values            0       0       0       0                   0                   0        0
## Sorted                 TRUE   FALSE   FALSE   FALSE               FALSE               FALSE    FALSE
##
```

They actually are. So we can use them in math, for example to calculate an elapsed time.

```
amm_int[, etime := t_start - t_start[1]]
amm_int
```

```
##       pmid     ct      cta   dt             t_start               t_end      j_NH3         etime
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216       0 secs
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000    6240 secs
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700   12480 secs
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090   18720 secs
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260   24960 secs
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490 2455740 secs
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460 2461980 secs
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709 2468220 secs
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536 2474460 secs
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350 2480700 secs
```

That should be a grouped operation, presumably.

```
amm_int[, etime := t_start - t_start[1], by = pmid]
amm_int
```

```
##       pmid     ct      cta   dt             t_start               t_end     j_NH3       etime
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216      0 secs
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000   6240 secs
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700  12480 secs
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090  18720 secs
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260  24960 secs
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490 636480 secs
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460 642720 secs
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709 648960 secs
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536 655200 secs
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350 661440 secs
```

We can set units using the `difftime()` function.

```
amm_int[, etime2 := as.numeric(t_start - t_start[1], units = 'hours'), by = pmid]
amm_int
```

```
##       pmid     ct      cta   dt             t_start               t_end     j_NH3       etime     eti
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216      0 secs   0.000
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000   6240 secs   1.733
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700  12480 secs   3.466
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090  18720 secs   5.200
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260  24960 secs   6.933
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490 636480 secs 176.800
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460 642720 secs 178.533
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709 648960 secs 180.266
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536 655200 secs 182.000
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350 661440 secs 183.733
```

(Notice that I have used a new column in this last example because data.tables seem to hold tight to column types.)

Now, how about cases where date/time data are not read in correctly?

```
amm_int <- read.csv('../data/NH3_emis_acid_interval.csv')
amm_int <- data.table(amm_int)
dfsumm(amm_int)
```

```
##
##  3489 rows and 7 columns
##  3489 unique rows
##                       pmid      ct     cta      dt             t_start               t_end   j_NH3
## Class              integer numeric numeric numeric           character           character numeric
## Minimum               1950    1.73    1.73    1.73 2020-11-18 13:40:00 2020-11-18 15:24:00       0
## Maximum               1980     185    7220    4.53 2020-12-17 06:53:00 2020-12-17 08:37:00     1.4
## Mean                  1970    85.5    1990    1.74                <NA>                <NA>  0.0867
## Unique (excld. NA)      36     174     508       2                3489                3489    3343
## Missing values           0       0       0       0                   0                   0       0
## Sorted                TRUE   FALSE   FALSE   FALSE               FALSE               FALSE   FALSE
##
```

Now we have character data–ultimately more flexible, but requiring more effort.

The easiest way to convert *to* date/time in R is with the lubridate package.

```
library(lubridate)
amm_int[, date_time_start := ymd_hms(t_start)]
amm_int
```

```
##        pmid     ct      cta   dt             t_start               t_end     j_NH3     date_time_star
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216 2020-11-18 13:40:0
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000 2020-11-18 15:24:0
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700 2020-11-18 17:08:0
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090 2020-11-18 18:52:0
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260 2020-11-18 20:36:0
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490 2020-12-16 23:49:0
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460 2020-12-17 01:33:0
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709 2020-12-17 03:17:0
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536 2020-12-17 05:01:0
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350 2020-12-17 06:45:0
```

```
dfsumm(amm_int)
```

```
##
##  3489 rows and 8 columns
##  3489 unique rows
##                         pmid      ct     cta      dt             t_start               t_end   j_NH3
## Class              integer numeric numeric numeric           character           character numeric
## Minimum               1950    1.73    1.73    1.73 2020-11-18 13:40:00 2020-11-18 15:24:00       0 2
## Maximum               1980     185    7220    4.53 2020-12-17 06:53:00 2020-12-17 08:37:00     1.4 2
## Mean                  1970    85.5    1990    1.74                <NA>                <NA>  0.0867 2
## Unique (excld. NA)      36     174     508       2                3489                3489    3343
## Missing values           0       0       0       0                   0                   0       0
## Sorted                TRUE   FALSE   FALSE   FALSE               FALSE               FALSE   FALSE
##
```

The package has a lot of variations on the function we use below, for example, with month first, and without time.

Even more flexible is the `as.POSIXct()` function. But I have been using it for more than a decade and still have to check the abbreviations in the help file for `strptime`.

```
amm_int[, date_time_end := as.POSIXct(t_end, format = '%Y-%m-%d %H:%M:%S')]
amm_int
```

```
##        pmid     ct      cta   dt             t_start               t_end     j_NH3     date_time_star
##    1: 1947   1.73   1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216 2020-11-18 13:40:0
##    2: 1947   3.46   3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000 2020-11-18 15:24:0
##    3: 1947   5.19   5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700 2020-11-18 17:08:0
##    4: 1947   6.92   6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090 2020-11-18 18:52:0
##    5: 1947   8.65   8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260 2020-11-18 20:36:0
##   ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490 2020-12-16 23:49:0
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460 2020-12-17 01:33:0
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709 2020-12-17 03:17:0
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536 2020-12-17 05:01:0
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350 2020-12-17 06:45:0
```

In Python, the Pandas function does not automatically recognize our date/time columns here.

