

Online Analytical Chemistry notes: data manipulation

Sasha D. Hafner

14:35 - 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	8
New variables (adding columns)	14
Subsetting	16
Merging	18
Interpolation	20
Integration	25
Grouped operations	27
Dates and times	30
Reshaping	36
Logs, reports, and exported data	39

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 similar 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 situation 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 operations 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 visualization

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:

```
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 C1H302 C3H701 C2H502 C7H1102 C10H17
## 1 12/7/2023 10:34    45267.44 0.26983 0.23759 0.19983 0.00494390 0.0046413
## 2 12/7/2023 10:34    45267.44 0.26303 0.25205 0.18137 0.00074215 0.0012158
## 3 12/7/2023 10:34    45267.44 0.27097 0.22796 0.19361 0.00399780 0.0021266
## 4 12/7/2023 10:34    45267.44 0.24479 0.19712 0.17835 0.00546340 0.0026745
## 5 12/7/2023 10:34    45267.44 0.28258 0.23840 0.18143 0.00438240 0.0055961
## 6 12/7/2023 10:34    45267.44 0.18797 0.22651 0.18668 0.00426480 0.0053425
##      C9H1501 C8H1502 C9H1502 C8H1303 C9H1503 C8H1304 C9H1504
## 1 0.032477 0.00222460 0.017653 0.0015724 0.00074916 -0.00032970 0.00033354
## 2 0.035656 0.00027488 0.017610 0.0000506 0.00132800 0.00002640 0.00030222
## 3 0.038066 -0.00016937 0.019828 0.0011651 0.00235960 0.00037222 0.00008730
## 4 0.038752 0.00192130 0.024411 0.0020600 0.00228380 -0.00026098 -0.00073415
## 5 0.036147 -0.00021588 0.016462 0.0014995 0.00194660 -0.00074100 0.00060288
## 6 0.032794 -0.00059051 0.017606 0.0015675 0.00017333 -0.00062942 0.00020794
##      C10H1704
## 1 0.00101170
## 2 -0.00087358
## 3 -0.00040130
## 4 0.00076622
## 5 0.00073954
## 6 -0.00051198
```

```
summary(voc)
```

```
## time_string      time_number      C1H302      C3H701
## Length:12737      Min.      :45267      Min.      : -0.04658      Min.      : -0.02714
## Class :character  1st Qu.:45268      1st Qu.: 0.03517      1st Qu.: 0.02615
## Mode  :character  Median :45268      Median : 0.92146      Median : 1.06835
##                               Mean  :45268      Mean  : 0.71118      Mean  : 1.18624
##                               3rd Qu.:45268      3rd Qu.: 1.22900      3rd Qu.: 2.07972
##                               Max.  :45268      Max.  : 1.64360      Max.  : 6.18590
##                               NA's   :189        NA's   :189
##      C2H502      C7H1102      C10H17      C9H1501
## Min.      : -0.02579      Min.      : -0.00457      Min.      : -0.00265      Min.      : -0.00357
## 1st Qu.: 0.01155      1st Qu.: 0.00313      1st Qu.: 0.00120      1st Qu.: 0.02411
```

```
## Median : 0.30210 Median : 0.07951 Median : 2.42820 Median : 2.84250
## Mean : 0.27231 Mean : 0.06684 Mean : 3.69658 Mean : 2.68297
## 3rd Qu.: 0.41262 3rd Qu.: 0.12494 3rd Qu.: 5.78310 3rd Qu.: 5.29805
## Max. : 4.77970 Max. : 0.15418 Max. : 17.59490 Max. : 6.08970
## NA's :189 NA's :189 NA's :189 NA's :189
## C8H1502 C9H1502 C8H1303 C9H1503
## Min. : -0.00333 Min. : -0.00361 Min. : -0.00309 Min. : -0.00259
## 1st Qu.: 0.00136 1st Qu.: 0.00679 1st Qu.: 0.00104 1st Qu.: 0.00117
## Median : 0.01112 Median : 1.11245 Median : 0.02538 Median : 0.04290
## Mean : 0.00983 Mean : 0.92017 Mean : 0.02044 Mean : 0.03942
## 3rd Qu.: 0.01758 3rd Qu.: 1.78940 3rd Qu.: 0.03766 3rd Qu.: 0.07405
## Max. : 0.10247 Max. : 1.90220 Max. : 0.05152 Max. : 0.09259
## NA's :189 NA's :189 NA's :189 NA's :189
## C8H1304 C9H1504 C10H1704
## Min. : -0.00211 Min. : -0.00236 Min. : -0.00554
## 1st Qu.: 0.00038 1st Qu.: 0.00038 1st Qu.: 0.00036
## Median : 0.00423 Median : 0.00309 Median : 0.00208
## Mean : 0.00694 Mean : 0.00463 Mean : 0.00279
## 3rd Qu.: 0.01384 3rd Qu.: 0.00872 3rd Qu.: 0.00524
## Max. : 0.02475 Max. : 0.01859 Max. : 0.01205
## NA's :189 NA's :189 NA's :189
```

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

```
## time_string time_number C1H302 C3H701 C2H502 C7H1102
## 1: 12/7/2023 10:34 45267.44 0.26983000 0.23759000 0.19983000 0.00494390
## 2: 12/7/2023 10:34 45267.44 0.26303000 0.25205000 0.18137000 0.00074215
## 3: 12/7/2023 10:34 45267.44 0.27097000 0.22796000 0.19361000 0.00399780
## 4: 12/7/2023 10:34 45267.44 0.24479000 0.19712000 0.17835000 0.00546340
## 5: 12/7/2023 10:34 45267.44 0.28258000 0.23840000 0.18143000 0.00438240
## ---
## 12733: 12/7/2023 17:38 45267.73 0.03405808 0.02747731 0.01098923 0.01535654
## 12734: 12/7/2023 17:38 45267.73 0.03853077 0.02504192 0.01067731 0.01657769
## 12735: 12/7/2023 17:38 45267.73 0.03404269 0.02224269 0.01141692 0.01477038
## 12736: 12/7/2023 17:38 45267.73 0.03497692 0.02501538 0.01088462 0.01620885
## 12737: 12/7/2023 17:38 45267.73 0.03452077 0.02439692 0.01039846 0.01530308
## C10H17 C9H1501 C8H1502 C9H1502 C8H1303 C9H1503
## 1: 4.641300e-03 0.0324770 0.002224600 0.0176530 0.001572400 0.00074916
## 2: 1.215800e-03 0.0356560 0.000274880 0.0176100 0.000050600 0.00132800
## 3: 2.126600e-03 0.0380660 -0.000169370 0.0198280 0.001165100 0.00235960
## 4: 2.674500e-03 0.0387520 0.001921300 0.0244110 0.002060000 0.00228380
## 5: 5.596100e-03 0.0361470 -0.000215880 0.0164620 0.001499500 0.00194660
## ---
## 12733: 1.477731e-03 0.1282731 0.002591538 0.1057462 0.004469231 0.01511654
## 12734: 6.119231e-04 0.1302038 0.002445654 0.1070077 0.004283846 0.01568962
## 12735: 1.053115e-03 0.1296077 0.002466885 0.1051923 0.003843115 0.01483231
## 12736: 8.473462e-04 0.1285192 0.002529231 0.1074577 0.003875000 0.01596577
## 12737: 1.123654e-05 0.1264692 0.002807692 0.1037269 0.003662269 0.01484231
## C8H1304 C9H1504 C10H1704
## 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
```

```
summary(voc)
```

```
## time_string      time_number      C1H302      C3H701
## Length:12737      Min.      :45267      Min.      : -0.04658      Min.      : -0.02714
## Class :character      1st Qu.:45268      1st Qu.:  0.03517      1st Qu.:  0.02615
## Mode  :character      Median :45268      Median :  0.92146      Median :  1.06835
##      Mean      :45268      Mean      :  0.71118      Mean      :  1.18624
##      3rd Qu.:45268      3rd Qu.:  1.22900      3rd Qu.:  2.07972
##      Max.      :45268      Max.      :  1.64360      Max.      :  6.18590
##      NA's      :189      NA's      :189
##      C2H502      C7H1102      C10H17      C9H1501
## Min.      : -0.02579      Min.      : -0.00457      Min.      : -0.00265      Min.      : -0.00357
## 1st Qu.:  0.01155      1st Qu.:  0.00313      1st Qu.:  0.00120      1st Qu.:  0.02411
## Median :  0.30210      Median :  0.07951      Median :  2.42820      Median :  2.84250
## Mean      :  0.27231      Mean      :  0.06684      Mean      :  3.69658      Mean      :  2.68297
## 3rd Qu.:  0.41262      3rd Qu.:  0.12494      3rd Qu.:  5.78310      3rd Qu.:  5.29805
## Max.      :  4.77970      Max.      :  0.15418      Max.      :17.59490      Max.      :  6.08970
## NA's      :189      NA's      :189      NA's      :189      NA's      :189
##      C8H1502      C9H1502      C8H1303      C9H1503
## Min.      : -0.00333      Min.      : -0.00361      Min.      : -0.00309      Min.      : -0.00259
## 1st Qu.:  0.00136      1st Qu.:  0.00679      1st Qu.:  0.00104      1st Qu.:  0.00117
## Median :  0.01112      Median :  1.11245      Median :  0.02538      Median :  0.04290
## Mean      :  0.00983      Mean      :  0.92017      Mean      :  0.02044      Mean      :  0.03942
## 3rd Qu.:  0.01758      3rd Qu.:  1.78940      3rd Qu.:  0.03766      3rd Qu.:  0.07405
## Max.      :  0.10247      Max.      :  1.90220      Max.      :  0.05152      Max.      :  0.09259
## NA's      :189      NA's      :189      NA's      :189      NA's      :189
##      C8H1304      C9H1504      C10H1704
## Min.      : -0.00211      Min.      : -0.00236      Min.      : -0.00554
## 1st Qu.:  0.00038      1st Qu.:  0.00038      1st Qu.:  0.00036
## Median :  0.00423      Median :  0.00309      Median :  0.00208
## Mean      :  0.00694      Mean      :  0.00463      Mean      :  0.00279
## 3rd Qu.:  0.01384      3rd Qu.:  0.00872      3rd Qu.:  0.00524
## Max.      :  0.02475      Max.      :  0.01859      Max.      :  0.01205
## NA's      :189      NA's      :189      NA's      :189
```

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.

