An introduction to using gcplyr

Mike Blazanin

Contents

Getting started

`gcplyr` is a package that implements a number of functions to make it easier to import, manipulate, and analyze bacterial growth from data collected in multiwell plate readers ("growth curves"). Without `gcplyr`, importing and analyzing plate reader data can be a complicated process that has to be tailored for each experiment, requiring many lines of code. With `gcplyr` many of those steps are now just a single line of code.

This document gives a walkthrough of how to use `gcplyr`'s most common functions.

To get started, all you need is the data file with the growth curve measures saved in a tabular format (.csv, .xls, or .xlsx) to your computer.

Users often want to combine their data with some information on experimental design elements of their growth curve plate(s). For instance, this might include which strains went into which wells. You can save this information into a tabular file as well (see Reading design elements from files), or you can just keep it handy to enter it directly through a function later on (see Generating designs in R).

Let's get started by loading `gcplyr`. We're also going to load a couple packages we'll need later.

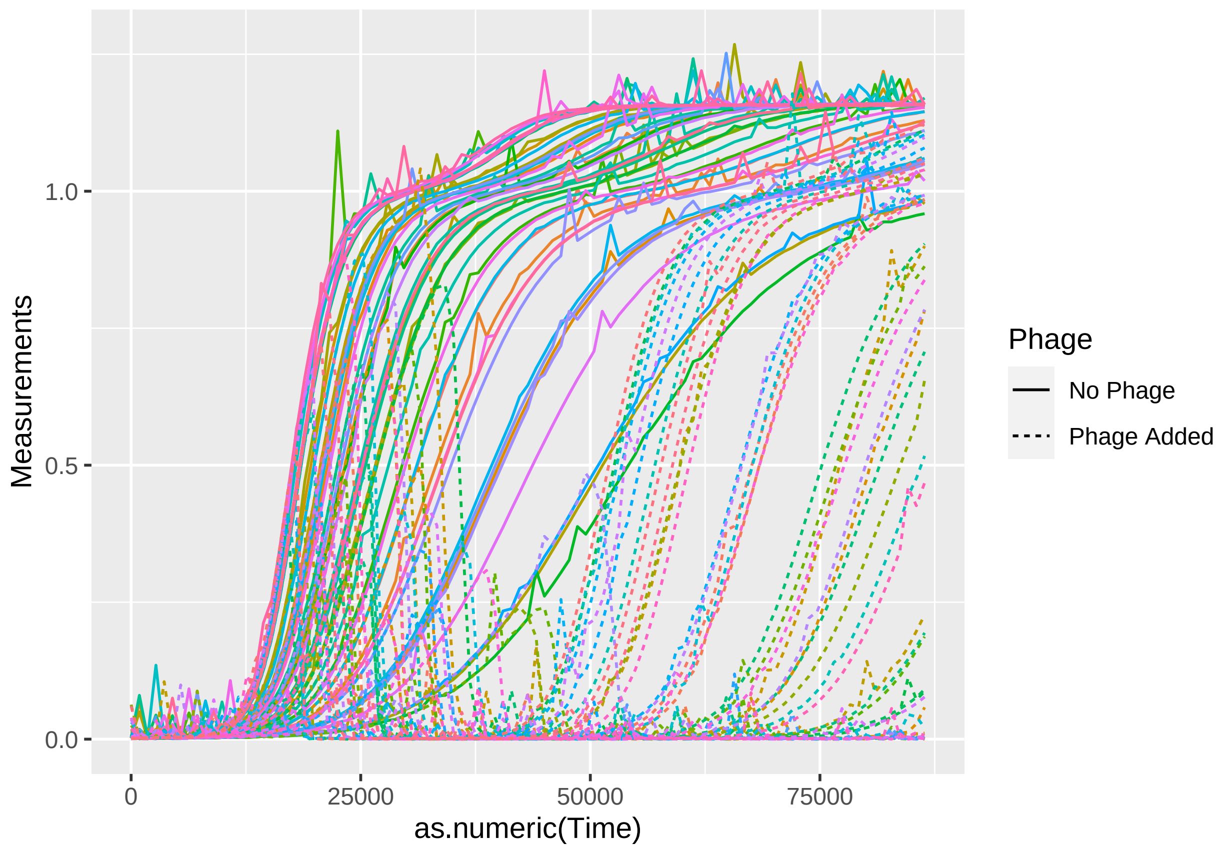
```
library(gcplyr)

library(dplyr)
library(ggplot2)
library(lubridate)
```

A quick demo of `gcplyr`

Before digging into the details of the various options that `gcplyr` provides to users, here's a simple example of what a final `gcplyr` script can look like. This script imports data from files created by a plate reader, combines it with design files created by the user, then calculates the maximum density and area-under-the-curve. **Don't worry about understanding all the details of how the code works right now.** Each of these steps is explained in depth in later sections of this document. Here, we're just providing a demonstration of what analyzing growth curve data with `gcplyr` can look like.

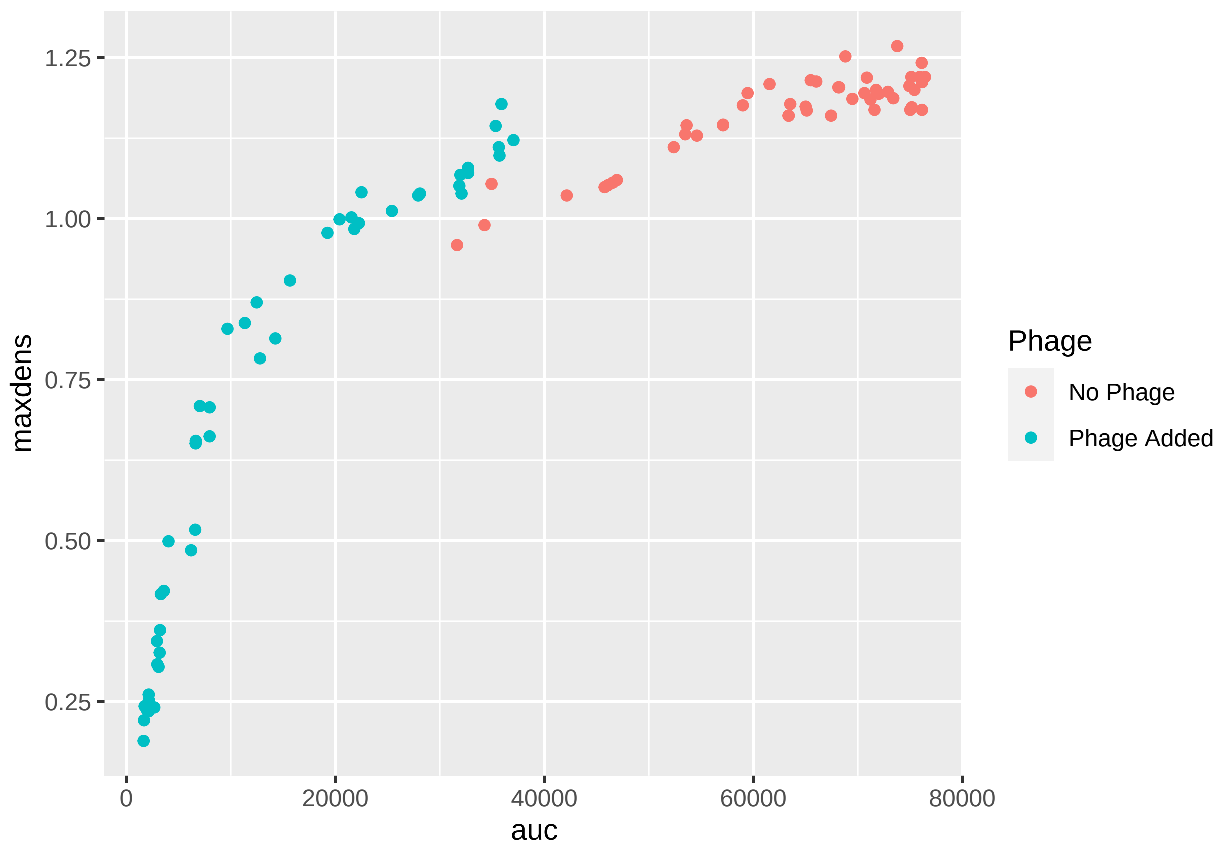
```
#Read in our data
# (our plate reader data is saved in "widedata.csv")
data_wide <- read_wides(files = "widedata.csv")

#Transform our data to be tidy-shaped
data_tidy <-
```

```
  trans_wide_to_tidy(wides = data_wide, id_cols = c("file", "Time"))

#Import our designs
# (saved in the files Bacteria_strain.csv and Phage.csv)
designs <- import_blockdesigns(files = c("Bacteria_strain.csv", "Phage.csv"))

#Merge our designs and data
data_merged <- merge_dfs(data_tidy, designs)
#> Joining, by = "Well"

#Plot the data
ggplot(data = data_merged,
       aes(x = as.numeric(Time), y = Measurements, color = Well)) +
  geom_line(aes(lty = Phage)) +
  guides(color = "none")
```




```
#Voila! 8 lines of code and all your data is imported & plotted!

#Calculate two common metrics of bacterial growth:
# the maximum density, saving it to a column named 'maxdens'
# the area-under-the-curve, saving it to a column named 'auc'
data_sum <- summarize(
  group_by(data_merged, Well, Bacteria_strain, Phage),
  maxdens = max(Measurements, na.rm = TRUE),
  auc = auc(y = Measurements, x = as.numeric(Time)))
```

```
#> `summarise()` has grouped output by 'Well', 'Bacteria_strain'. You can override
#> using the `.groups` argument.

#Print some of the max densities and auc's
head(data_sum)
#> # A tibble: 6 x 5
#> # Groups:   Well, Bacteria_strain [6]
#>   Well  Bacteria_strain Phage        maxdens    auc
#>   <chr> <chr>           <chr>          <dbl>  <dbl>
#> 1 A1    Strain 1        No Phage       1.15  57102.
#> 2 A10   Strain 4        Phage Added    0.999 20403.
#> 3 A11   Strain 5        Phage Added    0.984 21812.
#> 4 A12   Strain 6        Phage Added    0.189  1652.
#> 5 A2    Strain 2        No Phage       1.20  68206.
#> 6 A3    Strain 3        No Phage       1.13  54593.

#Plot the results for max density and area under the curve in presence vs absence of phage
ggplot(data = data_sum,
       aes(x = auc, y = maxdens, color = Phage)) +
  geom_point()
```



Data layouts

With that demonstration done, let's dig into some more details of how your input data might be organized and what `gcplyr` does. Growth curve data and design elements can be organized in one of three different tabular layouts: block-shaped, wide-shaped, and tidy-shaped, described below.

Tidy-shaped data is the best layout for analyses, but most plate readers output block-shaped or wide-shaped data, and most user-created design files will be block-shaped. Thus, `gcplyr` works by reshaping block-shaped into wide-shaped data, and wide-shaped data into tidy-shaped data, then running any analyses.

So, what are these three data layouts, and how can you tell which of them your data is in?

Block-shaped

In block-shaped data, the organization of the data corresponds directly with the layout of the physical multi-well plate it was generated from. For instance, a data point from the third row and fourth column of the `data.frame` will be from the well in the third row and fourth column in the physical plate. Because of this, a timeseries of growth curve data that is block-shaped will consist of many separate block-shaped data.frames, each corresponding to a single timepoint.

For example, here is a block-shaped `data.frame` of a 96-well plate (with "…" indicating Columns 4 - 10, not shown). In this example, all the data shown would be from a single timepoint.

	Column 1	Column 2	Column 3	…	Column 11	Column 12
Row A	0.060	0.083	0.086	…	0.082	0.085
Row B	0.099	0.069	0.065	…	0.066	0.078
Row C	0.081	0.071	0.070	…	0.064	0.084
Row D	0.094	0.075	0.065	…	0.067	0.087
Row E	0.052	0.054	0.072	…	0.079	0.065
Row F	0.087	0.095	0.091	…	0.075	0.058
Row G	0.095	0.079	0.099	…	0.063	0.075
Row H	0.056	0.069	0.070	…	0.053	0.078

Wide-shaped

In wide-shaped data, each column of the dataframe corresponds to a single well from the plate, and each row of the dataframe corresponds to a single timepoint. Typically, headers contain the well names.

For example, here is a wide-shaped dataframe of a 96-well plate (here, "…" indicates the 91 columns A4 - H10, not shown). Each row of this dataframe corresponds to a single timepoint.

Time	A1	A2	A3	…	H11	H12
0	0.060	0.083	0.086	…	0.053	0.078
1	0.012	0.166	0.172	…	0.106	0.156
2	0.024	0.332	0.344	…	0.212	0.312
3	0.048	0.664	0.688	…	0.424	0.624
4	0.096	1.128	0.976	…	0.848	1.148
5	0.162	1.256	1.152	…	1.096	1.296
6	0.181	1.292	1.204	…	1.192	1.352
7	0.197	1.324	1.288	…	1.234	1.394

Tidy-shaped

In tidy-shaped data, there is a single column that contains all the plate reader measurements, with each unique measurement having its own row. Additional columns specify the timepoint, which well the data comes from, and any other design elements.

Note that, in tidy-shaped data, the number of rows equals the number of wells times the number of timepoints. For instance, with a 96 well plate and 100 timepoints, that will be 9600 rows. (Yes, that's a lot of rows! But don't worry, tidy-shaped data is the best format for downstream analyses.) Tidy-shaped data is common in a number of `R` packages, including `ggplot`, where it's sometimes called a "long" format. If you want to read more about tidy-shaped data and why it's ideal for analyses, see: Wickham, Hadley. Tidy data. The Journal of Statistical Software, vol. 59, 2014.

Timepoint	Well	Measurement
1	A1	0.060
1	A2	0.083
1	A3	0.086
...	...	...
7	H10	1.113
7	H11	1.234
7	H12	1.394

Importing data

Once you've determined what format your data is in, you can begin importing it using the `read_*` or `import_*` functions of `gcplyr`.