```
amm_int = pd.read_csv('../data/NH3_emis_acid_interval.csv')
print(amm_int.dtypes)
```

```
## pmid         int64
## ct         float64
## cta        float64
## dt         float64
## t_start     object
## t_end       object
## j_NH3      float64
## dtype: object
```

So we can use the `to_datetime()` function from the same package.

```
amm_int['date_time_start'] = pd.to_datetime(amm_int['t_start'])
print(amm_int.dtypes)
```

```
## pmid                        int64
## ct                        float64
## cta                       float64
## dt                        float64
## t_start                    object
## t_end                      object
## j_NH3                     float64
## date_time_start    datetime64[ns]
## dtype: object
```

And we can now do math (but I haven't looked into unit issues yet).

```
print(amm_int['date_time_start'] - min(amm_int['date_time_start']))
```

```
## 0       0 days 00:00:00
## 1       0 days 01:44:00
## 2       0 days 03:28:00
## 3       0 days 05:12:00
## 4       0 days 06:56:00
##               ...
## 3484   28 days 10:09:00
## 3485   28 days 11:53:00
## 3486   28 days 13:37:00
## 3487   28 days 15:21:00
## 3488   28 days 17:05:00
## Name: date_time_start, Length: 3489, dtype: timedelta64[ns]
```

Alternatively, we can use the `parse_dates` argument at the time the file is read in.

```
amm_int = pd.read_csv('../data/NH3_emis_acid_interval.csv', parse_dates = ['t_start', 't_end'])
print(amm_int.dtypes)
```

```
## pmid                 int64
## ct                 float64
## cta                float64
## dt                 float64
## t_start    datetime64[ns]
## t_end      datetime64[ns]
## j_NH3              float64
## dtype: object
```

```
amm_int['t_start'] - min(amm_int['t_start'])
```

```
## 0       0 days 00:00:00
## 1       0 days 01:44:00
## 2       0 days 03:28:00
## 3       0 days 05:12:00
## 4       0 days 06:56:00
##               ...
## 3484   28 days 10:09:00
## 3485   28 days 11:53:00
## 3486   28 days 13:37:00
## 3487   28 days 15:21:00
## 3488   28 days 17:05:00
## Name: t_start, Length: 3489, dtype: timedelta64[ns]
```

Reshaping

A given dataset can be organized in a variety of ways. In some cases, a certain structure may be needed (or at least helpful) for a particular purpose. We might recognize two general categories: "long" or "tall", where each variable shows up in only a single column, and "wide", where a single variable is present in multiple columns.

We can use the same data to demonstrate. They are originally in a more-or-less long format. We will simplify things a bit by getting rid of all but one replicate bottle (`reactor`) for each condition.

```
dat <- fread('../data/slurry_emis_small.csv')
dat <- dat[reactor != 'bg', ]
dim(dat)
```

```
## [1] 6 7
```

```
args(dcast)
```

```
## function (data, formula, fun.aggregate = NULL, ..., margins = NULL,
##     subset = NULL, fill = NULL, value.var = guess(data))
## NULL
```

```
datwide <- dcast(dat, day ~ temp + gas, value.var = 'ch4')
datwide
```

```
##    day 20_co2 30_co2
## 1:   5 11.374  16.00
## 2:  18 45.500 124.80
## 3:  32 22.170  81.29
```

This wide format is useful when individual observations need to be compared between treatments or experimental units at fixed times. R graphics and data analysis functions generally do not require it, however.

We could go even "longer" than the original structure.

```
datlong <- melt(dat, id.vars = c('reactor', 'gas', 'temp', 'day'))
datlong
```

```
##     reactor gas temp day variable     value
##  1:      R1 co2   20   5      ch4  11.37400
##  2:      R1 co2   20  18      ch4  45.50000
##  3:      R1 co2   20  32      ch4  22.17000
##  4:      R5 co2   30   5      ch4  16.00000
##  5:      R5 co2   30  18      ch4 124.80000
##  6:      R5 co2   30  32      ch4  81.29000
##  7:      R1 co2   20   5      co2 338.30000
##  8:      R1 co2   20  18      co2 230.00000
##  9:      R1 co2   20  32      co2 210.00000
## 10:      R5 co2   30   5      co2 371.50000
## 11:      R5 co2   30  18      co2 440.00000
## 12:      R5 co2   30  32      co2 415.00000
## 13:      R1 co2   20   5     flow   0.08200
## 14:      R1 co2   20  18     flow   0.08400
## 15:      R1 co2   20  32     flow   0.07400
## 16:      R5 co2   30   5     flow   0.07475
## 17:      R5 co2   30  18     flow   0.06900
## 18:      R5 co2   30  32     flow   0.07360
```

We have not lost or gained any data here, but now have the numeric value of every single response variable in one column.

In Python . . .

```
# NTS
# WIP
# dat
# airw = airtot.pivot_table(index = ['aircleaner', 'compound_name'], columns = ['flow_dir'], values = [
```

Logs, reports, and exported data

R and Python users can export data and related information to facilitate data checking, but also to create a record. In R, the rmarkdown package can be used to combine descriptive text with R code and results. This document was made with it. Data frames can be written out with `write.csv()` or the data.table function `fwrite()`. For Python, the pandas function `to_csv()` can be used.