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

```
##
## 12737 rows and 15 columns
## 12556 unique rows
##      time_string time_number C1H302 C3H701 C2H502 C7H1102
## Class      character      numeric numeric numeric numeric
## Minimum      12/7/2023 10:34      45300 -0.0466 -0.0271 -0.0258 -0.00457
```

```
## Maximum      12/7/2023 17:38      45300      1.64      6.19      4.78      0.154
## Mean          <NA>      45300      0.711      1.19      0.272      0.0668
## Unique (excl. NA)      425      76      9084      11454      11215      9363
## Missing values      0      0      189      189      189      189
## Sorted      TRUE      TRUE      FALSE      FALSE      FALSE      FALSE
##
##              C10H17  C9H1501  C8H1502  C9H1502  C8H1303  C9H1503
## Class      numeric  numeric  numeric  numeric  numeric  numeric
## Minimum    -0.00264 -0.00357 -0.00333 -0.00361 -0.00309 -0.00259
## Maximum      17.6      6.09      0.102      1.9      0.0515      0.0926
## Mean         3.7      2.68      0.00983      0.92      0.0204      0.0394
## Unique (excl. NA)    12213    11293    11019      8976    11291    11664
## Missing values      189      189      189      189      189      189
## Sorted      FALSE      FALSE      FALSE      FALSE      FALSE      FALSE
##
##              C8H1304  C9H1504  C10H1704
## Class      numeric  numeric  numeric
## Minimum    -0.00211 -0.00236 -0.00554
## Maximum      0.0248      0.0186      0.012
## Mean         0.00694      0.00463      0.00279
## Unique (excl. NA)    11355    11840    12142
## Missing values      189      189      189
## Sorted      FALSE      FALSE      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  C1H302  ...  C8H1304  C9H1504  C10H1704
## 0      12/7/2023 10:34    45267.4414  0.269830  ... -0.000330  0.000334  0.001012
## 1      12/7/2023 10:34    45267.4414  0.263030  ...  0.000026  0.000302 -0.000874
## 2      12/7/2023 10:34    45267.4414  0.270970  ...  0.000372  0.000087 -0.000401
```

```
## 3      12/7/2023 10:34  45267.4414  0.244790  ... -0.000261 -0.000734  0.000766
## 4      12/7/2023 10:34  45267.4414  0.282580  ... -0.000741  0.000603  0.000740
## ...      ...      ...      ...      ...      ...      ...
## 12732  12/7/2023 17:38  45267.7344  0.034058  ...  0.001797  0.001630  0.001364
## 12733  12/7/2023 17:38  45267.7344  0.038531  ...  0.002282  0.001850  0.000998
## 12734  12/7/2023 17:38  45267.7344  0.034043  ...  0.001947  0.001774  0.001329
## 12735  12/7/2023 17:38  45267.7344  0.034977  ...  0.002223  0.001807  0.000838
## 12736  12/7/2023 17:38  45267.7344  0.034521  ...  0.001944  0.002314  0.000816
##
## [12737 rows x 15 columns]
```

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

```
##          time_number      C1H302  ...      C9H1504      C10H1704
## count  12737.000000  12548.000000  ...  12548.000000  12548.000000
## mean   45267.587726    0.711176  ...    0.004631    0.002787
## std      0.085145    0.535461  ...    0.004636    0.002814
## min    45267.441400   -0.046582  ...   -0.002358   -0.005539
## 25%    45267.515600    0.035175  ...    0.000377    0.000358
## 50%    45267.585900    0.921455  ...    0.003088    0.002078
## 75%    45267.660200    1.229000  ...    0.008722    0.005239
## max    45267.734400    1.643600  ...    0.018593    0.012048
##
## [8 rows x 14 columns]
```

It can be helpful to turn it sideways.

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

```
##          count      mean  ...      75%      max
## time_number  12737.0  45267.587726  ...  45267.660200  45267.734400
## C1H302      12548.0    0.711176  ...    1.229000    1.643600
## C3H701      12548.0    1.186239  ...    2.079725    6.185900
## C2H502      12548.0    0.272310  ...    0.412623    4.779700
## C7H1102     12548.0    0.066836  ...    0.124940    0.154180
## C10H17      12548.0    3.696581  ...    5.783100   17.594900
## C9H1501     12548.0    2.682968  ...    5.298050    6.089700
## C8H1502     12548.0    0.009833  ...    0.017579    0.102470
## C9H1502     12548.0    0.920174  ...    1.789400    1.902200
## C8H1303     12548.0    0.020440  ...    0.037664    0.051525
## C9H1503     12548.0    0.039420  ...    0.074049    0.092587
## C8H1304     12548.0    0.006944  ...    0.013842    0.024751
## C9H1504     12548.0    0.004631  ...    0.008722    0.018593
## C10H1704    12548.0    0.002787  ...    0.005239    0.012048
##
## [14 rows x 8 columns]
```

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).

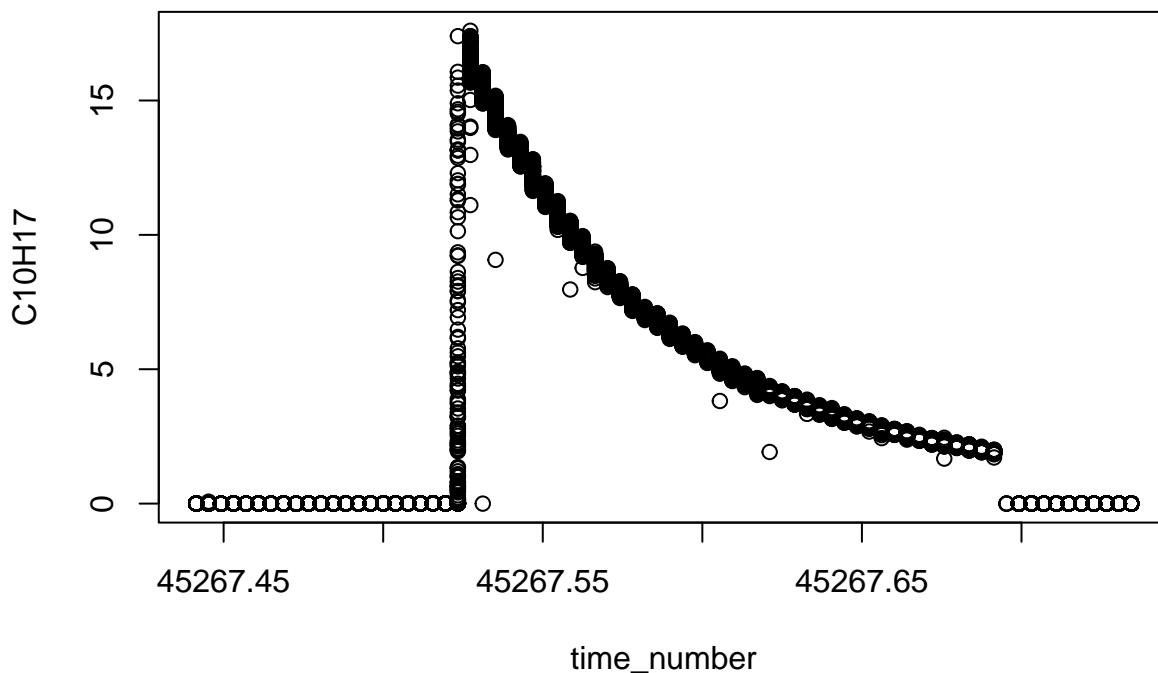
```
head(voc)
```

```
##          time_string time_number  C1H302  C3H701  C2H502  C7H1102  C10H17
## 1: 12/7/2023 10:34    45267.44  0.26983  0.23759  0.19983  0.00494390  0.0046413
```