If your data is block-shaped: use `import_blockmeasures` and start in the next section: **Importing block-shaped data**

If your data is wide-shaped: use `read_wides` and skip down to the **Importing wide-shaped data** section

If your data is already tidy-shaped: use `read_tidys` and skip down to the **Importing tidy-shaped data** section.

Importing block-shaped data

To import block-shaped data, use the `import_blockmeasures` function. `import_blockmeasures` only requires a list of filenames (or relative file paths) and **will return a wide-shaped data.frame** that you can save in R.

A basic example

Here's a simple example. First, we need to create a series of example block-shaped `.csv` files. **Don't worry how this code works**. When working with real growth curve data, these files would be output by the plate reader. All you need to do is put the file names in `R` in a vector, here we've stored the file names in `temp_filenames`.

```
#This code just creates a series of block-shaped example files
#Don't worry about how it works - when working with real growth
#curves data, all these files would be created by the plate reader
temp_filenames <-
  paste("Plate1-",
        paste(example_widedata$Time %/% 3600,
              formatC((example_widedata$Time %% 3600) %/% 60,
                      width = 2, flag = 0),
```

```r
              formatC((example_widedata$Time %% 3600) %% 60,
                      width = 2, flag = 0),
              sep = "_"), ".csv", sep = "")
for (i in 1:length(temp_filenames)) {
  temp_filenames[i] <- strsplit(temp_filenames[i], split = "\\\\")[[1]][
    length(strsplit(temp_filenames[i], split = "\\\\")[[1]])]
}
for (i in 1:length(temp_filenames)) {
  write.table(
    cbind(
      matrix(c("", "", "", "", "A", "B", "C", "D", "E", "F", "G", "H"),
             nrow = 12),
      rbind(rep("", 12),
            matrix(c("Time", example_widedata$Time[i], rep("", 10)), ncol = 12),
            rep("", 12),
            matrix(1:12, ncol = 12),
            matrix(
              (example_widedata[i, 2:ncol(example_widedata)]/(5*10**8)),
              ncol = 12))
    ),
    file = temp_filenames[i], quote = FALSE, row.names = FALSE, sep = ",",
    col.names = FALSE)
}
```

If you've saved all the files to a single folder, you can easily get a vector with all their names using `list.files`. If your folder contains other files, you can specify a regular expression `pattern` to limit it to just those you want to import:

```r
#Here we print all the files we're going to read
list.files(pattern = "Plate1.*csv")
#>  [1] "Plate1-0_00_00.csv"  "Plate1-0_15_00.csv"  "Plate1-0_30_00.csv"
#>  [4] "Plate1-0_45_00.csv"  "Plate1-1_00_00.csv"  "Plate1-1_15_00.csv"
#>  [7] "Plate1-1_30_00.csv"  "Plate1-1_45_00.csv"  "Plate1-10_00_00.csv"
#> [10] "Plate1-10_15_00.csv" "Plate1-10_30_00.csv" "Plate1-10_45_00.csv"
#> [13] "Plate1-11_00_00.csv" "Plate1-11_15_00.csv" "Plate1-11_30_00.csv"
#> [16] "Plate1-11_45_00.csv" "Plate1-12_00_00.csv" "Plate1-12_15_00.csv"
#> [19] "Plate1-12_30_00.csv" "Plate1-12_45_00.csv" "Plate1-13_00_00.csv"
#> [22] "Plate1-13_15_00.csv" "Plate1-13_30_00.csv" "Plate1-13_45_00.csv"
#> [25] "Plate1-14_00_00.csv" "Plate1-14_15_00.csv" "Plate1-14_30_00.csv"
#> [28] "Plate1-14_45_00.csv" "Plate1-15_00_00.csv" "Plate1-15_15_00.csv"
#> [31] "Plate1-15_30_00.csv" "Plate1-15_45_00.csv" "Plate1-16_00_00.csv"
#> [34] "Plate1-16_15_00.csv" "Plate1-16_30_00.csv" "Plate1-16_45_00.csv"
#> [37] "Plate1-17_00_00.csv" "Plate1-17_15_00.csv" "Plate1-17_30_00.csv"
#> [40] "Plate1-17_45_00.csv" "Plate1-18_00_00.csv" "Plate1-18_15_00.csv"
#> [43] "Plate1-18_30_00.csv" "Plate1-18_45_00.csv" "Plate1-19_00_00.csv"
#> [46] "Plate1-19_15_00.csv" "Plate1-19_30_00.csv" "Plate1-19_45_00.csv"
#> [49] "Plate1-2_00_00.csv"  "Plate1-2_15_00.csv"  "Plate1-2_30_00.csv"
#> [52] "Plate1-2_45_00.csv"  "Plate1-20_00_00.csv" "Plate1-20_15_00.csv"
#> [55] "Plate1-20_30_00.csv" "Plate1-20_45_00.csv" "Plate1-21_00_00.csv"
#> [58] "Plate1-21_15_00.csv" "Plate1-21_30_00.csv" "Plate1-21_45_00.csv"
#> [61] "Plate1-22_00_00.csv" "Plate1-22_15_00.csv" "Plate1-22_30_00.csv"
#> [64] "Plate1-22_45_00.csv" "Plate1-23_00_00.csv" "Plate1-23_15_00.csv"
#> [67] "Plate1-23_30_00.csv" "Plate1-23_45_00.csv" "Plate1-24_00_00.csv"
```

```
#> [70] "Plate1-3_00_00.csv"  "Plate1-3_15_00.csv"  "Plate1-3_30_00.csv"
#> [73] "Plate1-3_45_00.csv"  "Plate1-4_00_00.csv"  "Plate1-4_15_00.csv"
#> [76] "Plate1-4_30_00.csv"  "Plate1-4_45_00.csv"  "Plate1-5_00_00.csv"
#> [79] "Plate1-5_15_00.csv"  "Plate1-5_30_00.csv"  "Plate1-5_45_00.csv"
#> [82] "Plate1-6_00_00.csv"  "Plate1-6_15_00.csv"  "Plate1-6_30_00.csv"
#> [85] "Plate1-6_45_00.csv"  "Plate1-7_00_00.csv"  "Plate1-7_15_00.csv"
#> [88] "Plate1-7_30_00.csv"  "Plate1-7_45_00.csv"  "Plate1-8_00_00.csv"
#> [91] "Plate1-8_15_00.csv"  "Plate1-8_30_00.csv"  "Plate1-8_45_00.csv"
#> [94] "Plate1-9_00_00.csv"  "Plate1-9_15_00.csv"  "Plate1-9_30_00.csv"
#> [97] "Plate1-9_45_00.csv"

#Here we save them to the temp_filenames variable
temp_filenames <- list.files(pattern = "Plate1.*csv")
```

Here's what one of the files looks like (where the values are absorbance/optical density):

```
print_df(read.csv(temp_filenames[1], header = FALSE, colClasses = "character"))
#>
#>    Time       0
#>
#>        1       2     3       4       5       6       7       8        9      10      11      12
#> A 6e-12   4e-12 6e-12   6e-12   4e-12   6e-12   4e-12   4e-12    4e-12   4e-12   4e-12   4e-12
#> B 2e-12   4e-12 6e-12   4e-12   5e-11   4e-12 2.8e-11   4e-12 1.26e-10   6e-12   2e-12   6e-12
#> C 4e-12 3.4e-11 6e-12   4e-12   4e-12   2e-12   6e-12   4e-12    4e-12   4e-12   6e-12   6e-12
#> D 4e-12   2e-12 6e-12   6e-12   4e-12   4e-12   6e-12   4e-12    6e-12   4e-12   4e-12   2e-12
#> E 4e-12   4e-12 6e-12   6e-12   4e-12   4e-12   2e-12   4e-12    6e-12   2e-12 6.2e-11   6e-12
#> F 2e-12   4e-12 4e-12   2e-12   6e-12 1.4e-11   4e-12   4e-12    6e-12 2.2e-11   2e-12   4e-12
#> G 4e-12   6e-12 4e-12   6e-12 7.8e-11   6e-12   2e-12   2e-12    6e-12 7.2e-11   2e-12   6e-12
#> H 4e-12   2e-12 4e-12 3.8e-11   6e-12   6e-12   2e-12 1.2e-10    4e-12   2e-12   2e-12 3.8e-11
```

This file corresponds to all the reads for a single plate taken at the very first timepoint. We can see that the second row of the file contains some metadata about the timepoint when this plate read read was taken. Then, the data itself starts with column headers on row 4 and rownames in column 1.

If we want to read these files into `R`, we simply provide `import_blockmeasures` with the vector of file names, and save the result to some `R` object (here, `imported_blockdata`). Since our data doesn't start on the first row and column of the file, we simply need to specify what row/column it does start on using the `startrow`, `startcol`, `endrow`, and `endcol` arguments. (`import_blockmeasures` assumes that your data starts on the first row and column and ends on the last row and column, so you don't have to specify when your data meets those criteria).

```
#Now let's read it with import_blockmeasures
imported_blockdata <- import_blockmeasures(
  files = temp_filenames, startrow = 4)

head(imported_blockdata, c(6, 8))
#>       block_name       A1      A2      A3       A4      A5      A6      A7
#> 1 Plate1-0_00_00    6e-12 4.0e-12   6e-12  6.0e-12 4.0e-12 6.0e-12 4.0e-12
#> 2 Plate1-0_15_00 1.36e-10   2e-12 2.0e-12 4.00e-12   6e-12 4.0e-12   4e-12
#> 3 Plate1-0_30_00    4e-12 4.0e-12   2e-12    4e-12 2.0e-12   6e-12   6e-12
#> 4 Plate1-0_45_00    4e-12   4e-12 3.6e-11    4e-12 3.6e-11   4e-12 4.0e-12
#> 5 Plate1-1_00_00    4e-12   6e-12 3.2e-11  4.0e-12 4.0e-12   4e-12   4e-12
#> 6 Plate1-1_15_00    4e-12   4e-12   6e-12    4e-12 4.0e-12 4.0e-12   6e-12
```

Here we can see that import_blockmeasures has created a wide-shaped R object containing the data from all of our reads. It has also added the file names under the `block_name` column, so that we can easily track which row came from which file.

If you're looking at your data in Excel or a similar spreadsheet program, you'll notice that the columns aren't nicely numbered. Instead, they're coded by letter. Rather than have to count by hand what columns your data starts and ends on, just specify the column by letter and `import_blockmeasures` will translate that to a number for you! (in this example we don't have to specify a start column, since the data starts in the first column, but I do so just to show this letter-style functionality).

```
#We can specify rows or columns by Excel-style letters too
imported_blockdata <- import_blockmeasures(
  files = temp_filenames,
  startrow = 4, startcol = "A")
```

Specifying metadata

Sometimes, your input files will have information you want to import that's not included in the main block of data. For instance, with block-shaped data the timepoint is nearly always specified somewhere in the input file. `import_blockmeasures` can include that information as well via the `metadata` argument.

For example, let's return to our most-recent example files:

```
print_df(read.csv(temp_filenames[1], header = FALSE, colClasses = "character"))
#>
#>    Time       0
#>
#>        1       2     3       4       5       6       7       8        9      10      11      12
#> A 6e-12    4e-12 6e-12   6e-12   4e-12   6e-12   4e-12   4e-12    4e-12   4e-12   4e-12   4e-12
#> B 2e-12    4e-12 6e-12   4e-12   5e-11   4e-12 2.8e-11   4e-12 1.26e-10   6e-12   2e-12   6e-12
#> C 4e-12 3.4e-11 6e-12   4e-12   4e-12   2e-12   6e-12   4e-12    4e-12   4e-12   6e-12   6e-12
#> D 4e-12    2e-12 6e-12   6e-12   4e-12   4e-12   6e-12   4e-12    6e-12   4e-12   4e-12   2e-12
#> E 4e-12    4e-12 6e-12   6e-12   4e-12   4e-12   2e-12   4e-12    6e-12   2e-12 6.2e-11   6e-12
#> F 2e-12    4e-12 4e-12   2e-12   6e-12 1.4e-11   4e-12   4e-12    6e-12 2.2e-11   2e-12   4e-12
#> G 4e-12    6e-12 4e-12   6e-12 7.8e-11   6e-12   2e-12   2e-12    6e-12 7.2e-11   2e-12   6e-12
#> H 4e-12    2e-12 4e-12 3.8e-11   6e-12   6e-12   2e-12 1.2e-10    4e-12   2e-12   2e-12 3.8e-11
```

In these files, the timepoint information was located in the 2nd row and 3rd column. Here's how we could specify that metadata in our `import_blockmeasures` command:

```
#Reading the blockcurves files with metadata included
imported_blockdata <- import_blockmeasures(
  files = temp_filenames,
  startrow = 4,
  metadata = list("time" = c(2, 3)))

head(imported_blockdata, c(6, 8))
#>        block_name time       A1      A2      A3       A4      A5      A6
#> 1 Plate1-0_00_00    0    6e-12 4.0e-12   6e-12  6.0e-12 4.0e-12 6.0e-12
#> 2 Plate1-0_15_00  900 1.36e-10   2e-12 2.0e-12 4.00e-12   6e-12 4.0e-12
#> 3 Plate1-0_30_00 1800    4e-12 4.0e-12   2e-12    4e-12 2.0e-12   6e-12
#> 4 Plate1-0_45_00 2700    4e-12   4e-12 3.6e-11    4e-12 3.6e-11   4e-12
#> 5 Plate1-1_00_00 3600    4e-12   6e-12 3.2e-11  4.0e-12 4.0e-12   4e-12
#> 6 Plate1-1_15_00 4500    4e-12   4e-12   6e-12    4e-12 4.0e-12 4.0e-12
```

You can see that the metadata you specified has been added as a column in our output `data.frame`. When specifying metadata, the metadata argument must be a list of named vectors. Each vector should have two elements specifying the location of the metadata in the input files: the first element is the row, the second element is the column.