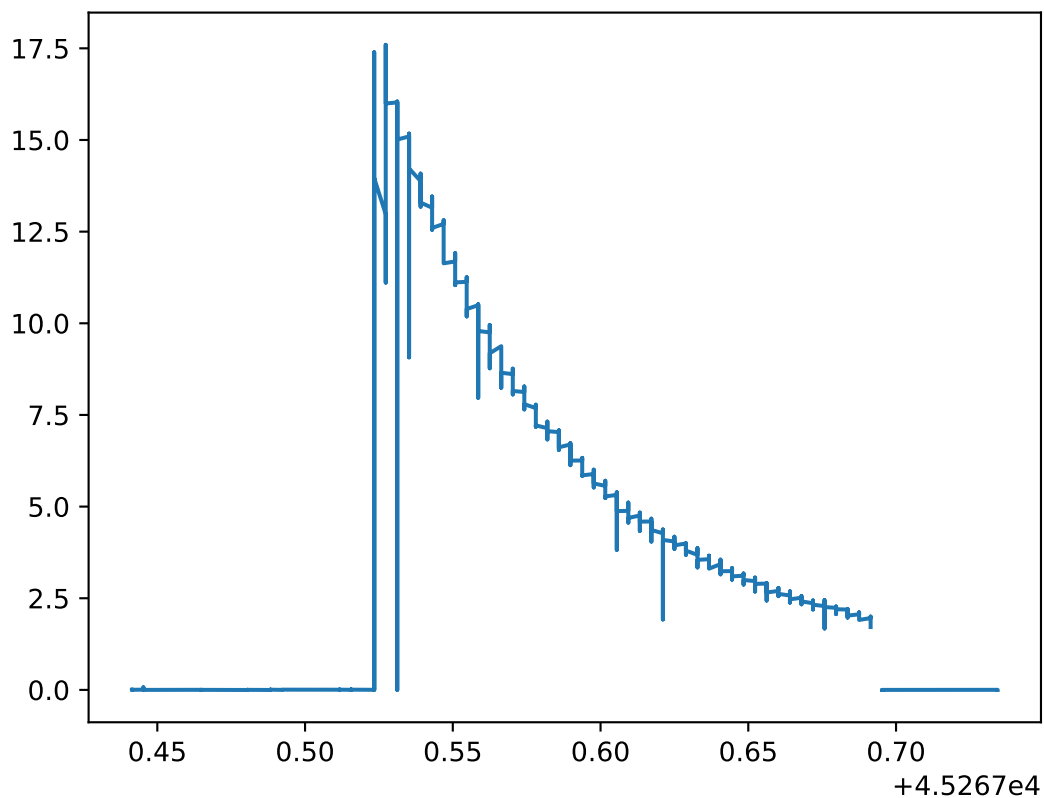
```
## 2: 12/7/2023 10:34 45267.44 0.26303 0.25205 0.18137 0.00074215 0.0012158
## 3: 12/7/2023 10:34 45267.44 0.27097 0.22796 0.19361 0.00399780 0.0021266
## 4: 12/7/2023 10:34 45267.44 0.24479 0.19712 0.17835 0.00546340 0.0026745
## 5: 12/7/2023 10:34 45267.44 0.28258 0.23840 0.18143 0.00438240 0.0055961
## 6: 12/7/2023 10:34 45267.44 0.18797 0.22651 0.18668 0.00426480 0.0053425
##      C9H1501      C8H1502      C9H1502      C8H1303      C9H1503      C8H1304      C9H1504
## 1: 0.032477 0.00222460 0.017653 0.0015724 0.00074916 -0.00032970 0.00033354
## 2: 0.035656 0.00027488 0.017610 0.0000506 0.00132800 0.00002640 0.00030222
## 3: 0.038066 -0.00016937 0.019828 0.0011651 0.00235960 0.00037222 0.00008730
## 4: 0.038752 0.00192130 0.024411 0.0020600 0.00228380 -0.00026098 -0.00073415
## 5: 0.036147 -0.00021588 0.016462 0.0014995 0.00194660 -0.00074100 0.00060288
## 6: 0.032794 -0.00059051 0.017606 0.0015675 0.00017333 -0.00062942 0.00020794
##      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.

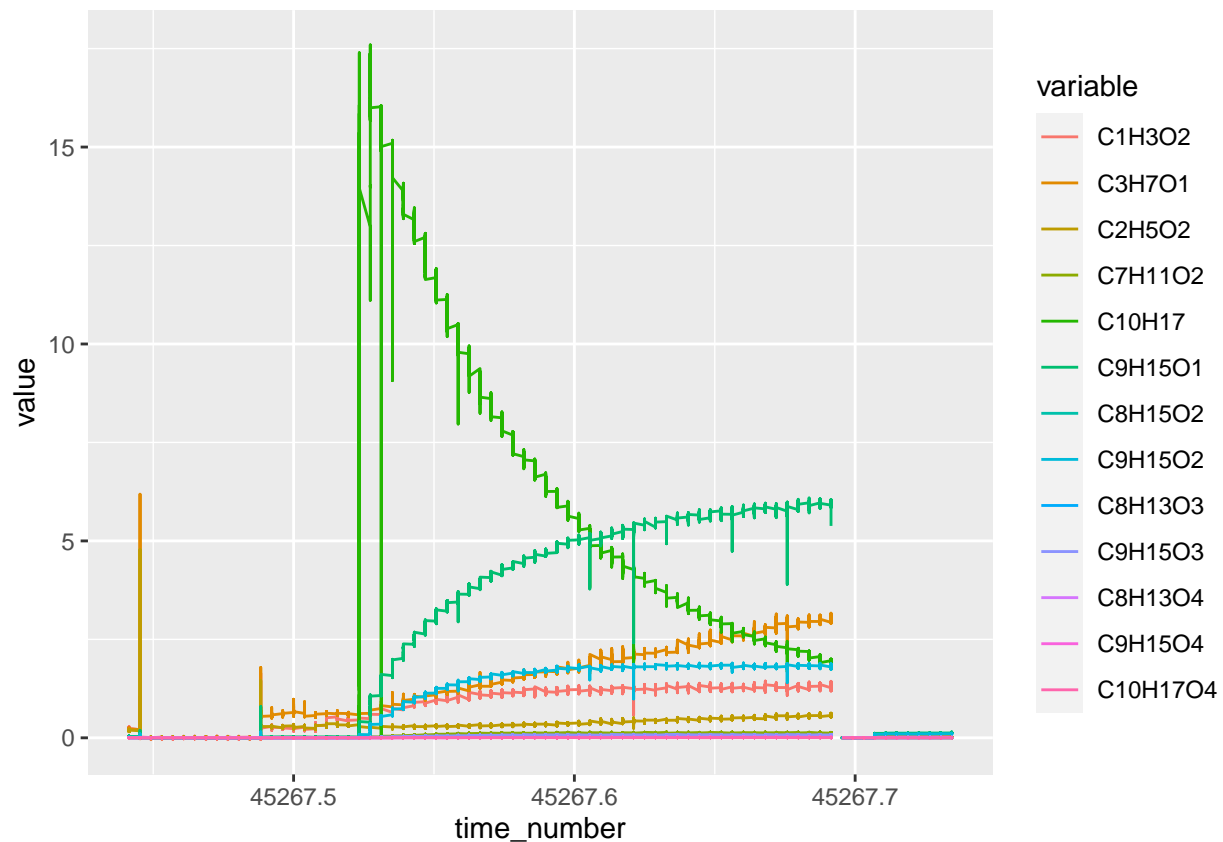
`head(voc)`

```
##      time_string time_number C1H302 C3H701 C2H502 C7H1102 C10H17
## 1: 12/7/2023 10:34 45267.44 0.26983 0.23759 0.19983 0.00494390 0.0046413
## 2: 12/7/2023 10:34 45267.44 0.26303 0.25205 0.18137 0.00074215 0.0012158
## 3: 12/7/2023 10:34 45267.44 0.27097 0.22796 0.19361 0.00399780 0.0021266
## 4: 12/7/2023 10:34 45267.44 0.24479 0.19712 0.17835 0.00546340 0.0026745
## 5: 12/7/2023 10:34 45267.44 0.28258 0.23840 0.18143 0.00438240 0.0055961
## 6: 12/7/2023 10:34 45267.44 0.18797 0.22651 0.18668 0.00426480 0.0053425
##      C9H1501 C8H1502 C9H1502 C8H1303 C9H1503 C8H1304 C9H1504
## 1: 0.032477 0.00222460 0.017653 0.0015724 0.00074916 -0.00032970 0.00033354
## 2: 0.035656 0.00027488 0.017610 0.0000506 0.00132800 0.00002640 0.00030222
## 3: 0.038066 -0.00016937 0.019828 0.0011651 0.00235960 0.00037222 0.00008730
## 4: 0.038752 0.00192130 0.024411 0.0020600 0.00228380 -0.00026098 -0.00073415
## 5: 0.036147 -0.00021588 0.016462 0.0014995 0.00194660 -0.00074100 0.00060288
## 6: 0.032794 -0.00059051 0.017606 0.0015675 0.00017333 -0.00062942 0.00020794
##      C10H1704
## 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()
```

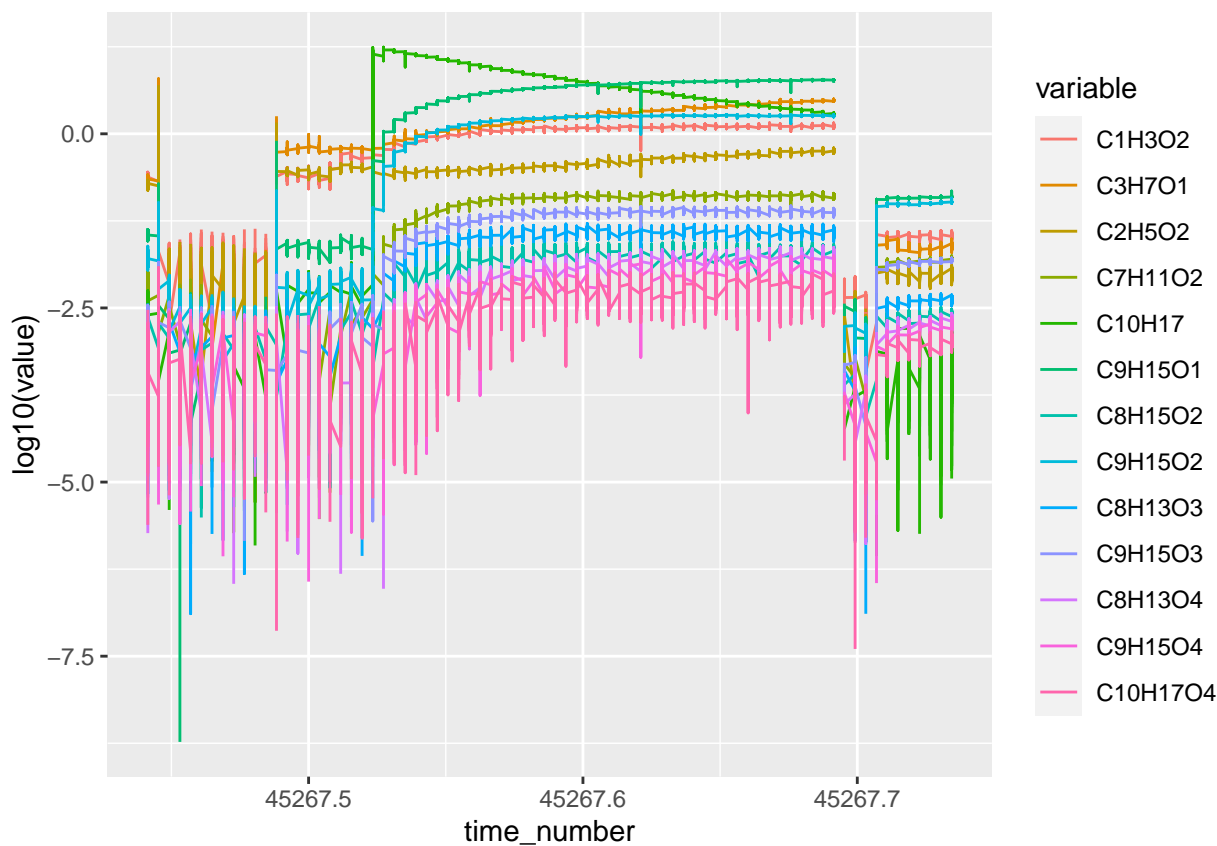


```
library(ggplot2)
ggplot(vocl, aes(time_number, log10(value), colour = variable)) +
  geom_line()
```

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

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

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

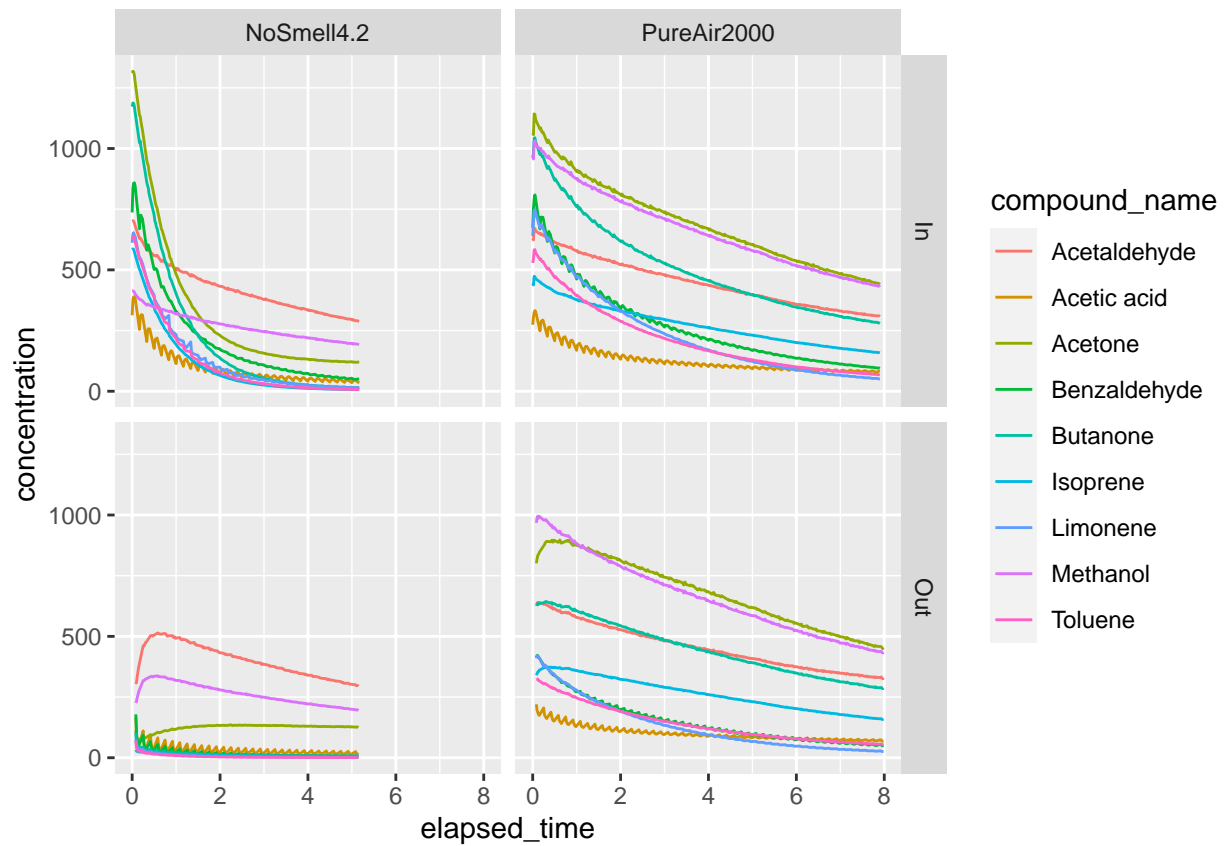


```
air <- fread('../data/air_cleaners.csv')
air
```

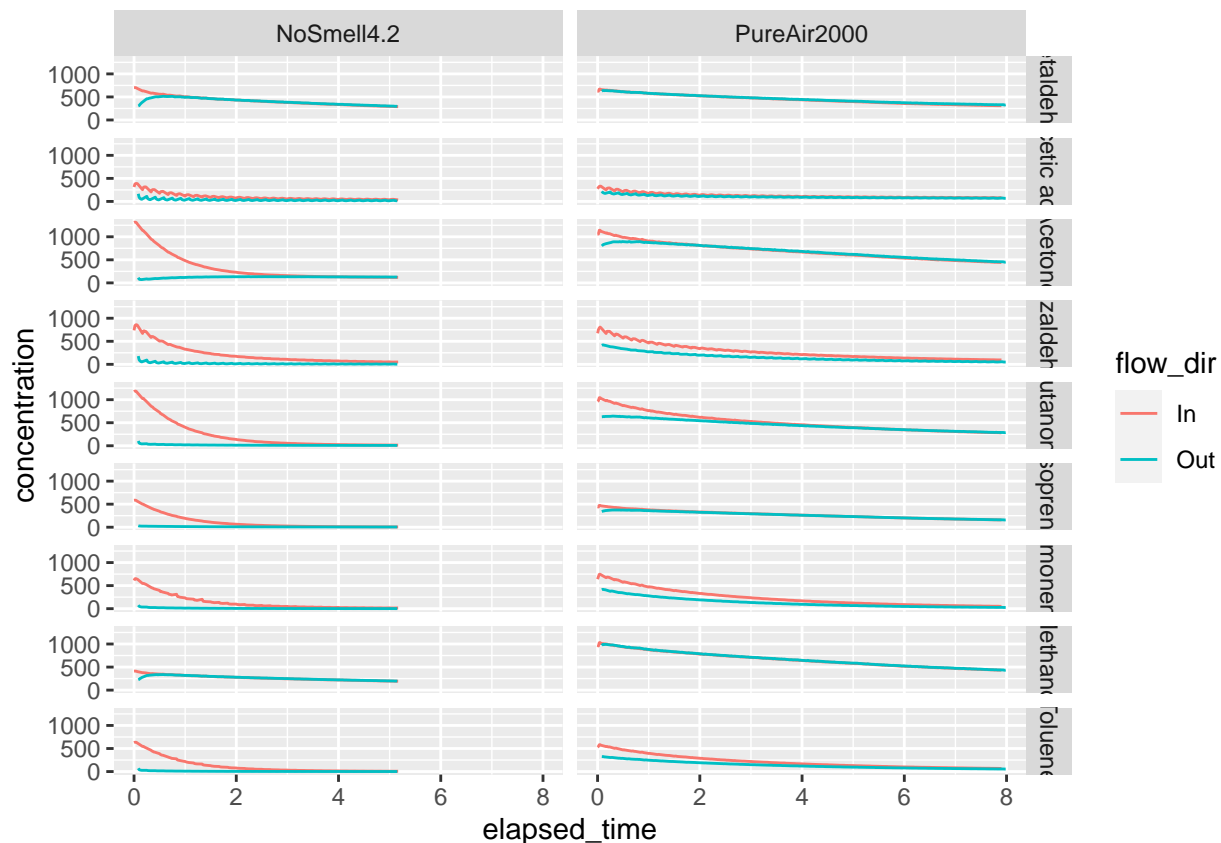
```
##      aircleaner      timestamp elapsed_time  form  mtzr compound_name
##      1: PureAir2000 3/10/2022 13:45    0.000000 CH40  32.0335   Methanol
##      2: PureAir2000 3/10/2022 13:45    0.000000 C7H8  92.0699   Toluene
##      3: PureAir2000 3/10/2022 13:45    0.000000 C5H8  68.0699   Isoprene
##      4: PureAir2000 3/10/2022 13:45    0.000000 C7H6O 106.0491 Benzaldehyde
##      5: PureAir2000 3/10/2022 13:45    0.000000 C3H6O  58.0491   Acetone
##      ---
## 7115: NoSmell4.2 3/30/2022 18:05    5.166667 CH40  32.0335   Methanol
## 7116: NoSmell4.2 3/30/2022 18:05    5.166667 C7H6O 106.0491 Benzaldehyde
## 7117: NoSmell4.2 3/30/2022 18:05    5.166667 C3H6O  58.0491   Acetone
## 7118: NoSmell4.2 3/30/2022 18:05    5.166667 C2H4O2 60.0284   Acetic acid
## 7119: NoSmell4.2 3/30/2022 18:05    5.166667 C7H8  92.0699   Toluene
##      flow_dir concentration
##      1:      In    971.682900
##      2:      In    528.657950
##      3:      In    443.841800
##      4:      In    673.093200
##      5:      In   1063.301000
##      ---
## 7115:      In    193.325450
## 7116:      In     50.615150
## 7117:      In    120.169100
## 7118:      In     33.904700
```