And just like how you can specify `startrow`, `startcol`, etc. with Excel-style lettering, the location of metadata can also be specified with Excel-style lettering.

```
#Reading the blockcurves files with metadata included
imported_blockdata <- import_blockmeasures(
  files = temp_filenames,
  startrow = 4,
  metadata = list("time" = c(2, "C")))
```

Reading multiple blocks from a single file

`import_blockmeasures` can also import multiple blocks from a single file, which some plate readers may output. In this case, you simply have to specify a vector of rows and columns that define the location of each block within the file.

First, let's create an example file. **Don't worry about how this code works**, normally this file would be created by the plate reader.

```
#This code just creates an example file with multiple blocks
#Don't worry about how it works - when working with real growth
#curves data, this would be created by the plate reader
write_blocks(read_blocks(files = temp_filenames,
                         startrow = 4,
                         metadata = list("time" = c(2, "C"))),
             file = "blocks_single.csv",
             output_format = "single",
             block_name_location = "file")
```

Let's take a look at what the file looks like:

```
print_df(head(read.csv("blocks_single.csv", header = FALSE,
                       colClasses = "character"),
              c(20, 8)))
#> block_name Plate1-0_00_00
#>       time              0
#>                         1       2       3       4       5       6       7
#>          A          6e-12   4e-12   6e-12   6e-12   4e-12   6e-12   4e-12
#>          B          2e-12   4e-12   6e-12   4e-12   5e-11   4e-12 2.8e-11
#>          C          4e-12 3.4e-11   6e-12   4e-12   4e-12   2e-12   6e-12
#>          D          4e-12   2e-12   6e-12   6e-12   4e-12   4e-12   6e-12
#>          E          4e-12   4e-12   6e-12   6e-12   4e-12   4e-12   2e-12
#>          F          2e-12   4e-12   4e-12   2e-12   6e-12 1.4e-11   4e-12
#>          G          4e-12   6e-12   4e-12   6e-12 7.8e-11   6e-12   2e-12
#>          H          4e-12   2e-12   4e-12 3.8e-11   6e-12   6e-12   2e-12
#>
#> block_name Plate1-0_15_00
#>       time            900
#>                         1       2       3       4       5       6       7
#>          A       1.36e-10   2e-12   2e-12   4e-12   6e-12   4e-12   4e-12
```

```
#>           B          4e-12    1e-10    4e-12 1.44e-10    6e-12    2e-12    4e-12
#>           C          4e-12    6e-12    2e-12    4e-12    4e-12 3.6e-11    2e-12
#>           D          2e-12    4e-12 1.6e-10    4e-12    2e-12    4e-12    6e-12
#>           E          4e-12    4e-12    2e-12  1.2e-11    2e-12    2e-12    4e-12
```

We can see that the first block has some metadata above it, then the block of data itself. After that there's an empty row before the next block starts. In fact, if we look at the whole file, we'll notice that all the blocks go from column 1 ("A" in Excel) to column 13 ("M" in Excel), they start on rows 3, 15, 27, 39, etc, and end on rows 11, 23, 35, 47, etc. When we look in the file, we can also see that the very last block starts on row 1155 and ends on row 1163. Let's read this information in using `import_blockmeasures` (in this example we don't *have* to specify a start column, since the data starts in the first column, but I do to be explicit):

```
imported_blockdata <- import_blockmeasures(
  "blocks_single.csv",
  startrow = seq(from = 3, to = 1155, by = 12),
  endrow = seq(from = 11, to = 1163, by = 12),
  startcol = 1, endcol = 13)
```

Here we've used the built-in `R` function `seq` to generate the full vector of `startrows` and `endrows`. If we take a look, we can see that it's been read successfully:

```
head(imported_blockdata, c(6, 8))
#>      block_name       A1      A2      A3       A4      A5      A6      A7
#> 1 blocks_single    6e-12 4.0e-12   6e-12  6.0e-12 4.0e-12 6.0e-12 4.0e-12
#> 2 blocks_single 1.36e-10   2e-12 2.0e-12 4.00e-12   6e-12 4.0e-12   4e-12
#> 3 blocks_single    4e-12 4.0e-12   2e-12    4e-12 2.0e-12   6e-12   6e-12
#> 4 blocks_single    4e-12   4e-12 3.6e-11    4e-12 3.6e-11   4e-12 4.0e-12
#> 5 blocks_single    4e-12   6e-12 3.2e-11  4.0e-12 4.0e-12   4e-12   4e-12
#> 6 blocks_single    4e-12   4e-12   6e-12    4e-12 4.0e-12 4.0e-12   6e-12
```

Now let's add some metadata. Because we're reading from a single file, we need to specify the metadata slightly differently. Instead of the metadata being a single vector `c(row,column)` with the location, it's going to be a list of two vectors, one with the row numbers, and one with the column numbers.

Going back to the file, we can see that the time of the block is saved in the second column, in rows 2, 14, 26, 38, ... through 1154.

```
imported_blockdata <- import_blockmeasures(
  "blocks_single.csv",
  startrow = seq(from = 3, to = 1155, by = 12),
  endrow = seq(from = 11, to = 1163, by = 12),
  startcol = 1, endcol = 13,
  metadata = list("time" = list(seq(from = 2, to = 1154, by = 12), 2)))
```

And now if we take a look at the resulting object, we can see that the time metadata has been incorporated.

```
head(imported_blockdata, c(6, 8))
#>      block_name time       A1      A2      A3       A4      A5      A6
#> 1 blocks_single    0    6e-12 4.0e-12   6e-12  6.0e-12 4.0e-12 6.0e-12
#> 2 blocks_single  900 1.36e-10   2e-12 2.0e-12 4.00e-12   6e-12 4.0e-12
#> 3 blocks_single 1800    4e-12 4.0e-12   2e-12    4e-12 2.0e-12   6e-12
#> 4 blocks_single 2700    4e-12   4e-12 3.6e-11    4e-12 3.6e-11   4e-12
#> 5 blocks_single 3600    4e-12   6e-12 3.2e-11  4.0e-12 4.0e-12   4e-12
#> 6 blocks_single 4500    4e-12   4e-12   6e-12    4e-12 4.0e-12 4.0e-12
```

Notes for more advanced use

Note that `import_blockmeasures` is essentially a wrapper function that calls `read_blocks`, `uninterleave`, and `trans_block_to_wide`. Any arguments for those functions can be passed to `import_blockmeasures`.

If you find yourself needing even more control over the process of importing block-shaped measures files, each of the functions is available for users to call themselves. So you can run the steps manually, first reading with `read_blocks`, separating plates as needed with `uninterleave`, then transforming to wide with `trans_block_to_wide`.

What to do next

Now that you've imported your block-shaped data, you'll need to transform it for later analyses. Jump directly to the **Transforming data** section.

Importing wide-shaped data

To import wide-shaped data, use the `read_wides` function. `read_wides` only requires a filename (or vector of filenames, or relative file paths) and will return a `data.frame` (or list of data.frames) that you can save in R.

A basic example

Here's a simple example. First, we need to create an example wide-shaped .csv file. **Don't worry how this code works**. When working with real growth curve data, these files would be output by the plate reader. All you need to do is know the file name(s) to put in you `R` code. In this example, the file name is `widedata.csv`.

```
#This code just creates a wide-shaped example file where the data doesn't
#start on the first row.
#Don't worry about how it works - when working with real growth
#curves data, this file would be created by the plate reader
temp_example_widedata <- example_widedata
colnames(temp_example_widedata) <- paste("V", 1:ncol(temp_example_widedata),
                                         sep = "")
modified_example_widedata <-
  rbind(
    as.data.frame(matrix("", nrow = 4, ncol = ncol(example_widedata))),
    colnames(example_widedata),
    temp_example_widedata)
modified_example_widedata[1:2, 1:2] <-
  c("Experiment name", "Start date", "Experiment_1", as.character(Sys.Date()))

write.table(modified_example_widedata, file = "widedata.csv",
          row.names = FALSE, col.names = FALSE, sep = ",")
write.table(modified_example_widedata, file = "widedata2.csv",
          row.names = FALSE, col.names = FALSE, sep = ",")
```

Here's what the start of the file looks like (where the values are absorbance/optical density):

```
#Let's take a peek at what this file looks like
print_df(head(read.csv("widedata.csv", header = FALSE,
                        colClasses = "character"),
              c(10, 10)))
#> Experiment name Experiment_1
#>      Start date   2022-11-10
#>
#>
#>            Time           A1    B1    C1    D1    E1    F1    G1    H1    A2
#>               0        0.003 0.001 0.002 0.002 0.002 0.001 0.002 0.002 0.002
#>             900        0.068 0.002 0.002 0.001 0.002 0.002 0.001 0.002 0.001
#>            1800        0.002 0.002 0.002 0.003 0.002 0.002 0.003 0.001 0.002
#>            2700        0.002 0.003 0.003 0.044 0.135 0.002 0.003 0.012 0.002
#>            3600        0.002 0.002 0.003 0.002 0.002 0.002 0.003 0.003 0.003
```

This file contains all the reads for a single plate taken across all timepoints. We can see that the first two rows contain some metadata saved by the plate reader, like the name of the experiment and the date of the experiment. Then, we can see that the data starts on the 5th row with a header. The first column contains the timepoint information, and each subsequent column corresponds to a well in the plate.

If we want to read this file into `R`, we simply provide `read_wides` with the file name, and save the result to some `R` object (here, `imported_widedata`). Since our data doesn't start on the first row and column of the file, we simply need to specify what row/column it does start on using the `startrow`, `startcol`, `endrow`, and `endcol` arguments. (`read_wides` assumes that your data starts on the first row and column and ends on the last row and column, so you don't have to specify when you data meets those criteria. Also note that `header = TRUE` by default').

```
imported_widedata <- read_wides(files = "widedata.csv", startrow = 5)
```

The resulting `data.frame` looks like this:

```
head(imported_widedata, c(6, 10))
#>        file Time    A1    B1    C1    D1    E1    F1    G1    H1
#> 6  widedata    0 0.003 0.001 0.002 0.002 0.002 0.001 0.002 0.002
#> 7  widedata  900 0.068 0.002 0.002 0.001 0.002 0.002 0.001 0.002
#> 8  widedata 1800 0.002 0.002 0.002 0.003 0.002 0.002 0.003 0.001
#> 9  widedata 2700 0.002 0.003 0.003 0.044 0.135 0.002 0.003 0.012
#> 10 widedata 3600 0.002 0.002 0.003 0.002 0.002 0.002 0.003 0.003
#> 11 widedata 4500 0.002 0.003 0.002 0.043 0.017 0.001 0.002 0.003
```

Note that *read_wides automatically saves the filename* the data was imported from into the first column of the output `data.frame`. This is done to ensure that later on, `data.frames` from multiple plates can be combined without fear of losing the identity of each plate.

If you're looking at your data in Excel or a similar spreadsheet program, you'll notice that the columns aren't nicely numbered. Instead, they're coded by letter. Rather than have to count by hand what columns your data starts and ends on, just specify the column by letter and `read_wides` will translate that to a number for you! (in this example we don't have to specify a start column, since the data starts in the first column, but I do so just to show this letter-style functionality).

```
imported_widedata <- read_wides(files = "widedata.csv",
                                startrow = 5, startcol = "A")
```

Note that if you have multiple files you'd like to read in, you can do so directly with a single `read_wides` command. In this case, read_wides will return a list containing all the data.frames:

```
#If we had multiple wide-shaped data files to import
imported_widedata <- read_wides(files = c("widedata.csv", "widedata2.csv"))
```

Specifying metadata

Sometimes, your input files will have information you want to import that's not included in the main block of data. For instance, many readers will output information like the experiment name and date into a header in the file. `read_wides` can include that information as well via the `metadata` argument.

The `metadata` argument should be a list of named vectors. Each vector should be of length 2, with the first entry specifying the row and the second entry specifying the column where the metadata is located.

For example, in our previous example files, the experiment name was located in the 2nd row, 2nd column, and the start date was located in the 3rd row, 2nd column. Here's how we could specify that metadata:

```
imported_widedata <- read_wides(files = "widedata.csv",
                                startrow = 5,
                                metadata = list("experiment_name" = c(1, 2),
                                                "start_date" = c(2, 2)))
head(imported_widedata, c(6, 8))
#>        file experiment_name start_date Time    A1    B1    C1    D1
#> 6  widedata    Experiment_1 2022-11-10    0 0.003 0.001 0.002 0.002
#> 7  widedata    Experiment_1 2022-11-10  900 0.068 0.002 0.002 0.001
#> 8  widedata    Experiment_1 2022-11-10 1800 0.002 0.002 0.002 0.003
#> 9  widedata    Experiment_1 2022-11-10 2700 0.002 0.003 0.003 0.044
#> 10 widedata    Experiment_1 2022-11-10 3600 0.002 0.002 0.003 0.002
#> 11 widedata    Experiment_1 2022-11-10 4500 0.002 0.003 0.002 0.043
```

And just like how you can specify `startrow`, `startcol`, etc. with Excel-style lettering, the location of metadata can also be specified with Excel-style lettering.

```
imported_widedata <- read_wides(files = "widedata.csv",
                                startrow = 5,
                                metadata = list("experiment_name" = c(1, "B"),
                                                "start_date" = c(2, "B")))
```

Reading multiple wides from a single file

In the rare case that you have multiple wide-shaped datasets saved into a single file, `read_wides` can import that as well. Refer to the earlier section **Reading multiple blocks from a single file**, since the syntax for such operations is the same for `read_wides` as it is for `import_blockmeasures`.

What to do next

Now that you've imported your wide-shaped data, you'll need to transform it for later analyses. Continue on to the **Transforming data** section.

Importing tidy-shaped data

To import tidy-shaped data, you could use the built-in `R` functions like `read.table`. However, if you need a few more options, you can use the `gcplyr` function `read_tidys`. Unlike the built-in option, `read_tidys` can import multiple tidy-shaped files at once, can add the filename as a column in the resulting data.frame, and can handle files where the tidy-shaped information doesn't start on the first row and column.