```
## 7119:      In      7.061408
```

```
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.

```
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
```

```
names(dat)
```

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

```
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
## Length:6          Min.   : 11.37   Min.   :210.0   Min.   : 5.00
## Class :character  1st Qu.: 17.54   1st Qu.:257.1   1st Qu.: 8.25
## Mode  :character  Median : 33.84   Median :354.9   Median :18.00
##                               Mean  : 50.19   Mean  :334.1   Mean  :18.33
##                               3rd Qu.: 72.34   3rd Qu.:404.1   3rd Qu.:28.50
##                               Max.   :124.80   Max.   :440.0   Max.   :32.00
##   gas          temp          flow
## Length:6          Min.   : 20   Min.   :0.06900
## Class :character  1st Qu.: 20   1st Qu.:0.07370
## Mode  :character  Median : 25   Median :0.07437
##                               Mean  : 25   Mean  :0.07623
##                               3rd Qu.: 30   3rd Qu.:0.08019
##                               Max.   : 30   Max.   :0.08400
```

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
##  1: 1947    1.73    1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00
##  2: 1947    3.46    3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00
##  3: 1947    5.19    5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00
##  4: 1947    6.92    6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00
##  5: 1947    8.65    8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00
## ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00
##      j_NH3
##  1: 0.0088216
##  2: 0.0000000
##  3: 0.0061700
##  4: 0.0136090
##  5: 0.0154260
## ---
## 3485: 0.0100490
## 3486: 0.0098460
## 3487: 0.0095709
## 3488: 0.0099536
## 3489: 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
## 1: 1947 tank 2020-11-18 97.3 3.9108 0.040193 1 1.73
## 2: 1947 tank 2020-11-18 97.3 3.9108 0.040193 1 3.46
## 3: 1947 tank 2020-11-18 97.3 3.9108 0.040193 1 5.19
## 4: 1947 tank 2020-11-18 97.3 3.9108 0.040193 1 6.92
## 5: 1947 tank 2020-11-18 97.3 3.9108 0.040193 1 8.65
## ---
## 3485: 1982 none 2020-12-09 66.5 18.4340 0.277210 4 178.19
## 3486: 1982 none 2020-12-09 66.5 18.4340 0.277210 4 179.92
## 3487: 1982 none 2020-12-09 66.5 18.4340 0.277210 4 181.65
## 3488: 1982 none 2020-12-09 66.5 18.4340 0.277210 4 183.38
## 3489: 1982 none 2020-12-09 66.5 18.4340 0.277210 4 185.11
## cta dt t_start t_end j_NH3
## 1: 1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00 0.0088216
## 2: 3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00 0.0000000
## 3: 5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00 0.0061700
## 4: 6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00 0.0136090
## 5: 8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00 0.0154260
## ---
## 3485: 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00 0.0100490
## 3486: 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00 0.0098460
## 3487: 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00 0.0095709
```

```
## 3488: 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00 0.0099536
## 3489: 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00 0.0116350
```

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      ... tan_app e_cum_final e_rel_final date_int
## 0      1947      1.73      1.7333      1.73      ...      97.3      3.9108      0.040193      1
## 1      1947      3.46      3.4667      1.73      ...      97.3      3.9108      0.040193      1
## 2      1947      5.19      5.2000      1.73      ...      97.3      3.9108      0.040193      1
## 3      1947      6.92      6.9333      1.73      ...      97.3      3.9108      0.040193      1
## 4      1947      8.65      8.6667      1.73      ...      97.3      3.9108      0.040193      1
## ...      ...      ...      ...      ...      ...      ...      ...      ...      ...
## 3484    1982    178.19    178.5300    1.73      ...      66.5     18.4340     0.277210      4
## 3485    1982    179.92    180.2700    1.73      ...      66.5     18.4340     0.277210      4
## 3486    1982    181.65    182.0000    1.73      ...      66.5     18.4340     0.277210      4
## 3487    1982    183.38    183.7300    1.73      ...      66.5     18.4340     0.277210      4
## 3488    1982    185.11    185.4700    1.73      ...      66.5     18.4340     0.277210      4
##
## [3489 rows x 13 columns]
```

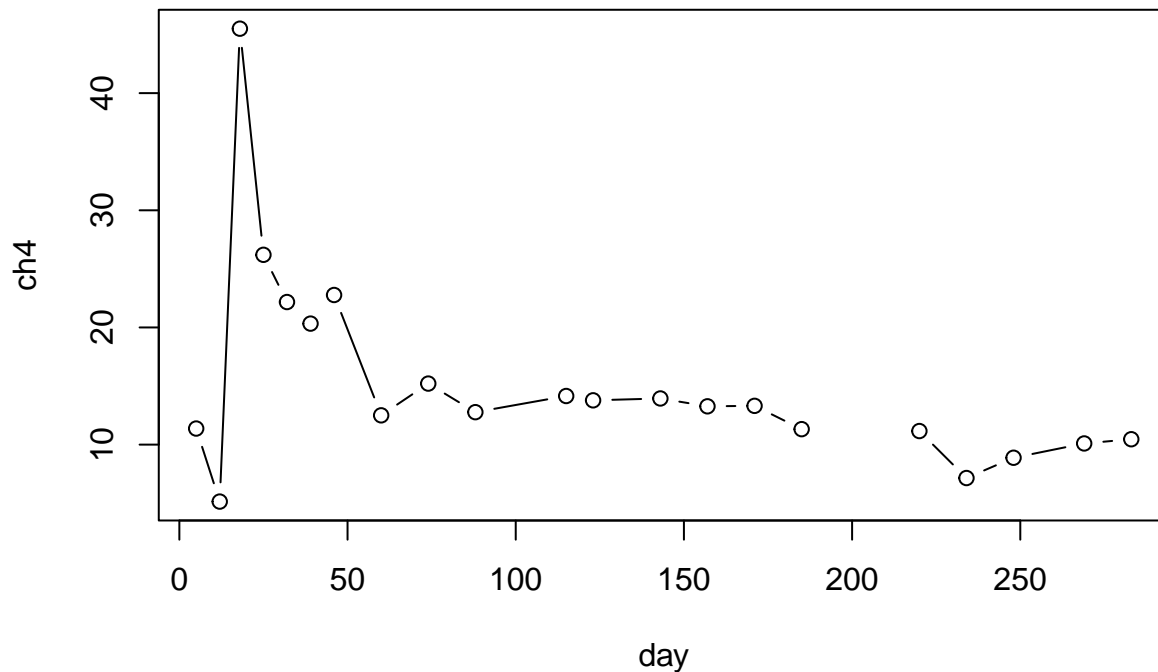
Both functions are flexible, and can merge on multiple columns, keep or drop unmatched rows, and add suffixes to columns as needed.

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.