`read_tidys` only requires a filename (or vector of filenames, or relative file paths) and will return a `data.frame` (or list of data.frames) that you can save in R.

If you've read in your tidy-shaped data, you won't need to transform it, so you can skip down to the **Including design elements** section.

Transforming data

Now that you've gotten your data into the `R` environment, we need to transform it before we can do analyses. To reiterate, this is necessary because most plate readers that generate growth curve data outputs it in block-shaped or wide-shaped files, but tidy-shaped `data.frame`s are the best shape for analyses and required by gcplyr.

You can transform your `data.frame`s using the `trans_*` functions in `gcplyr`.

Transforming from wide-shaped to tidy-shaped

If the data you've read into theRenvironment is wide-shaped (or you've gotten wide-shaped data by transforming your originally block-shaped data), you'll transform it to tidy-shaped using `trans_wide_to_tidy`.

First, you need to provide `trans_wide_to_tidy` with theRobject created by `read_wides` or by `trans_block_to_wide`.

Then, you have to specify one of: * the columns your data (the spectrophotometric measures) are in via `data_cols` * what columns your non-data (e.g. time and other information) are in via `id_cols`

```
imported_blocks_now_tidy <- trans_wide_to_tidy(
  wides = imported_blockdata,
  id_cols = c("block_name", "time"))

imported_wides_now_tidy <- trans_wide_to_tidy(
  wides = imported_widedata,
  id_cols = c("file", "experiment_name", "start_date", "Time"))

print(head(imported_blocks_now_tidy), row.names = FALSE)
#>     block_name time Well Measurements
#>  blocks_single    0   A1        6e-12
#>  blocks_single    0   A2        4e-12
#>  blocks_single    0   A3        6e-12
#>  blocks_single    0   A4        6e-12
#>  blocks_single    0   A5        4e-12
#>  blocks_single    0   A6        6e-12
```

Including design elements

During analysis of growth curve data, we often want to incorporate information about the experimental design. For example, which bacteria are present in which wells, or which wells have received certain treatments. `gcplyr` enables incorporation of design elements in two ways:

1. Design elements can be imported from files
2. Design elements can be generated programmatically using `make_design`

Reading design elements from files

Users have two options for how to read design elements from files, depending on the shape of the design files that they have created:

- If design files are block-shaped, they can be read with `import_blockdesigns`
- If design files are tidy-shaped, they can simply be read with `read_tidys`

Importing block-shaped design files

To import block-shaped design files, you can use the `import_blockdesigns` function, which will return a tidy-shaped designs data frame (or list of data frames).

`import_blockdesigns` only requires a list of filenames (or relative file paths) and will return a data.frame (or list of data frames) in a **tidy format** that you can save in R. That's right, it reads in block-shaped designs but returns a tidy-shaped data frame!

A basic example Let's take a look at an example. First, we need to create an example file for the sake of this tutorial. **Don't worry how the below code works**, just imagine that you've created this file in Excel.

```
write.csv(
  file = "mydesign.csv",
  x = matrix(rep(c("Tr1", "Tr2"), each = 48),
             nrow = 8, ncol = 12, dimnames = list(LETTERS[1:8], 1:12)))
```

Now let's take a look at what the file looks like:

```
print_df(read.csv("mydesign.csv", header = FALSE, colClasses = "character"))
#>     1   2   3   4   5   6   7   8   9  10  11  12
#> A Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
#> B Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
#> C Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
#> D Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
#> E Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
#> F Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
#> G Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
#> H Tr1 Tr1 Tr1 Tr1 Tr1 Tr1 Tr2 Tr2 Tr2 Tr2 Tr2 Tr2
```

Here we can see that our design has Treatment 1 on the left-hand side of the plate (wells in columns 1 through 6), and Treatment 2 on the right-hand side of the plate (wells in columns 7 through 12). Let's import this design using `import_blockdesigns`. Since this block contains the treatment numbers, we've given the `block_names` as "Treatment_numbers". If no `block_names` is provided, `import_blockdesigns` will automatically name it according to the file name.

```
my_design <- import_blockdesigns(files = "mydesign.csv",
                                 block_names = "Treatment_numbers")
head(my_design, 20)
#>    Well Treatment_numbers
#> 1    A1               Tr1
#> 2    A2               Tr1
#> 3    A3               Tr1
#> 4    A4               Tr1
#> 5    A5               Tr1
#> 6    A6               Tr1
#> 7    A7               Tr2
#> 8    A8               Tr2
#> 9    A9               Tr2
#> 10  A10               Tr2
#> 11  A11               Tr2
#> 12  A12               Tr2
#> 13   B1               Tr1
#> 14   B2               Tr1
#> 15   B3               Tr1
#> 16   B4               Tr1
#> 17   B5               Tr1
#> 18   B6               Tr1
#> 19   B7               Tr2
#> 20   B8               Tr2
```

Importing multiple block-shaped design elements What do you do if you have multiple design components? For instance, what if you have several different bacterial strains each with several different treatments? In that case, simply save each design component as a separate file, and import them all in one go with `import_blockdesigns`.

First, let's create another example designs file. Again, **don't worry how the below code works**, just imagine that you've created this file in Excel.

```
write.csv(
  file = "mydesign2.csv",
  x = matrix(rep(c("StrA", "StrB", "StrC", "StrD"), each = 24),
             nrow = 8, ncol = 12, dimnames = list(LETTERS[1:8], 1:12),
             byrow = TRUE))
```

Now let's take a look at what the file looks like:

```
print_df(read.csv("mydesign2.csv", header = FALSE, colClasses = "character"))
#>      1    2    3    4    5    6    7    8    9   10   11   12
#> A StrA StrA StrA StrA StrA StrA StrA StrA StrA StrA StrA StrA
#> B StrA StrA StrA StrA StrA StrA StrA StrA StrA StrA StrA StrA
#> C StrB StrB StrB StrB StrB StrB StrB StrB StrB StrB StrB StrB
#> D StrB StrB StrB StrB StrB StrB StrB StrB StrB StrB StrB StrB
#> E StrC StrC StrC StrC StrC StrC StrC StrC StrC StrC StrC StrC
#> F StrC StrC StrC StrC StrC StrC StrC StrC StrC StrC StrC StrC
#> G StrD StrD StrD StrD StrD StrD StrD StrD StrD StrD StrD StrD
#> H StrD StrD StrD StrD StrD StrD StrD StrD StrD StrD StrD StrD
```

Here we can see that our design has Strain A in the first two rows, Strain B in the next two rows, and so on.

Let's now import both designs using `import_blockdesigns`. Since our two blocks contain the treatment numbers and then the strain letters, we've given the `block_names` as `c("Treatment_numbers", "Strain_letters")`. If no `block_names` is provided, `import_blockdesigns` will automatically name it according to the file name.

```
my_design <-
  import_blockdesigns(files = c("mydesign.csv", "mydesign2.csv"),
                      block_names = c("Treatment_numbers", "Strain_letters"))
head(my_design, 20)
#>    Well Treatment_numbers Strain_letters
#> 1    A1               Tr1           StrA
#> 2    A2               Tr1           StrA
#> 3    A3               Tr1           StrA
#> 4    A4               Tr1           StrA
#> 5    A5               Tr1           StrA
#> 6    A6               Tr1           StrA
#> 7    A7               Tr2           StrA
#> 8    A8               Tr2           StrA
#> 9    A9               Tr2           StrA
#> 10  A10               Tr2           StrA
#> 11  A11               Tr2           StrA
#> 12  A12               Tr2           StrA
#> 13   B1               Tr1           StrA
#> 14   B2               Tr1           StrA
#> 15   B3               Tr1           StrA
#> 16   B4               Tr1           StrA
#> 17   B5               Tr1           StrA
#> 18   B6               Tr1           StrA
#> 19   B7               Tr2           StrA
#> 20   B8               Tr2           StrA
```

Notes for more advanced use Note that `import_blockdesigns` is essentially a wrapper function that calls `read_blocks`, `paste_blocks`, `trans_block_to_wide`, `trans_wide_to_tidy`, and then `separate_tidys`. Any arguments for those functions can be passed to `import_blockdesigns`.

For instance, if your design files do not start on the first row and first column, you can specify a `startrow` or `startcol` just like when you were using `read_blocks`. Or if your designs are located in a sheet other than the first sheet, you can specify `sheet`.

Additionally, if you've already pasted together your design elements yourself, then you should specify what string is being used as a separator via the `sep` argument (that gets passed to `separate_tidys`).

If you find yourself needing even more control over the process of importing block-shaped design files, each of the functions is available for users to call themselves. So you can run the steps manually, first reading with `read_blocks`, pasting as needed with `paste_blocks`, transforming to tidy with `trans_block_to_wide` and `trans_wide_to_tidy`, and finally separating design elements with `separate_tidys`.

Importing tidy-shaped design files

Just like measures data, to import tidy-shaped designs you could use the built-inRfunctions like `read.table`. However, if you need a few more options, you can use the `gcplyr` function `read_tidys`. Unlike the built-in option, `read_tidys` can import multiple tidy-shaped files at once, can add the filename as a column in the resulting data.frame, and can handle files where the tidy-shaped information doesn't start on the first row and column.

`read_tidys` only requires a filename (or vector of filenames, or relative file paths) and will return a `data.frame` (or list of data.frames) that you can save in R.

Once these design elements have been read into the `R` environment, you won't need to transform them. So you can skip down to learning how to merge them with your data in the **Merging spectrophotometric and design data** section.

Generating designs in R

If you'd rather make your design data.frames in R, `gcplyr` has a helper function that makes it easy to do so: `make_design`. `make_design` can create:

- block-shaped data.frames with your design information (e.g. for outputting to files)
- tidy-shaped data.frames with your design information (e.g. for merging with tidy-shaped plate reader data)

An example with a single design

Let's start with a simple example demonstrating the basic use of `make_design` (we'll move on to more complicated designs afterwards).

For example, let's imagine a growth curve experiment where a 96 well plate (12 columns and 8 rows) has a different bacterial strain in each row, but the first and last columns and first and last rows were left empty.

Row names	Column 1	Column 2	Column 3	…	Column 11	Column 12
Row A	Blank	Blank	Blank	…	Blank	Blank
Row B	Blank	Strain #1	Strain #1	…	Strain #1	Blank
Row B	Blank	Strain #2	Strain #2	…	Strain #2	Blank
…	…	…	…	…	…	…
Row G	Blank	Strain #5	Strain #5	…	Strain #5	Blank
Row G	Blank	Strain #6	Strain #6	…	Strain #6	Blank
Row H	Blank	Blank	Blank	…	Blank	Blank

Typing a design like this manually into a spreadsheet can be tedious. But generating a design `data.frame` using `make_design` is easier.

`make_design` first needs some general information, like the `nrows` and `ncols` in the plate, and the `output_format` you'd like (typically `blocks` or `tidy`).

Then, for each different design component, `make_design` needs five different pieces of information:

- a vector containing the possible values
- a vector specifying which rows these values should be applied to
- a vector specifying which columns these values should be applied to
- a string or vector of the pattern of these values
- a Boolean for whether this pattern should be filled byrow (defaults to TRUE)

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12,
  Bacteria = list(c("Str1", "Str2", "Str3",
                    "Str4", "Str5", "Str6"),
                  2:7,
                  2:11,
                  "123456",
                  FALSE)
)
```

So for our example above, we can see:

- the possible values are `c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6")`
- the rows these values should be applied to are rows `2:7`
- the columns these values should be applied to are columns `2:11`
- the pattern these values should be filled in by is `"123456"`
- and these values should *not* be filled by row, they should be filled by column

This entire list is passed with a name (here, "Bacteria"), that will be used as the resulting column header.

What does the result look like?

```
my_design_blk
#> [[1]]
#> [[1]]$data
#>   1  2      3      4      5      6      7      8      9      10     11     12
#> A NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> B NA "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" NA
#> C NA "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" NA
#> D NA "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" NA
#> E NA "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" NA
#> F NA "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" NA
#> G NA "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" NA
#> H NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#>
#> [[1]]$metadata
#> block_name
#> "Bacteria"
```

We can see that `make_design` has created a block-shaped `data.frame` containing the design elements as requested, and has attached a `metadata` containing the `block_name` (this is useful for later transformation to tidy-shaped, or if you're generating multiple design elements).

A few notes on the pattern

One of the most important elements of every argument passed to `make_design` is the string or vector specifying the pattern of values.

Oftentimes, it will be most convenient to simply use single-characters to correspond to the values. This is the default behavior of `make_design`, which splits the pattern string into individual characters, and then uses those characters to correspond to the indices of the values you provided.

For instance, in the example above, I used the numbers 1 through 6 to correspond to the values `"Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"`.

It's important to **note that the "0" character is reserved for `NA` values.** There is an example of this later.

If you have more than 9 values, you can use letters (uppercase and/or lowercase). In that case, you just have to specify a `lookup_tbl_start` so that the function knows what letter you're using as the `1` index. If no `lookup_tbl_start` is specified, the default is to count numbers first, then uppercase letters, then lowercase letters. For instance, in the previous example, I could have equivalently done:

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12, lookup_tbl_start = "A",
  Bacteria = list(
    c("Str1", "Str2", "Str3", "Str4", "Str5", "Str6"),
    2:7,
    2:11,
    "ABCDEF",
    FALSE)
)
```

Or I could have done:

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(
    c("Str1", "Str2", "Str3", "Str4", "Str5", "Str6"),
    2:7,
    2:11,
    "abcdef",
    FALSE)
)
```

Alternatively, you can use a separating character like a comma to delineate your indices. If you are doing so in order to use multicharacter indices (like numbers with more than one digit), all your indices will have to be numeric.

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12, pattern_split = ",",
  Bacteria = list(
    c("Str1", "Str2", "Str3", "Str4", "Str5", "Str6"),
    2:7,
    2:11,
    "1,2,3,4,5,6",
    FALSE)
)
```

If you find it easier to input the pattern as a vector rather than as a string that needs to be split, you can do that too. Just like when passing a string, if you're not using numbers, then uppercase letters, then lowercase letters for your indices, make sure to specify a different `lookup_tbl_start`:

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12,
  Bacteria = list(
    c("Str1", "Str2", "Str3", "Str4", "Str5", "Str6"),
    2:7,
    2:11,
    c(1,2,3,4,5,6),
    FALSE)
)
```

Continuing with the example: multiple designs

Now let's return to our example growth curve experiment. Imagine that now, *in addition* to having a different bacterial strain in each row, we also have a different media in each column in the plate.

Row names	Column 1	Column 2	Column 3	...	Column 11	Column 12
Row A	Blank	Blank	Blank	...	Blank	Blank
Row B	Blank	Media #1	Media #2	...	Media #10	Blank
...	...	...	...	...	...	...
Row G	Blank	Media #1	Media #2	...	Media #10	Blank
Row H	Blank	Blank	Blank	...	Blank	Blank

We can generate both the bacterial strain design and the media design simply by adding an additional argument to our make_design call.

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(c("Str1", "Str2", "Str3",
                    "Str4", "Str5", "Str6"),
                  2:7,
                  2:11,
                  "abcdef",
                  FALSE),
  Media = list(c("Med1", "Med2", "Med3",
                 "Med4", "Med5", "Med6",
                 "Med7", "Med8", "Med9",
                 "Med10", "Med11", "Med12"),
               2:7,
               2:11,
               "abcdefghij")
  )

my_design_blk
#> [[1]]
#> [[1]]$data
#>   1  2      3      4      5      6      7      8      9      10     11     12
#> A NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> B NA "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" NA
#> C NA "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" NA
```

```
#> D NA "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" NA
#> E NA "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" "Str4" NA
#> F NA "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" NA
#> G NA "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" NA
#> H NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#>
#> [[1]]$metadata
#> block_name
#> "Bacteria"
#>
#>
#> [[2]]
#> [[2]]$data
#>   1  2      3      4      5      6      7      8      9      10     11      12
#> A NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA      NA
#> B NA "Med1" "Med2" "Med3" "Med4" "Med5" "Med6" "Med7" "Med8" "Med9" "Med10" NA
#> C NA "Med1" "Med2" "Med3" "Med4" "Med5" "Med6" "Med7" "Med8" "Med9" "Med10" NA
#> D NA "Med1" "Med2" "Med3" "Med4" "Med5" "Med6" "Med7" "Med8" "Med9" "Med10" NA
#> E NA "Med1" "Med2" "Med3" "Med4" "Med5" "Med6" "Med7" "Med8" "Med9" "Med10" NA
#> F NA "Med1" "Med2" "Med3" "Med4" "Med5" "Med6" "Med7" "Med8" "Med9" "Med10" NA
#> G NA "Med1" "Med2" "Med3" "Med4" "Med5" "Med6" "Med7" "Med8" "Med9" "Med10" NA
#> H NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA      NA
#>
#> [[2]]$metadata
#> block_name
#>    "Media"
```

Here we can see that two blocks have been created, one with our bacterial strains, and one with our media.

Now, imagine after the experiment we discover that Bacterial Strain 4 and Media #6 were contaminated, and we'd like to exclude them from our analyses by marking them as `NA` in the design. We can simply modify our pattern string, placing a 0 anywhere we would like an `NA` to be filled in.

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(c("Str1", "Str2", "Str3",
                    "Str4", "Str5", "Str6"),
                  2:7,
                  2:11,
                  "abc0ef",
                  FALSE),
  Media = list(c("Med1", "Med2", "Med3",
                 "Med4", "Med5", "Med6",
                 "Med7", "Med8", "Med9",
                 "Med10", "Med11", "Med12"),
              2:7,
              2:11,
              "abcde0ghij")
  )

my_design_blk
#> [[1]]
#> [[1]]$data
```

```
#>   1  2      3      4      5      6      7      8      9      10     11     12
#> A NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> B NA "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" NA
#> C NA "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" NA
#> D NA "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" NA
#> E NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> F NA "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" NA
#> G NA "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" NA
#> H NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#>
#> [[1]]$metadata
#> block_name
#> "Bacteria"
#>
#>
#> [[2]]
#> [[2]]$data
#>   1  2      3      4      5      6      7  8      9      10     11      12
#> A NA NA     NA     NA     NA     NA     NA NA     NA     NA     NA      NA
#> B NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> C NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> D NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> E NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> F NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> G NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> H NA NA     NA     NA     NA     NA     NA NA     NA     NA     NA      NA
#>
#> [[2]]$metadata
#> block_name
#>    "Media"
```

Now we can see that our design has been easily modified to place NA's for those wells, which we can use after merging our designs with our data to exclude all of those wells from analyses.

However, the real strength of `make_design` is that it is not limited to simple alternating patterns. The pattern specified can be any pattern, which `make_design` will replicate sufficient times to cover the entire set of listed wells.

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(c("Str1", "Str2"),
                  2:7,
                  2:11,
                  "abaaabbbab",
                  FALSE),
  Media = list(c("Med1", "Med2", "Med3"),
               2:7,
               2:11,
               "aabbbc000abc"))

my_design_blk
#> [[1]]
#> [[1]]$data
```

```
#>   1  2      3      4      5      6      7      8      9      10     11     12
#> A NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> B NA "Str1" "Str2" "Str1" "Str1" "Str1" "Str1" "Str2" "Str1" "Str1" "Str1" NA
#> C NA "Str2" "Str2" "Str1" "Str2" "Str2" "Str2" "Str2" "Str1" "Str2" "Str2" NA
#> D NA "Str1" "Str1" "Str1" "Str1" "Str2" "Str1" "Str1" "Str1" "Str1" "Str2" NA
#> E NA "Str1" "Str2" "Str2" "Str2" "Str2" "Str1" "Str2" "Str2" "Str2" "Str2" NA
#> F NA "Str1" "Str1" "Str2" "Str1" "Str1" "Str1" "Str1" "Str2" "Str1" "Str1" NA
#> G NA "Str2" "Str2" "Str2" "Str1" "Str2" "Str2" "Str2" "Str2" "Str1" "Str2" NA
#> H NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#>
#> [[1]]$metadata
#> block_name
#> "Bacteria"
#>
#>
#> [[2]]
#> [[2]]$data
#>   1  2      3      4      5      6      7      8      9      10     11     12
#> A NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> B NA "Med1" "Med1" "Med2" "Med2" "Med2" "Med3" NA     NA     NA     "Med1" NA
#> C NA "Med2" "Med3" "Med1" "Med1" "Med2" "Med2" "Med2" "Med3" NA     NA     NA
#> D NA NA     "Med1" "Med2" "Med3" "Med1" "Med1" "Med2" "Med2" "Med2" "Med3" NA
#> E NA NA     NA     NA     "Med1" "Med2" "Med3" "Med1" "Med1" "Med2" "Med2" NA
#> F NA "Med2" "Med3" NA     NA     NA     "Med1" "Med2" "Med3" "Med1" "Med1" NA
#> G NA "Med2" "Med2" "Med2" "Med3" NA     NA     NA     "Med1" "Med2" "Med3" NA
#> H NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#>
#> [[2]]$metadata
#> block_name
#>    "Media"
```

gcplyr also includes an optional helper function for `make_design` called `make_designpattern`. `make_designpattern` just helps by reminding the user what arguments are necessary for each design and ensuring they're in the correct order. For example, the following produces the same `data.frame` as the above code:

```
my_design_blk <- make_design(
  output_format = "blocks",
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = make_designpattern(
    values = c("Str1", "Str2", "Str3",
               "Str4", "Str5", "Str6"),
    rows = 2:7, cols = 2:11, pattern = "abc0ef",
    byrow = FALSE),
  Media = make_designpattern(
    values = c("Med1", "Med2", "Med3",
               "Med4", "Med5", "Med6",
               "Med7", "Med8", "Med9",
               "Med10", "Med11", "Med12"),
    rows = 2:7, cols = 2:11, pattern = "abcde0ghij"))

my_design_blk
#> [[1]]
#> [[1]]$data
```

```
#>   1  2      3      4      5      6      7      8      9      10     11     12
#> A NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> B NA "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" "Str1" NA
#> C NA "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" "Str2" NA
#> D NA "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" "Str3" NA
#> E NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#> F NA "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" "Str5" NA
#> G NA "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" "Str6" NA
#> H NA NA     NA     NA     NA     NA     NA     NA     NA     NA     NA     NA
#>
#> [[1]]$metadata
#> block_name
#> "Bacteria"
#>
#>
#> [[2]]
#> [[2]]$data
#>   1  2      3      4      5      6      7  8      9      10     11      12
#> A NA NA     NA     NA     NA     NA     NA NA     NA     NA     NA      NA
#> B NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> C NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> D NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> E NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> F NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> G NA "Med1" "Med2" "Med3" "Med4" "Med5" NA "Med7" "Med8" "Med9" "Med10" NA
#> H NA NA     NA     NA     NA     NA     NA NA     NA     NA     NA      NA
#>
#> [[2]]$metadata
#> block_name
#>    "Media"
```

So far, we've been using the `blocks` option for `output_format`, because it's easy to see that our design matches what we'd intended with that format. However, **for merging our designs with plate reader data, we need it tidy-shaped**. Luckily, there's no need to transform it yourself, simply change the `output_format` argument option to `tidy`.

```
my_design_tdy <- make_design(
  output_format = "tidy",
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = make_designpattern(
    values = c("Str1", "Str2", "Str3",
               "Str4", "Str5", "Str6"),
    rows = 2:7, cols = 2:11, pattern = "abc0ef",
    byrow = FALSE),
  Media = make_designpattern(
    values = c("Med1", "Med2", "Med3",
               "Med4", "Med5", "Med6",
               "Med7", "Med8", "Med9",
               "Med10", "Med11", "Med12"),
    rows = 2:7, cols = 2:11, pattern = "abcde0ghij"))

head(my_design_tdy, 20)
#>    Well Bacteria Media
#> 1    A1       NA    NA
```

```
#> 2    A2       NA     NA
#> 3    A3       NA     NA
#> 4    A4       NA     NA
#> 5    A5       NA     NA
#> 6    A6       NA     NA
#> 7    A7       NA     NA
#> 8    A8       NA     NA
#> 9    A9       NA     NA
#> 10  A10       NA     NA
#> 11  A11       NA     NA
#> 12  A12       NA     NA
#> 13   B1       NA     NA
#> 14   B2     Str1   Med1
#> 15   B3     Str1   Med2
#> 16   B4     Str1   Med3
#> 17   B5     Str1   Med4
#> 18   B6     Str1   Med5
#> 19   B7     Str1     NA
#> 20   B8     Str1   Med7
```

Saving designs to files

Often after generating designs in `R` with `make_design`, you'll want to save those designs to files. This might be so that human-readable files documenting your designs are available without opening `R`. Or perhaps it's because you need to post the design files, for instance to Dryad as part of a manuscript submission.

If you'd like to save your designs to files, you can save them either tidy-shaped or block-shaped. Both formats can easily be read back into `R` by `gcplyr`.

Saving tidy-shaped designs These design files will be less human-readable, but easier to import and merge. Additionally, tidy-shaped files are often better for data repositories, like Dryad. To save tidy-shaped designs, simply use the built-in `write.csv` function.

```
#See the previous section where we created my_design_tdy
write.csv(x = my_design_tdy, file = "tidy_design.csv",
          row.names = FALSE)
```

Saving block-shaped designs These design files will be more human-readable but require slightly more computational steps to import and merge. For these, use the `gcplyr` function `write_blocks`. Typically, you'll use `write_blocks` to save files in one of two formats:

- `multiple` - each block will be saved to its own `.csv` file
- `single` - all the blocks will be saved to a single `.csv` file, with an empty row in between them

Saving block-shaped designs to multiple files The default setting for `write_blocks` is `output_format = 'multiple'`. This creates one `csv` file for each block, naming the files according to the `block_names` in the metadata for each block.

```
#See the previous section where we created my_design_blk
write_blocks(my_design_blk)
```

```r
#Let's see what the files look like
print_df(read.csv("Bacteria.csv", header = FALSE, colClasses = "character"))
#>   1    2    3    4    5    6    7    8    9   10   11  12
#> A
#> B   Str1 Str1 Str1 Str1 Str1 Str1 Str1 Str1 Str1 Str1
#> C   Str2 Str2 Str2 Str2 Str2 Str2 Str2 Str2 Str2 Str2
#> D   Str3 Str3 Str3 Str3 Str3 Str3 Str3 Str3 Str3 Str3
#> E
#> F   Str5 Str5 Str5 Str5 Str5 Str5 Str5 Str5 Str5 Str5
#> G   Str6 Str6 Str6 Str6 Str6 Str6 Str6 Str6 Str6 Str6
#> H

print_df(read.csv("Media.csv", header = FALSE, colClasses = "character"))
#>   1    2    3    4    5    6 7    8    9   10    11  12
#> A
#> B   Med1 Med2 Med3 Med4 Med5   Med7 Med8 Med9 Med10
#> C   Med1 Med2 Med3 Med4 Med5   Med7 Med8 Med9 Med10
#> D   Med1 Med2 Med3 Med4 Med5   Med7 Med8 Med9 Med10
#> E   Med1 Med2 Med3 Med4 Med5   Med7 Med8 Med9 Med10
#> F   Med1 Med2 Med3 Med4 Med5   Med7 Med8 Med9 Med10
#> G   Med1 Med2 Med3 Med4 Med5   Med7 Med8 Med9 Med10
#> H
```

Saving block-shaped designs to a single file The other setting for `write_blocks` is `output_format = 'single'`. This creates a single `csv` file that contains all the blocks, putting metadata like `block_names` in rows that precede each block.