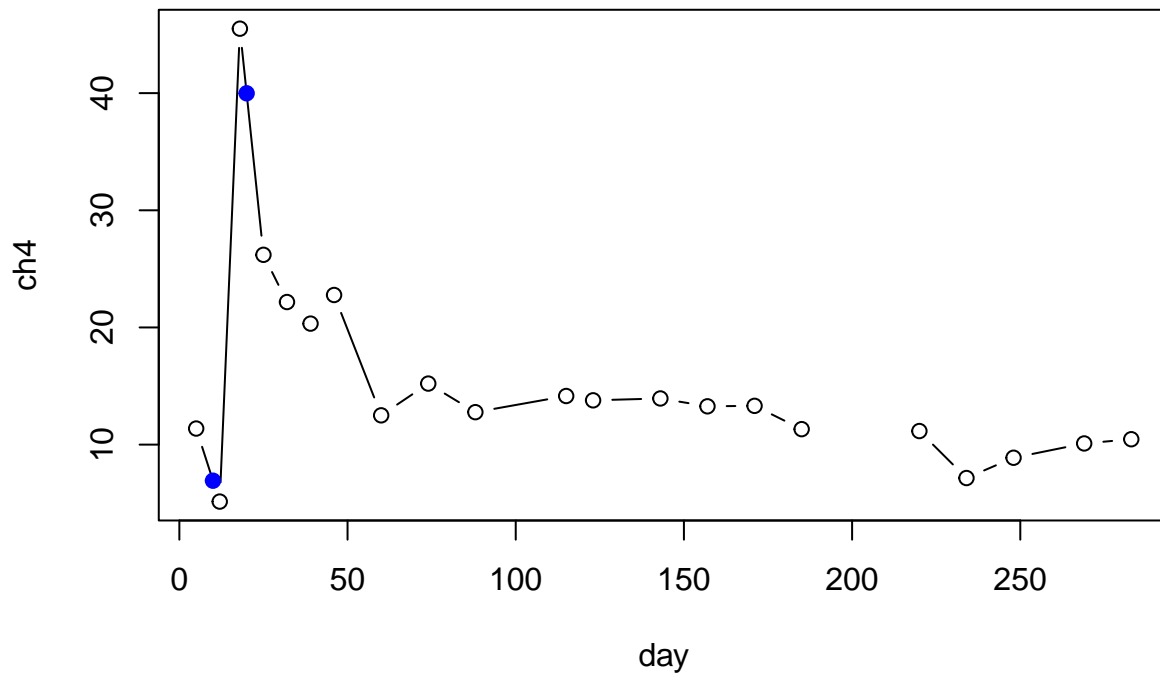
```
args(approx)
```

```
## 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)
```

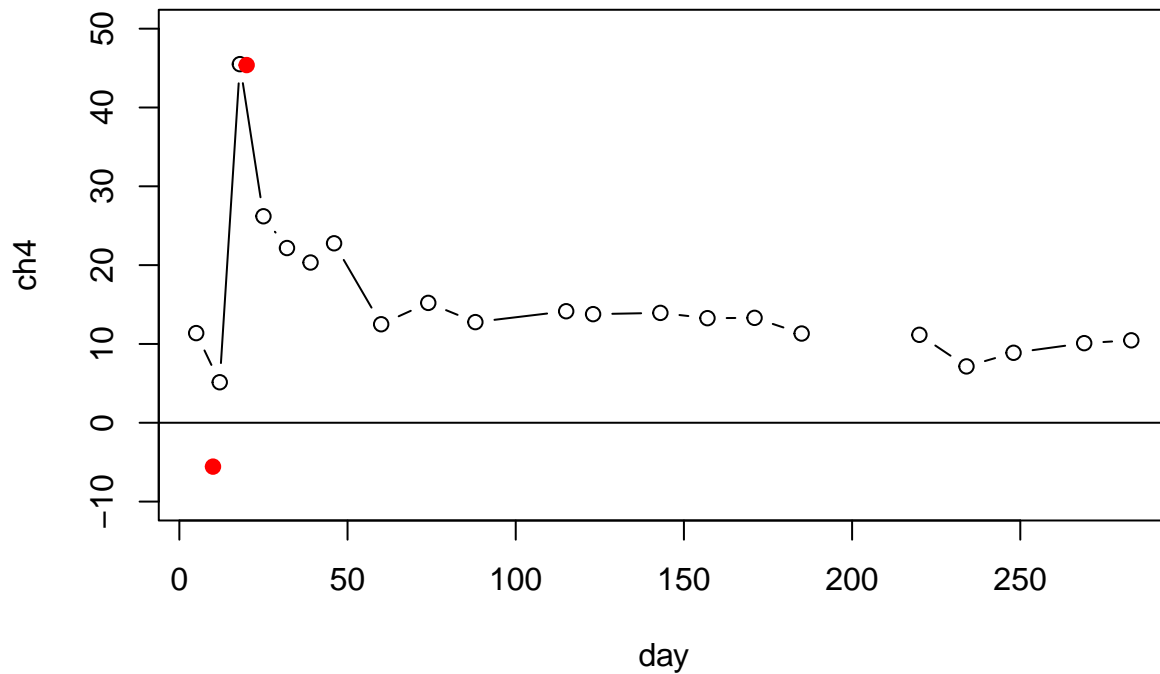


That function `approx` uses linear interpolation. There are more sophisticated methods that could be used in the `spline` function. But be sure the method is appropriate! As seen in this example, the default method is not always the most appropriate.

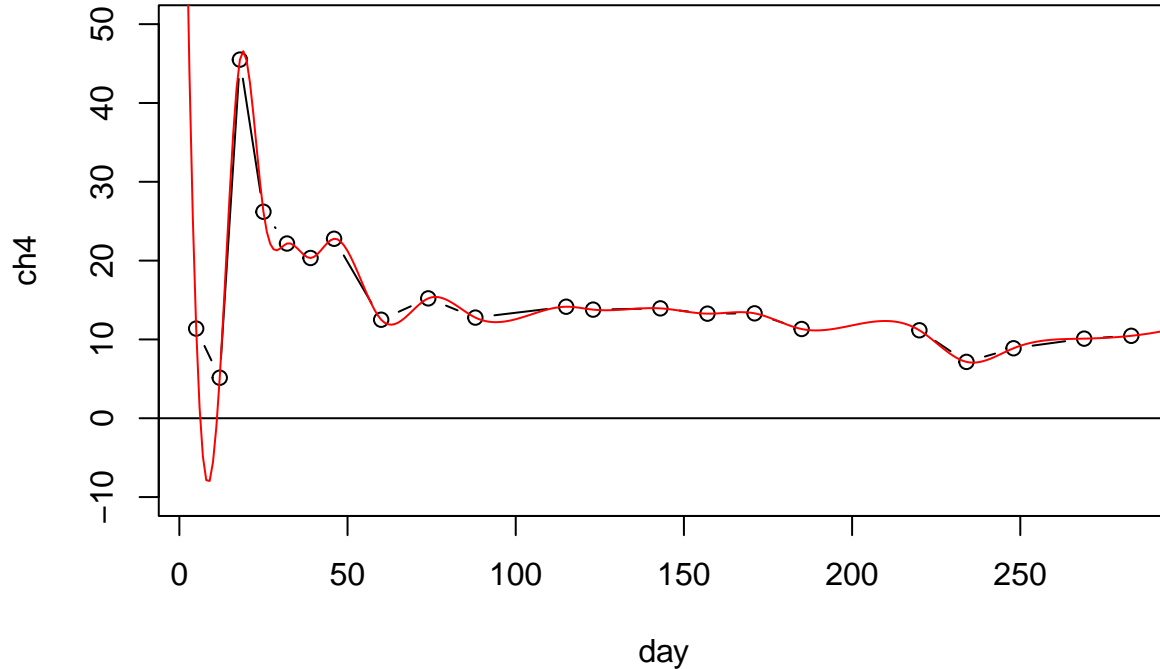
```
##?spline
args(spline)

## 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)
```



```
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:

```
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.