Let's take a look what the `single` output format looks like:

```r
#See the previous section where we created my_design_blk
write_blocks(my_design_blk, file = "Design.csv", output_format = "single")

#Let's see what the file looks like
print_df(read.csv("Design.csv", header = FALSE, colClasses = "character"))
#> block_name Bacteria
#>                    1    2    3    4    5    6    7    8    9   10    11  12
#>          A
#>          B          Str1 Str1 Str1 Str1 Str1 Str1 Str1 Str1 Str1  Str1
#>          C          Str2 Str2 Str2 Str2 Str2 Str2 Str2 Str2 Str2  Str2
#>          D          Str3 Str3 Str3 Str3 Str3 Str3 Str3 Str3 Str3  Str3
#>          E
#>          F          Str5 Str5 Str5 Str5 Str5 Str5 Str5 Str5 Str5  Str5
#>          G          Str6 Str6 Str6 Str6 Str6 Str6 Str6 Str6 Str6  Str6
#>          H
#>
#> block_name    Media
#>                    1    2    3    4    5    6    7    8    9   10    11  12
#>          A
#>          B          Med1 Med2 Med3 Med4 Med5      Med7 Med8 Med9 Med10
#>          C          Med1 Med2 Med3 Med4 Med5      Med7 Med8 Med9 Med10
#>          D          Med1 Med2 Med3 Med4 Med5      Med7 Med8 Med9 Med10
#>          E          Med1 Med2 Med3 Med4 Med5      Med7 Med8 Med9 Med10
#>          F          Med1 Med2 Med3 Med4 Med5      Med7 Med8 Med9 Med10
```

```
#>           G           Med1 Med2 Med3 Med4 Med5      Med7 Med8 Med9 Med10
#>           H
```

Here we can see all our design information has been saved to a single file, and the metadata has been added in rows before each block.

Best practices for saving designs to files It's best to leave the `make_design` and `write_blocks` commands in your analysis script, so that every time your analysis is run your design files are kept up to date. Just note that if your `make_design` command has `output_format = blocks`, you'll need to make a version where `output_format = tidy` that you can `merge_dfs` with your plate reader data.

Merging spectrophotometric and design data

Once we have both our design and data in theRenvironment and tidy-shaped, we can merge them using `merge_dfs`.

For this, we'll use the data in the `example_widedata` dataset that is included with `gcplyr`, and which was the source for our previous examples with `import_blockmeasures` and `read_wides`.

In the `example_widedata` dataset, we have 48 different bacterial strains. The left side of the plate has all 48 strains in a single well each, and the right side of the plate also has all 48 strains in a single well each:

Row names	Column 1	...	Column 6	Column 7	...	Column 12
Row A	Strain #1	...	Strain #6	Strain #1	...	Strain #6
Row B	Strain #7	...	Strain #12	Strain #7	...	Strain #12
...	...	...	...	...	...	...
Row G	Strain #37	...	Strain #42	Strain #37	...	Strain #42
Row H	Strain #43	...	Strain #48	Strain #43	...	Strain #48

Then, on the right hand side of the plate a phage was also inoculated (while the left hand side remained bacteria-only):

Row names	Column 1	...	Column 6	Column 7	...	Column 12
Row A	No Phage	...	No Phage	Phage Added	...	Phage Added
Row B	No Phage	...	No Phage	Phage Added	...	Phage Added
...	...	...	...	...	...	...
Row G	No Phage	...	No Phage	Phage Added	...	Phage Added
Row H	No Phage	...	No Phage	Phage Added	...	Phage Added

Let's generate our design:

```
example_design <- make_design(
  pattern_split = ",", nrows = 8, ncols = 12,
  "Bacteria_strain" = make_designpattern(
    values = paste("Strain", 1:48),
    rows = 1:8, cols = 1:6,
    pattern = 1:48,
    byrow = TRUE),
  "Bacteria_strain" = make_designpattern(
```

```
    values = paste("Strain", 1:48),
    rows = 1:8, cols = 7:12,
    pattern = 1:48,
    byrow = TRUE),
  "Phage" = make_designpattern(
    values = c("No Phage"),
    rows = 1:8, cols = 1:6,
    pattern = "1"),
  "Phage" = make_designpattern(
    values = c("Phage Added"),
    rows = 1:8, cols = 7:12,
    pattern = "1"))
```

Here's what the resulting data.frame looks like:

```
head(example_design, 20)
#>    Well Bacteria_strain       Phage
#> 1    A1        Strain 1    No Phage
#> 2    A2        Strain 2    No Phage
#> 3    A3        Strain 3    No Phage
#> 4    A4        Strain 4    No Phage
#> 5    A5        Strain 5    No Phage
#> 6    A6        Strain 6    No Phage
#> 7    A7        Strain 1 Phage Added
#> 8    A8        Strain 2 Phage Added
#> 9    A9        Strain 3 Phage Added
#> 10  A10        Strain 4 Phage Added
#> 11  A11        Strain 5 Phage Added
#> 12  A12        Strain 6 Phage Added
#> 13   B1        Strain 7    No Phage
#> 14   B2        Strain 8    No Phage
#> 15   B3        Strain 9    No Phage
#> 16   B4       Strain 10    No Phage
#> 17   B5       Strain 11    No Phage
#> 18   B6       Strain 12    No Phage
#> 19   B7        Strain 7 Phage Added
#> 20   B8        Strain 8 Phage Added
```

Now let's transform the `example_widedata` to tidy-shaped.

```
example_tidydata <- trans_wide_to_tidy(example_widedata,
                                       id_cols = "Time")
```

And finally, we merge the two using `merge_dfs`, saving the result to `ex_dat_mrg`, short for example_data_merged:

```
ex_dat_mrg <-
  merge_dfs(example_tidydata,
            example_design)
#> Joining, by = "Well"

head(ex_dat_mrg)
```

```
#>   Time Well Measurements Bacteria_strain    Phage
#> 1    0   A1        0.003        Strain 1 No Phage
#> 2    0   B1        0.001        Strain 7 No Phage
#> 3    0   C1        0.002       Strain 13 No Phage
#> 4    0   D1        0.002       Strain 19 No Phage
#> 5    0   E1        0.002       Strain 25 No Phage
#> 6    0   F1        0.001       Strain 31 No Phage
```

Pre-processing

Now that we have our data and designs merged, we're almost ready to start processing and analyzing them. However, first we need to carry out any necessary pre-processing steps, like excluding wells that were contaminated or empty, and converting time formats to numeric.

Pre-processing: excluding data

In some cases, we want to remove some of the wells from our growth curves data before we carry on with downstream analyses. For instance, they may have been left empty, contained negative controls, or were contaminated. We can use `dplyr`'s `filter` function to remove those wells that meet criteria we want to exclude.

For instance, let's imagine that we realized that we put the wrong media into Well B1, and so we should remove it from our analyses. In that case, we can simply:

```
#We have previously loaded dplyr, but if you haven't already then
#make sure to add the line:
#   library(dplyr)

example_data_and_designs_filtered <- filter(ex_dat_mrg, Well != "B1")
head(example_data_and_designs_filtered)
#>   Time Well Measurements Bacteria_strain    Phage
#> 1    0   A1        0.003        Strain 1 No Phage
#> 2    0   C1        0.002       Strain 13 No Phage
#> 3    0   D1        0.002       Strain 19 No Phage
#> 4    0   E1        0.002       Strain 25 No Phage
#> 5    0   F1        0.001       Strain 31 No Phage
#> 6    0   G1        0.002       Strain 37 No Phage
```

Now we can see that all rows from Well B1 have been excluded. We could do something similar if we realized that a Bacterial strain was contaminated. For instance, if strain 13 was contaminated, we could exclude it (and Well B1) as follows:

```
example_data_and_designs_filtered <-
  filter(ex_dat_mrg,
         Well != "B1", Bacteria_strain != "Strain 13")
head(example_data_and_designs_filtered)
#>   Time Well Measurements Bacteria_strain    Phage
#> 1    0   A1        0.003        Strain 1 No Phage
#> 2    0   D1        0.002       Strain 19 No Phage
#> 3    0   E1        0.002       Strain 25 No Phage
#> 4    0   F1        0.001       Strain 31 No Phage
```

```
#> 5     0    G1          0.002        Strain 37 No Phage
#> 6     0    H1          0.002        Strain 43 No Phage
```

Pre-processing: converting dates & times into numeric

Growth curve data produced by a plate reader often encodes the timestamp information as a string (e.g. "2:45:11" for 2 hours, 45 minutes, and 11 seconds), while downstream analyses need timestamp information as a numeric (e.g. number of seconds elapsed). Luckily, others have written great packages that make it easy to convert from common date-time text formats into plain numeric formats. Here, we'll see how to use `lubridate` to do so:

First we have to create a data frame with time saved as it might be by a plate reader. As usual, **don't worry how this block of code works**, since it's just creating an example file in the same format as that output by a plate reader.

```
ex_dat_mrg$Time <-
  paste(ex_dat_mrg$Time %/% 3600,
        formatC((ex_dat_mrg$Time %% 3600) %/% 60,
                width = 2, flag = 0),
        formatC((ex_dat_mrg$Time %% 3600) %% 60,
                width = 2, flag = 0),
        sep = ":")
```

Let's take a look at this data.frame. This shows the `Time` column as it might be written by a plate reader.

```
head(ex_dat_mrg)
#>      Time Well Measurements Bacteria_strain    Phage
#> 1 0:00:00   A1        0.003        Strain 1 No Phage
#> 2 0:00:00   B1        0.001        Strain 7 No Phage
#> 3 0:00:00   C1        0.002       Strain 13 No Phage
#> 4 0:00:00   D1        0.002       Strain 19 No Phage
#> 5 0:00:00   E1        0.002       Strain 25 No Phage
#> 6 0:00:00   F1        0.001       Strain 31 No Phage
```

We can see that our `Time` aren't written in an easy numeric. Instead, they're in a format that's easy for a human to understand (but unfortunately not very usable for analysis).

Let's use `lubridate` to convert this text into a usable format. `lubridate` has a whole family of functions that can parse text with hour, minute, and/or second components. You can use `hms` if your text contains hour, minute, and second information, `hm` if it only contains hour and minute information, and `ms` if it only contains minute and second information.

Since the example has all three, we'll use `hms`. Once `hms` has parsed the text, we'll use another function to convert the output of `hms` into a pure numeric value: `time_length`. By default, `time_length` returns in units of seconds, but you can change that by changing the `unit` argument to `time_length`. See `?time_length` for details.

```
#We have previously loaded lubridate, but if you haven't already then
#make sure to add the line:
#   library(lubridate)

ex_dat_mrg$Time <- time_length(hms(ex_dat_mrg$Time))
```

```
head(ex_dat_mrg)
#>   Time Well Measurements Bacteria_strain    Phage
#> 1    0   A1        0.003        Strain 1 No Phage
#> 2    0   B1        0.001        Strain 7 No Phage
#> 3    0   C1        0.002       Strain 13 No Phage
#> 4    0   D1        0.002       Strain 19 No Phage
#> 5    0   E1        0.002       Strain 25 No Phage
#> 6    0   F1        0.001       Strain 31 No Phage
```

And now we can see that we've gotten nice numeric `Time` values! So we can proceed with the next steps of the analysis.

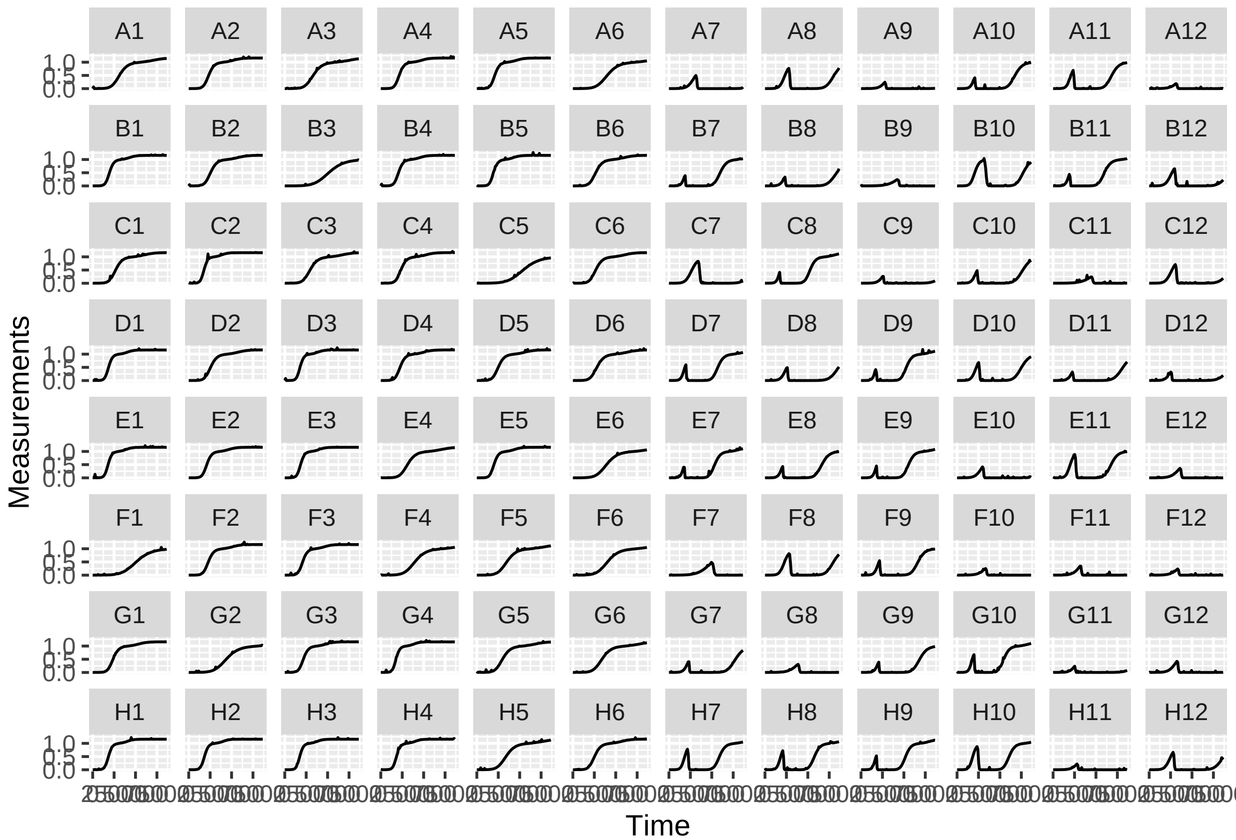
Plotting your data

Once your data has been merged and times have been converted to numeric, we can easily plot our data using the `ggplot2` package. That's because `ggplot2` was specifically built on the assumption that data would be tidy-shaped, which ours is! We won't go into depth on how to use `ggplot` here, but there are three main commands to the plot below:

- `ggplot` - the ggplot function is where you specify the `data.frame` you would like to use and the *aes*thetics of the plot (the x and y axes you would like)
- `geom_line` - tells `ggplot` how we would like to plot the data, in this case with a line (another common `geom` for time-series data is `geom_point`)
- `facet_wrap` - tells `ggplot` to plot each Well in a separate facet

We'll be using this format to plot our data throughout the remainder of this vignette

```
#We have previously loaded ggplot2, but if you haven't already then
#make sure to add the line:
#    library(ggplot2)

#First, we'll reorder the Well levels so they plot in the correct order
ex_dat_mrg$Well <-
  factor(ex_dat_mrg$Well,
         levels = paste(rep(LETTERS[1:8], each = 12), 1:12, sep = ""))

ggplot(data = ex_dat_mrg, aes(x = Time, y = Measurements)) +
  geom_line() +
  facet_wrap(~Well, nrow = 8, ncol = 12)
```



Generally speaking, **from here on you should plot your data frequently**, and in every way you can think of! **After every processing and analysis step, visualize both the input data and output data** to understand what the processing and analysis steps are doing and whether they are the right choices for your particular data (this vignette will be doing that too!)

How to process and analyze your data

With your data and design information pre-processed, **your dataset is now organized in a way that's easy to export and analyze**. It is also at this point that the next steps for what you can do diversify into many options.

Broadly speaking, there are two main approaches to analyzing growth curves data:

1. directly quantify attributes of the growth dynamics
2. fit the growth dynamics with a mathematical model, then extract parameters from the fitted model

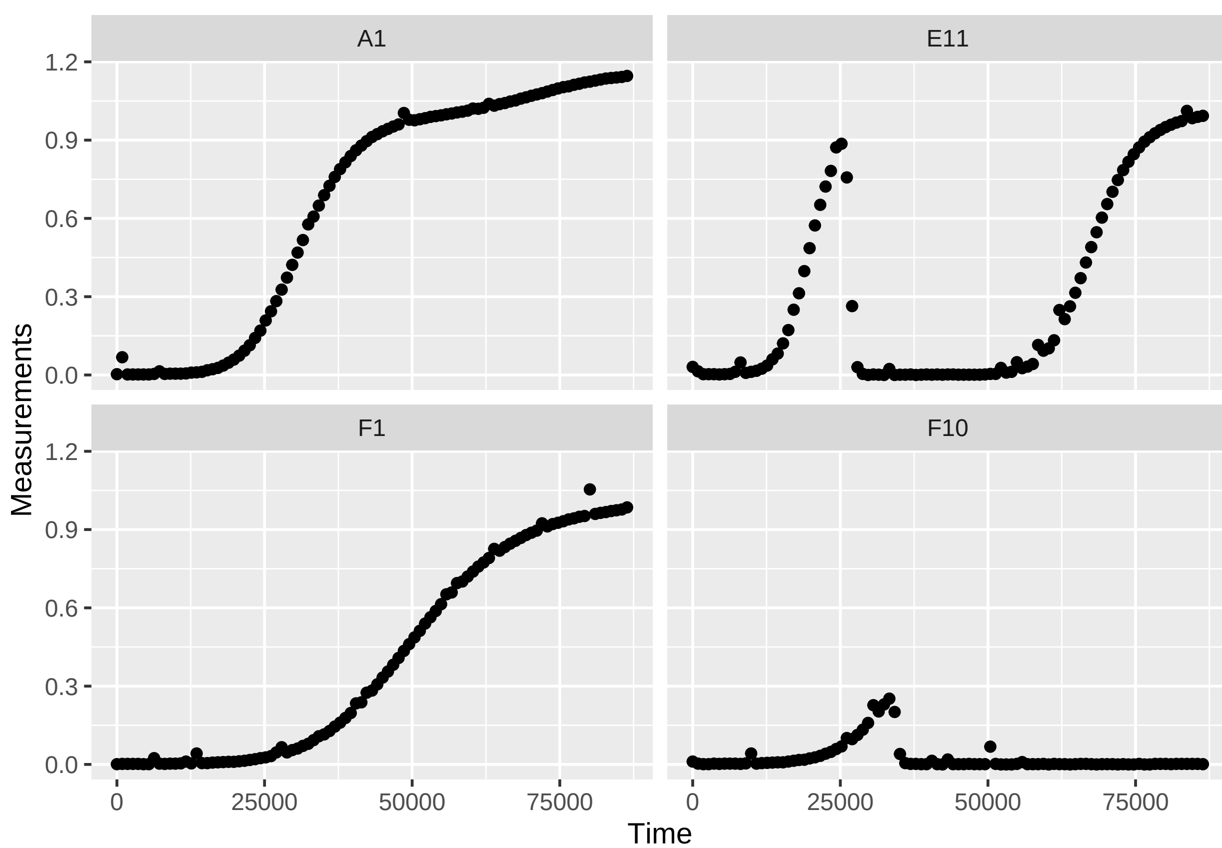
The remaining functions of `gcplyr` can facilitate analyses following the first approach: directly quantifying attributes of the observed dynamics. If you're interested in exploring model-fitting approaches, which can provide enormous analytical power, check out the **Other growth curve analysis packages** section. At this point, since the data is now well-organized, advanced users may also decide they want to write their own custom analyses (in lieu of, or alongside, `gcplyr`-based and/or fitting-based analyses).

So, how do we directly quantify attributes of growth curves? First, we can process our data by smoothing (if needed) and calculating derivatives. Then, we can analyze it. `gcplyr` has a number of functions that facilitate these steps. However, unlike the import, transformation, and merging steps we've done so far, different projects may require different analyses, and not all users will have the same analysis steps. The **Processing** and **Analyzing** sections of this document, therefore, are written to highlight the functions available and provide examples of common analyses that you may want to run, rather than prescribing a set of analysis steps that everyone must do.

#Processing data: smoothing Oftentimes, growth curve data produced by a plate reader will be noisy, and some degree of smoothing before analysis is necessary for analyses to succeed. `gcplyr` has a `smooth_data` function that can carry out such smoothing. **Generally, you should carry out as little smoothing as is necessary for your analyses to work**.

Let's take a look at a few wells from our example data, which have some noise.

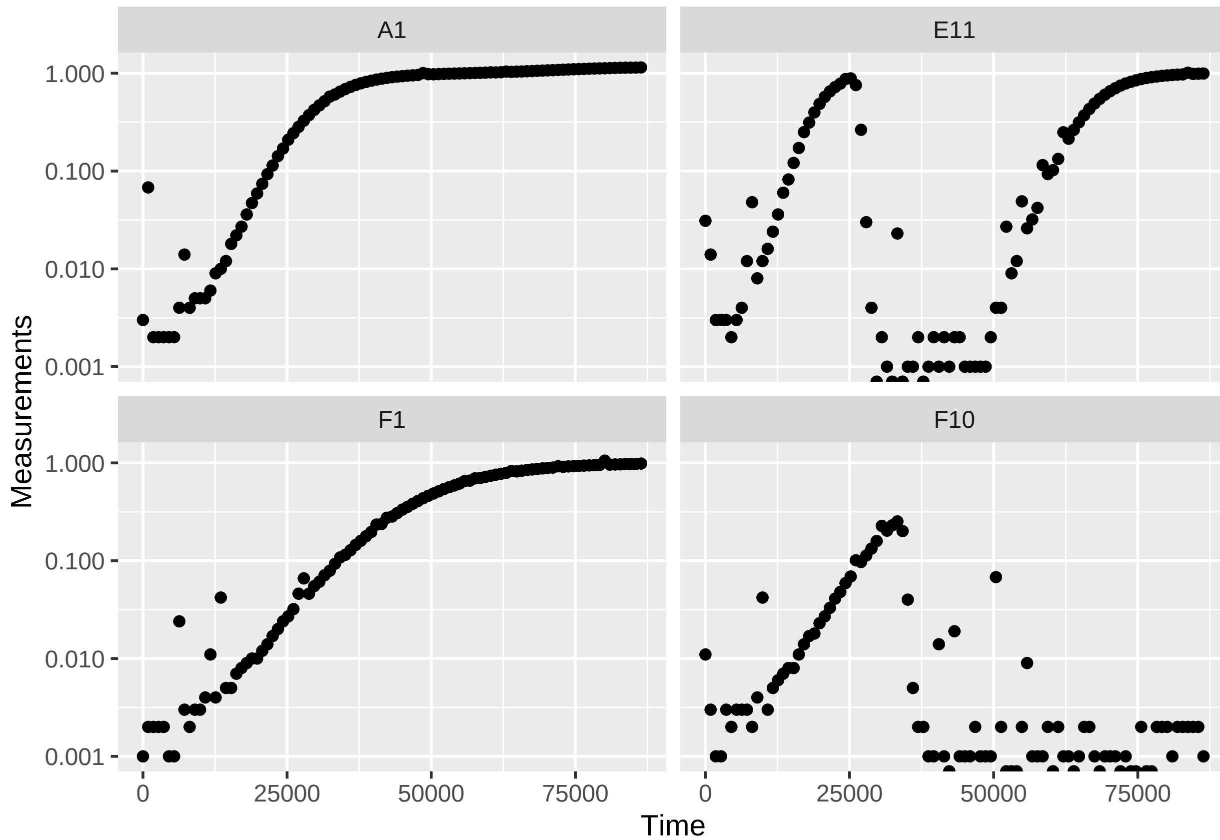
```
#Here we've chosen four wells that, from our previous plot, seem
# representative of the overall diversity of dynamics
#In your own code, you should visualize all your data
sample_wells <- c("A1", "F1", "F10", "E11")
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = Measurements)) +
  geom_point() +
  facet_wrap(~Well)
```



Well that doesn't look so bad! Let's take a quick peek at the data using a log scale for the y-axis. Log scales are particularly useful for plotting growth curves because exponential growth is a straight line when plotted on a log scale.

```
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = Measurements)) +
```

```
  geom_point() +
  facet_wrap(~Well) +
  scale_y_continuous(trans = "log10")
#> Warning: Transformation introduced infinite values in continuous y-axis
```



Oh. That looks a lot more noisy! In fact, this is a common occurrence: **at low densities, random noise tends to have a much larger effect** than at high densities. Unfortunately, steps like calculating maximum growth rate can be very sensitive to such noise, so we're going to need to smooth our data to prevent that from being a problem.

So, how can we smooth our data? Well, `smooth_data` has four different smoothing algorithms to choose from: `moving-average`, `moving-median`, `loess`, and `gam`.

- `moving-average` is a simple smoothing algorithm that primarily acts to reduce the effects of outliers on the data
- `moving-median` is another simple smoothing algorithm that primarily acts to reduce the effects of outliers on the data
- `loess` is a spline-fitting approach that uses polynomial-like curves, which produces curves with smoothly changing derivatives, but can in some cases create curvature artifacts not present in the original data
- `gam` is also a spline-fitting approach that is often recommended over `loess`, although you should evaluate its efficacy for your own data

Additionally, all four smoothing algorithms have a tuning parameter that controls how "smoothed" the data are. For whichever smoothing method you're using, **you should plot smoothing with multiple different tuning parameter values**, then choose the value that smooths the data as little as is necessary to reduce noise. Make sure to plot the smoothing for every well in your data, so that you're choosing the best setting for all your data and not just one well.

Smoothing data is a step that alters the values you will analyze. Because of that, and because there are so many options for how to smooth your data, it is a step that can be rife with pitfalls. I recommend starting with the simplest and least "smoothed" smoothing, plotting your results, and only increasing your smoothing as much as is needed to enable downstream analyses. Additionally, when sharing your findings, it's important to be transparent by sharing the raw data and smoothing methods, rather than treating the smoothed data as your source.

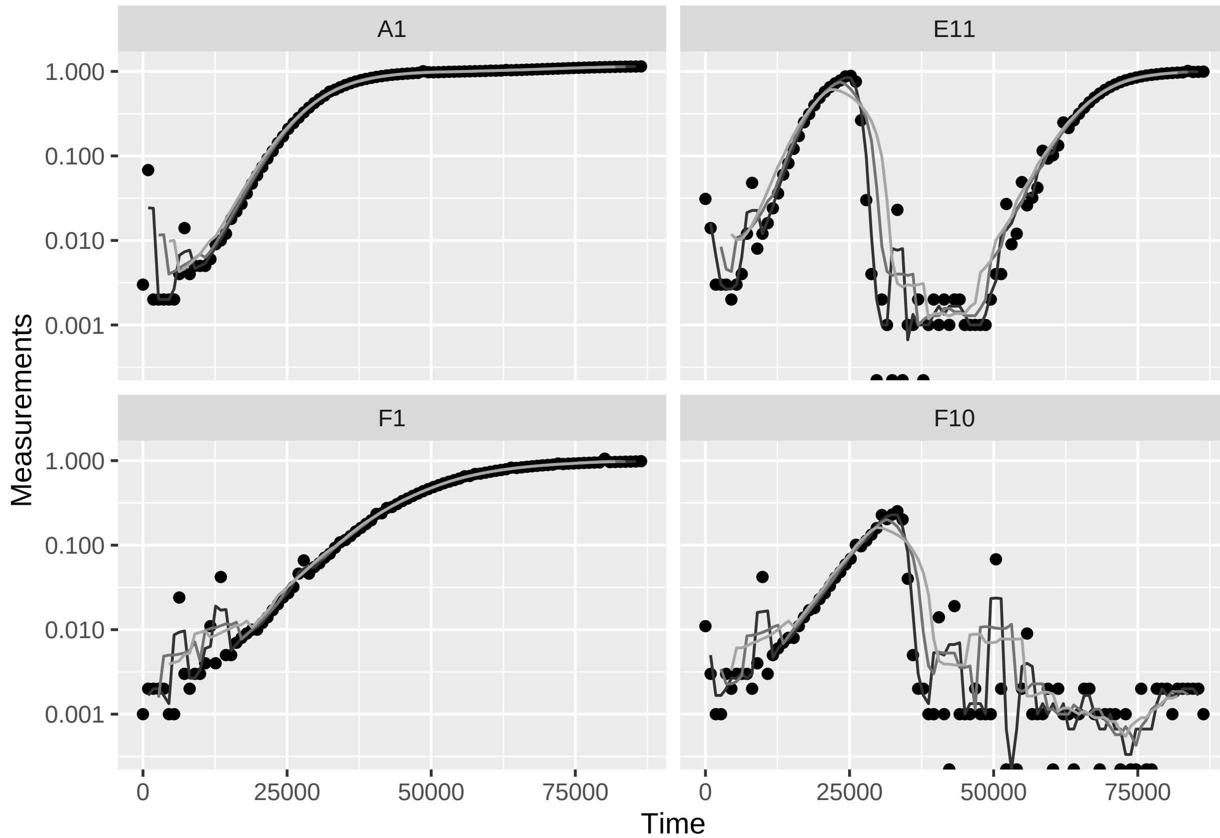
To use `smooth_data`, pass your x and y values, your method of choice, and any additional arguments needed for the method. It will return a vector of your smoothed y values.