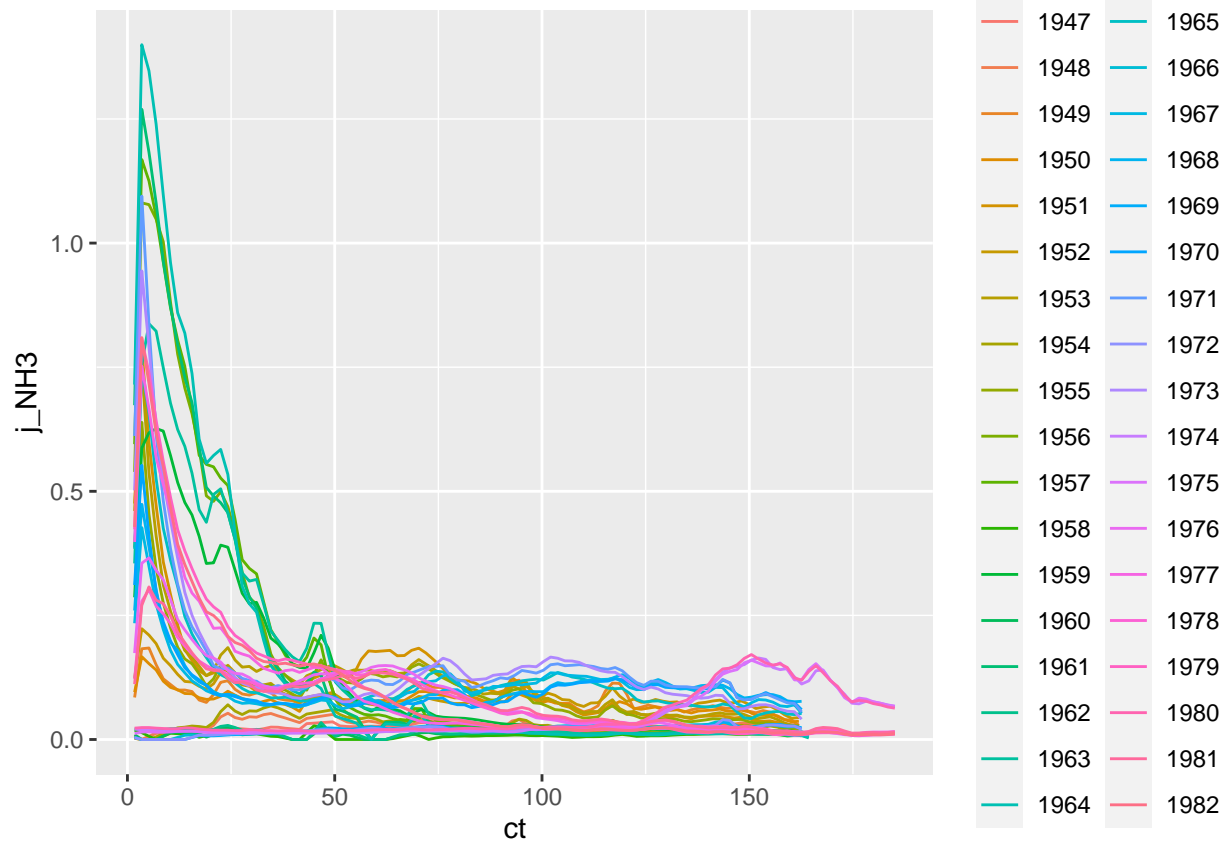
Integration

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

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



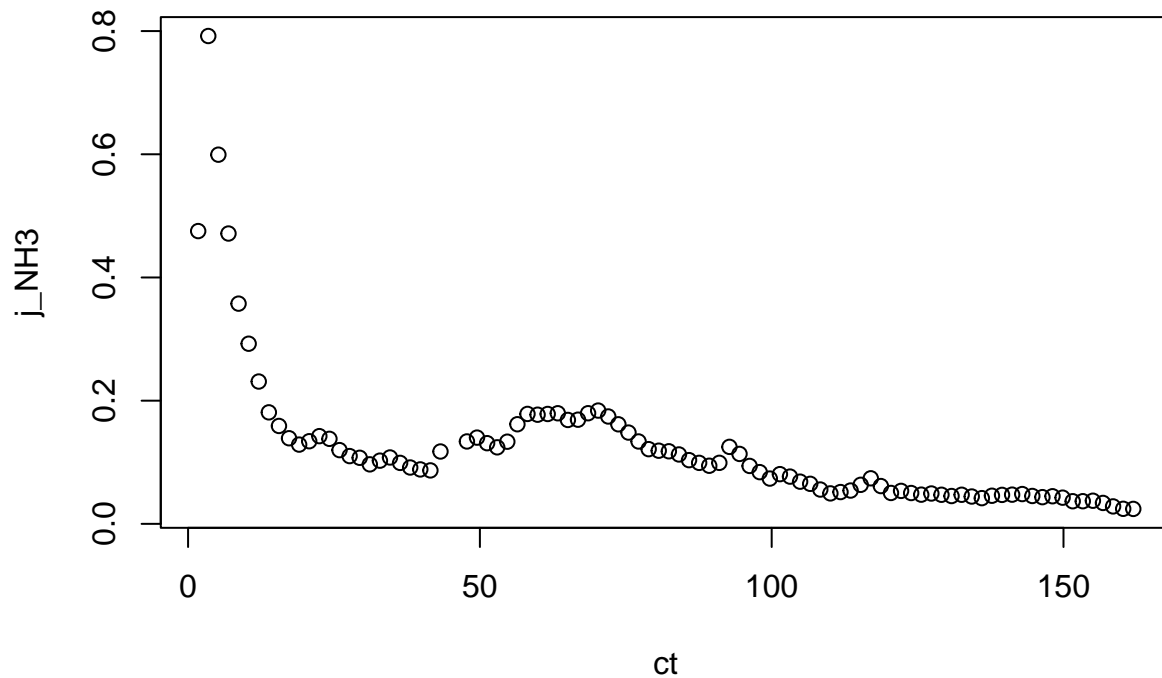
Here we have ammonia volatilization in mass of N as kg/h-ha and want kg/ha. There are some R packages with integration functions but I like a function I wrote called `mintegrate()` (for *measurement* integration, as opposed to other functions focused on function integration).

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

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

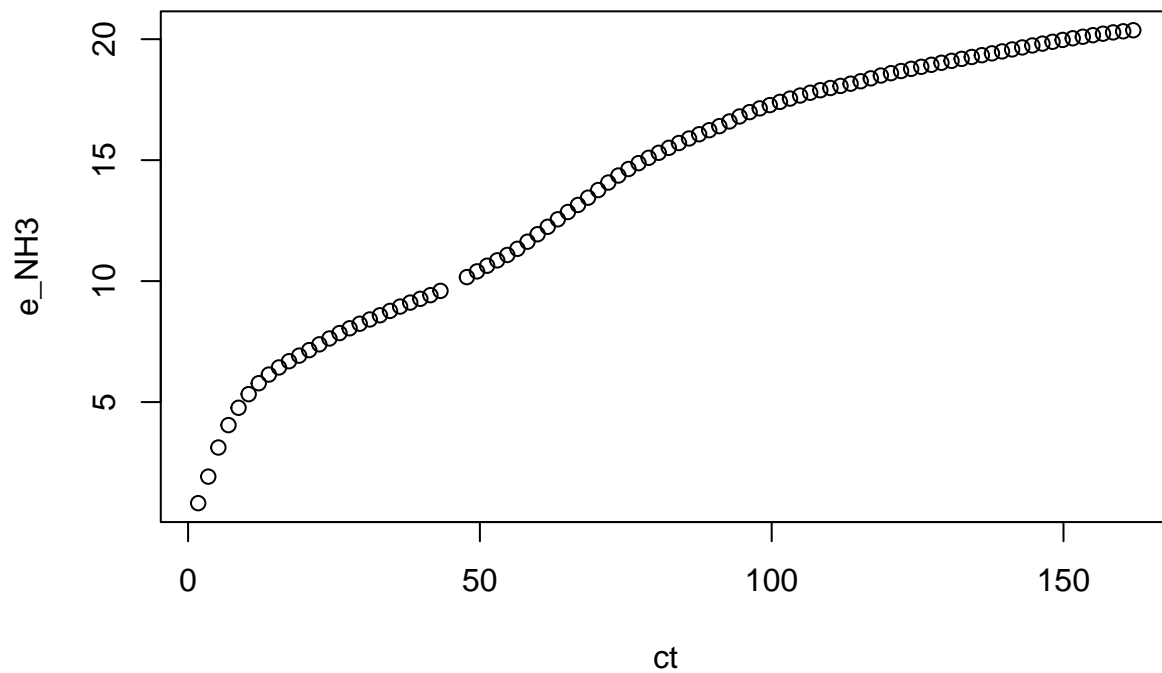
Let's apply it to a single flux curve.

```
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[, q_ch4 := flow * ch4]
dat
```

```
##      reactor      ch4   co2 day gas temp      flow      e_ch4      q_ch4
## 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
```

A few other notes about the integration follow. First, we implicitly assumed that gas flow and methane concentration were in compatible units like g m^{-3} and $\text{m}^3 \text{h}^{-1}$. In fact, that turns out to be wrong, and so the numeric `q_ch4` and `e_ch4` values are wrong. Don't follow my bad example and do something like this in your own data manipulation! Check units with pencil and paper or in comments. Second, did you notice that the first value for cumulative emission was > 0 ? This is because we set `lwr = 0`, so the first result will be the area of a rectangle from 0 to the first measurement.

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)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:data.table':
##
##   between, first, last

## The following objects are masked from 'package:stats':
##
##   filter, lag

## The following objects are masked from 'package:base':
##
##   intersect, setdiff, setequal, union

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_index()
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 interpret 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
## 1:	1947	1.73	1.7333	1.73	2020-11-18 13:40:00	2020-11-18 15:24:00
## 2:	1947	3.46	3.4667	1.73	2020-11-18 15:24:00	2020-11-18 17:08:00
## 3:	1947	5.19	5.2000	1.73	2020-11-18 17:08:00	2020-11-18 18:52:00
## 4:	1947	6.92	6.9333	1.73	2020-11-18 18:52:00	2020-11-18 20:36:00
## 5:	1947	8.65	8.6667	1.73	2020-11-18 20:36:00	2020-11-18 22:20:00
## ---						
## 3485:	1982	178.19	178.5300	1.73	2020-12-16 23:49:00	2020-12-17 01:33:00
## 3486:	1982	179.92	180.2700	1.73	2020-12-17 01:33:00	2020-12-17 03:17:00
## 3487:	1982	181.65	182.0000	1.73	2020-12-17 03:17:00	2020-12-17 05:01:00
## 3488:	1982	183.38	183.7300	1.73	2020-12-17 05:01:00	2020-12-17 06:45:00
## 3489:	1982	185.11	185.4700	1.73	2020-12-17 06:45:00	2020-12-17 08:29:00
##	j_NH3					
## 1:	0.0088216					
## 2:	0.0000000					
## 3:	0.0061700					
## 4:	0.0136090					
## 5:	0.0154260					
## ---						
## 3485:	0.0100490					
## 3486:	0.0098460					
## 3487:	0.0095709					
## 3488:	0.0099536					
## 3489:	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
## Class      integer numeric numeric numeric POSIXct, POSIXt
## Minimum      1950      1.73      1.73      1.73 2020-11-18 13:40:00
## Maximum      1980      185      7220      4.53 2020-12-17 06:53:00
## Mean          1970      85.5      1990      1.74 2020-12-02 23:50:58
## Unique (excl. NA) 36      174      508      2          3489
## Missing values 0        0        0        0          0
## Sorted        TRUE     FALSE     FALSE     FALSE     FALSE
##
##          t_end      j_NH3
## Class      POSIXct, POSIXt numeric
## Minimum      2020-11-18 15:24:00      0
## Maximum      2020-12-17 08:37:00      1.4
## Mean          2020-12-03 01:35:24 0.0867
## Unique (excl. NA)          3489      3343
## Missing values          0        0
## Sorted        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
## 1: 1947      1.73      1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00
## 2: 1947      3.46      3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00
## 3: 1947      5.19      5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00
## 4: 1947      6.92      6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00
## 5: 1947      8.65      8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00
## ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00
##          j_NH3      etime
## 1: 0.0088216      0 secs
## 2: 0.0000000     6240 secs
## 3: 0.0061700    12480 secs
## 4: 0.0136090    18720 secs
## 5: 0.0154260    24960 secs
## ---
## 3485: 0.0100490 2455740 secs
## 3486: 0.0098460 2461980 secs
## 3487: 0.0095709 2468220 secs
## 3488: 0.0099536 2474460 secs
## 3489: 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
##    1: 1947    1.73    1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00
##    2: 1947    3.46    3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00
##    3: 1947    5.19    5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00
##    4: 1947    6.92    6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00
##    5: 1947    8.65    8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00
## ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00
##      j_NH3      etime
##    1: 0.0088216      0 secs
##    2: 0.0000000    6240 secs
##    3: 0.0061700   12480 secs
##    4: 0.0136090   18720 secs
##    5: 0.0154260   24960 secs
## ---
## 3485: 0.0100490 636480 secs
## 3486: 0.0098460 642720 secs
## 3487: 0.0095709 648960 secs
## 3488: 0.0099536 655200 secs
## 3489: 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
##    1: 1947    1.73    1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00
##    2: 1947    3.46    3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00
##    3: 1947    5.19    5.2000 1.73 2020-11-18 17:08:00 2020-11-18 18:52:00
##    4: 1947    6.92    6.9333 1.73 2020-11-18 18:52:00 2020-11-18 20:36:00
##    5: 1947    8.65    8.6667 1.73 2020-11-18 20:36:00 2020-11-18 22:20:00
## ---
## 3485: 1982 178.19 178.5300 1.73 2020-12-16 23:49:00 2020-12-17 01:33:00
## 3486: 1982 179.92 180.2700 1.73 2020-12-17 01:33:00 2020-12-17 03:17:00
## 3487: 1982 181.65 182.0000 1.73 2020-12-17 03:17:00 2020-12-17 05:01:00
## 3488: 1982 183.38 183.7300 1.73 2020-12-17 05:01:00 2020-12-17 06:45:00
## 3489: 1982 185.11 185.4700 1.73 2020-12-17 06:45:00 2020-12-17 08:29:00
##      j_NH3      etime      etime2
##    1: 0.0088216      0 secs    0.000000
##    2: 0.0000000    6240 secs    1.733333
##    3: 0.0061700   12480 secs    3.466667
##    4: 0.0136090   18720 secs    5.200000
##    5: 0.0154260   24960 secs    6.933333
## ---
## 3485: 0.0100490 636480 secs 176.800000
## 3486: 0.0098460 642720 secs 178.533333
```



```
## 3487: 0.0095709 648960 secs 180.266667
## 3488: 0.0099536 655200 secs 182.000000
## 3489: 0.0116350 661440 secs 183.733333
```

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

```
##
## Attaching package: 'lubridate'
## The following objects are masked from 'package:data.table':
##
##      hour, isoweek, mday, minute, month, quarter, second, wday, week,
##      yday, year
## The following objects are masked from 'package:base':
##
##      date, intersect, setdiff, union
```

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

```
##           PMID      ct      cta      dt      t_start      t_end
##      1: 1947      1.73      1.7333 1.73 2020-11-18 13:40:00 2020-11-18 15:24:00
##      2: 1947      3.46      3.4667 1.73 2020-11-18 15:24:00 2020-11-18 17:08:00
```

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

```
dfsumm(amm_int)
```

```
##
## 3489 rows and 8 columns
## 3489 unique rows
##      pmid      ct      cta      dt      t_start
## Class      integer numeric numeric numeric      character
## Minimum      1950      1.73      1.73      1.73 2020-11-18 13:40:00
## Maximum      1980      185      7220      4.53 2020-12-17 06:53:00
## Mean      1970      85.5      1990      1.74      <NA>
## Unique (excl. NA)      36      174      508      2      3489
## Missing values      0      0      0      0      0
## Sorted      TRUE      FALSE      FALSE      FALSE      FALSE
##
##      t_end      j_NH3      date_time_start
## Class      character numeric      POSIXct, POSIXt
## Minimum      2020-11-18 15:24:00      0 2020-11-18 13:40:00
## Maximum      2020-12-17 08:37:00      1.4 2020-12-17 06:53:00
## Mean      <NA>      0.0867 2020-12-02 23:50:58
## Unique (excl. NA)      3489      3343      3489
## Missing values      0      0      0
## Sorted      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
```

```

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

```

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_smallish.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
## 7:     R12 26.700 1240.0 5 ar 20 0.05100
## 8:     R12 37.600 995.0 18 ar 20 0.05520
## 9:     R12 31.400 1028.0 32 ar 20 0.05405
## 10:    R13 81.300 2022.0 5 ar 30 0.06400
## 11:    R13 100.200 2000.0 18 ar 30 0.05750
## 12:    R13 126.600 2530.0 32 ar 30 0.05635
```

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

```
##      day 20_ar 20_co2 30_ar 30_co2
## 1:    5 26.7 11.374 81.3 16.00
## 2:   18 37.6 45.500 100.2 124.80
## 3:   32 31.4 22.170 126.6 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:     R12 ar 20 5      ch4    26.70000
## 8:     R12 ar 20 18     ch4    37.60000
## 9:     R12 ar 20 32     ch4    31.40000
## 10:    R13 ar 30 5      ch4    81.30000
## 11:    R13 ar 30 18     ch4   100.20000
## 12:    R13 ar 30 32     ch4   126.60000
## 13:     R1 co2 20 5      co2   338.30000
## 14:     R1 co2 20 18     co2   230.00000
## 15:     R1 co2 20 32     co2   210.00000
## 16:     R5 co2 30 5      co2   371.50000
## 17:     R5 co2 30 18     co2   440.00000
## 18:     R5 co2 30 32     co2   415.00000
## 19:    R12 ar 20 5      co2  1240.00000
## 20:    R12 ar 20 18     co2   995.00000
## 21:    R12 ar 20 32     co2  1028.00000
## 22:    R13 ar 30 5      co2  2022.00000
## 23:    R13 ar 30 18     co2  2000.00000
## 24:    R13 ar 30 32     co2  2530.00000
## 25:     R1 co2 20 5      flow    0.08200
## 26:     R1 co2 20 18     flow    0.08400
## 27:     R1 co2 20 32     flow    0.07400
## 28:     R5 co2 30 5      flow    0.07475
## 29:     R5 co2 30 18     flow    0.06900
```

```
## 30:      R5 co2   30  32   flow   0.07360
## 31:      R12 ar   20   5   flow   0.05100
## 32:      R12 ar   20  18   flow   0.05520
## 33:      R12 ar   20  32   flow   0.05405
## 34:      R13 ar   30   5   flow   0.06400
## 35:      R13 ar   30  18   flow   0.05750
## 36:      R13 ar   30  32   flow   0.05635
##      reactor gas temp day variable      value
```

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, use the `pivot_table()` function from the pandas module.

```
import pandas as pd
dat = pd.read_csv('../data/slurry_emis_smallish.csv')
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
## 6        R12   26.700  1240.0    5  ar    20  0.05100
## 7        R12   37.600   995.0   18  ar    20  0.05520
## 8        R12   31.400  1028.0   32  ar    20  0.05405
## 9        R13   81.300  2022.0    5  ar    30  0.06400
## 10       R13  100.200  2000.0   18  ar    30  0.05750
## 11       R13  126.600  2530.0   32  ar    30  0.05635
```

```
datwide = dat.pivot_table(index = ['day'], columns = ['temp', 'gas'], values = ['ch4'])
datwide
```

```
##      ch4
## temp    20      30
## gas     ar     co2     ar     co2
## day
## 5      26.7  11.374  81.3   16.00
## 18     37.6  45.500 100.2  124.80
## 32     31.4  22.170 126.6   81.29
```

```
datwide['ch4']
```

```
## temp    20      30
## gas     ar     co2     ar     co2
## day
## 5      26.7  11.374  81.3   16.00
## 18     37.6  45.500 100.2  124.80
## 32     31.4  22.170 126.6   81.29
```

It does seem a little difficult to use the output. The issue is the output is a “MultiIndex object”, which seems to me to be quite awkward to work with. I see a lot of discussion online about various ways to convert this to a regular data frame, but have not found a good general solution.

```
datwide.stack
```

```
## <bound method DataFrame.stack of          ch4
```

```
## temp      20      30
## gas      ar      co2      ar      co2
## day
## 5      26.7  11.374  81.3   16.00
## 18     37.6  45.500  100.2  124.80
## 32     31.4  22.170  126.6   81.29>
```

```
datwide.unstack
```

```
## <bound method DataFrame.unstack of      ch4
## temp      20      30
## gas      ar      co2      ar      co2
## day
## 5      26.7  11.374  81.3   16.00
## 18     37.6  45.500  100.2  124.80
## 32     31.4  22.170  126.6   81.29>
```

```
datwide = dat.pivot_table(index = ['day'], columns = ['temp', 'gas'], values = 'ch4')
datwide.reset_index(inplace = True)
datwide
```

```
## temp day      20      30
## gas      ar      co2      ar      co2
## 0      5  26.7  11.374  81.3   16.00
## 1     18  37.6  45.500  100.2  124.80
## 2     32  31.4  22.170  126.6   81.29
```

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.