Since your dataframe likely includes data from multiple wells (or even plates), we'll want to only smooth within each of those subsets. You can specify the groupings using the `subset_by` argument, which should be a vector as long as y whose unique values denote the subset groups. (Note: if you're using `smooth_data` within `dplyr::mutate` on a grouped `data.frame`, there's no need for the `subset_by` argument)

Smoothing with moving-average

For `moving-average`, the tuning parameter is `window_width_n`, which specifies how many data points wide the moving window used to calculate the average is. Specifying the `window_width_n` is required, and larger values will be more "smoothed". Here, we'll show moving averages with windows that are 3, 7, and 11 data points wide (because the window is centered on each data point, it must be an odd number of data points wide).

```
ex_dat_mrg$smoothed3 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "moving-average", subset_by = ex_dat_mrg$Well,
              window_width_n = 3)
ex_dat_mrg$smoothed7 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "moving-average", subset_by = ex_dat_mrg$Well,
              window_width_n = 7)
ex_dat_mrg$smoothed11 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "moving-average", subset_by = ex_dat_mrg$Well,
              window_width_n = 11)

#What does the smoothed data look like compared to the noisy original?
#Lighter lines are wider window_width_n's and more "smoothed"
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time)) +
  geom_point(aes(y = Measurements)) +
  geom_line(aes(y = smoothed3), color = "gray20") +
  geom_line(aes(y = smoothed7), color = "gray45") +
  geom_line(aes(y = smoothed11), color = "gray65") +
  facet_wrap(~Well) +
  scale_y_continuous(trans = "log10")
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Transformation introduced infinite values in continuous y-axis
#> Warning: Removed 2 row(s) containing missing values (geom_path).
#> Warning: Removed 6 row(s) containing missing values (geom_path).
#> Warning: Removed 10 row(s) containing missing values (geom_path).
```



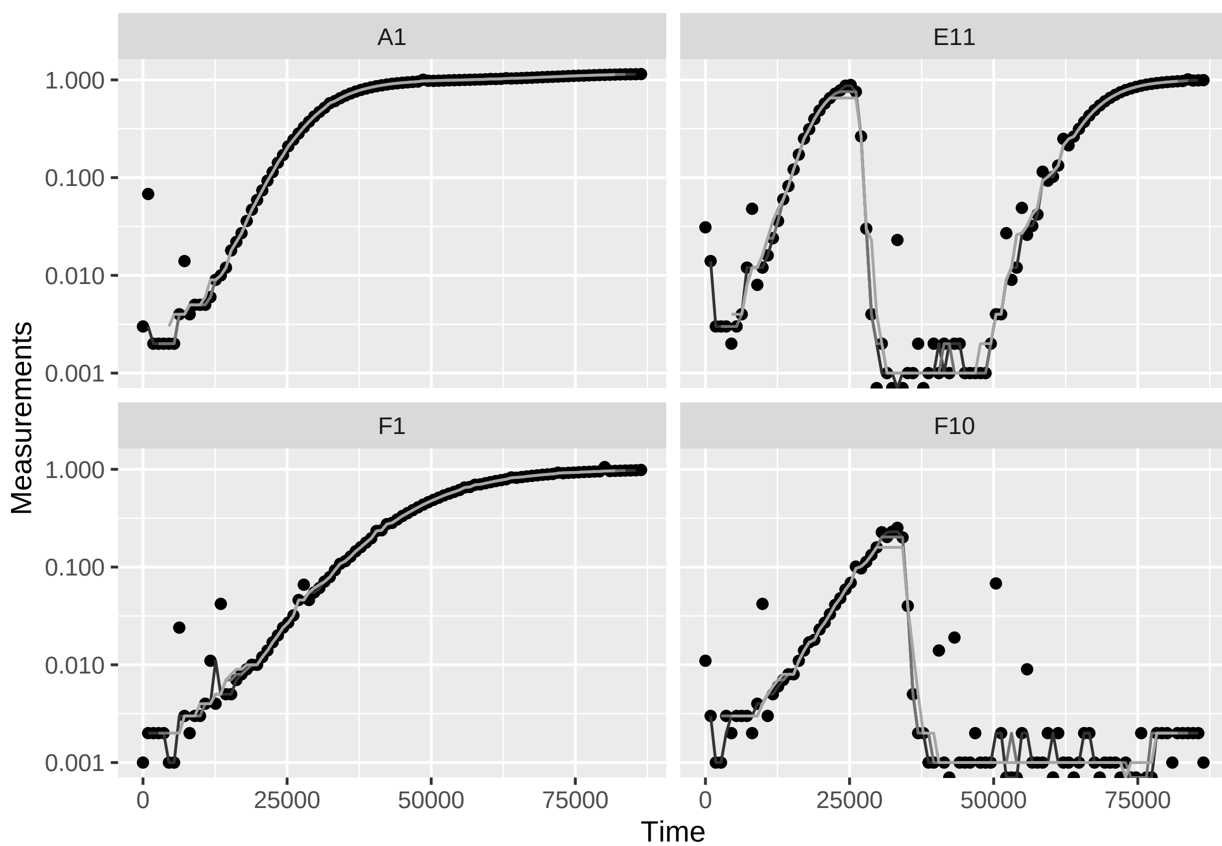
Here we can see that `moving-average` has helped reduce the effects of some of that early noise. However, with `window_width_n = 11` (the lightest line), the smoothing has started biasing our medium-density data points to be higher than they actually are. Based on this, we'd probably want to use a `window_width_n` less than 11. Unfortunately, with smaller `window_width_n` our early data is still being affected by that early noise, so we should explore other smoothing methods, or try combining multiple smoothing methods.

Smoothing with moving-median

For `moving-median`, the tuning parameter is also `window_width_n`, which specifies how many data points wide the moving window used to calculate the average is. Specifying the `window_width_n` is required, and larger values will be more "smoothed". Here, we'll show moving averages with windows that are 3, 7, and 11 data points wide (because the window is centered on each data point, it must be an odd number of data points wide).

```
ex_dat_mrg$smoothed3 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "moving-median", subset_by = ex_dat_mrg$Well,
              window_width_n = 3)
ex_dat_mrg$smoothed7 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "moving-median", subset_by = ex_dat_mrg$Well,
              window_width_n = 7)
ex_dat_mrg$smoothed11 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "moving-median", subset_by = ex_dat_mrg$Well,
```

```
              window_width_n = 11)

#What does the smoothed data look like compared to the noisy original?
#Lighter lines are wider window_width_n's and more "smoothed"
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time)) +
  geom_point(aes(y = Measurements)) +
  geom_line(aes(y = smoothed3), color = "gray20") +
  geom_line(aes(y = smoothed7), color = "gray45") +
  geom_line(aes(y = smoothed11), color = "gray65") +
  facet_wrap(~Well) +
  scale_y_continuous(trans = "log10")
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Transformation introduced infinite values in continuous y-axis
#> Transformation introduced infinite values in continuous y-axis
#> Transformation introduced infinite values in continuous y-axis
#> Warning: Removed 2 row(s) containing missing values (geom_path).
#> Warning: Removed 6 row(s) containing missing values (geom_path).
#> Warning: Removed 10 row(s) containing missing values (geom_path).
```

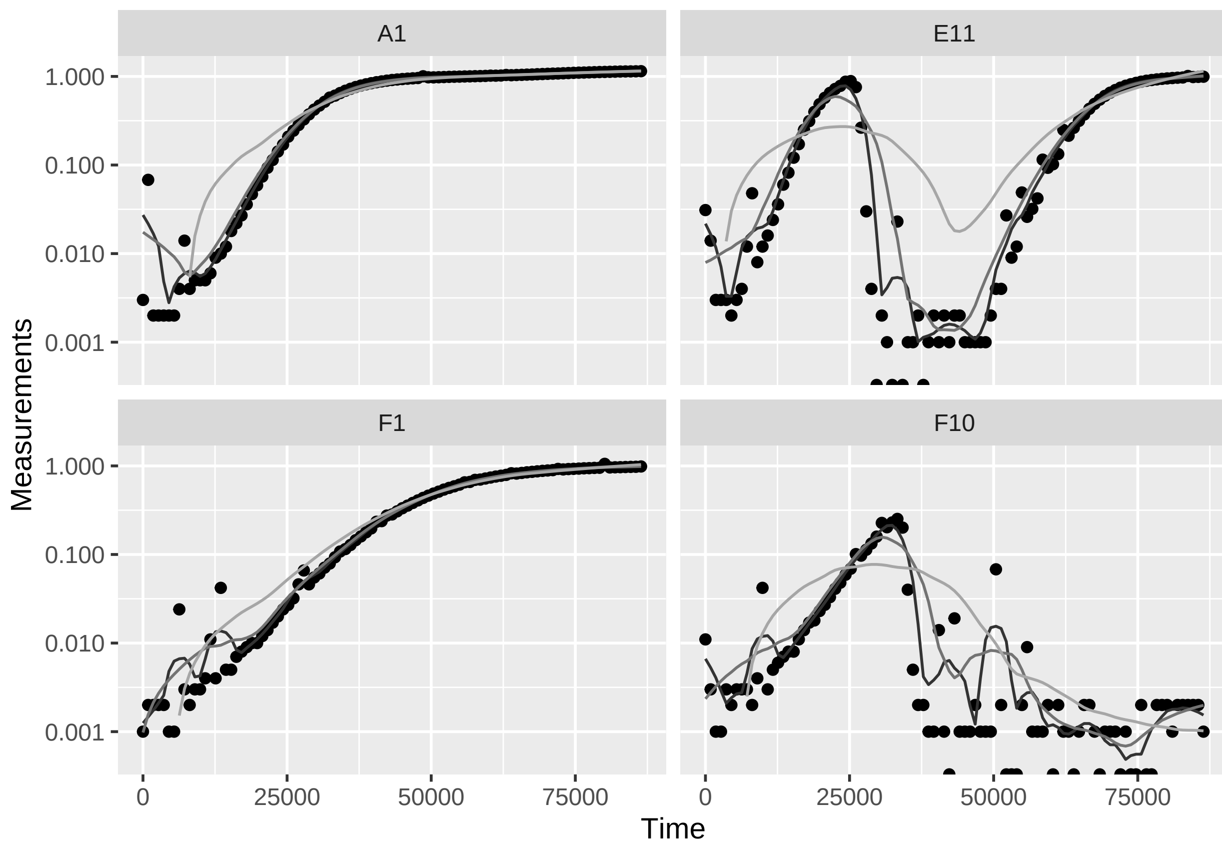


Here we can see that `moving-median` has really excluded that low-density noise, even with the smallest `window_width_n = 3`. Additionally, `moving-median` did not bias our larger data hardly at all, except with the widest `window_width_n`. However, it has produced a smoothed density that is fairly "jumpy", something that wider `window_width_n` did not fix. **This is common with moving-median**, so often you may need to try other smoothing methods or combining `moving-median` with other methods.

Smoothing with LOESS

For `loess`, the tuning parameter is the `span` argument. `loess` works by doing fits on subset windows of the data centered at each data point. These fits can be linear (`degree = 1`) or polynomial (typically `degree = 2`). `span` is the width of the window, as a fraction of all data points. For instance, with the default `span` of 0.75, 75% of the data points are included in each window. Thus, span values typically are between 0 and 1 (although see `?loess` for use of `span` values greater than 1), and larger values are more "smoothed". Here, we'll show `loess` smoothing with spans of 0.1, 0.2, and 0.5 and `degree = 1`.

```
ex_dat_mrg$smoothed1 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "loess", subset_by = ex_dat_mrg$Well,
              span = .1, degree = 1)
ex_dat_mrg$smoothed2 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "loess", subset_by = ex_dat_mrg$Well,
              span = .2, degree = 1)
ex_dat_mrg$smoothed5 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "loess", subset_by = ex_dat_mrg$Well,
              span = .5, degree = 1)

#What does the smoothed data look like compared to the noisy original?
#Lighter lines are larger span's and more "smoothed"
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time)) +
  geom_point(aes(y = Measurements)) +
  geom_line(aes(y = smoothed1), color = "gray20") +
  geom_line(aes(y = smoothed2), color = "gray45") +
  geom_line(aes(y = smoothed5), color = "gray65") +
  facet_wrap(~Well) +
  scale_y_continuous(trans = "log10")
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Warning in self$trans$transform(x): NaNs produced
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Warning: Removed 9 row(s) containing missing values (geom_path).
```



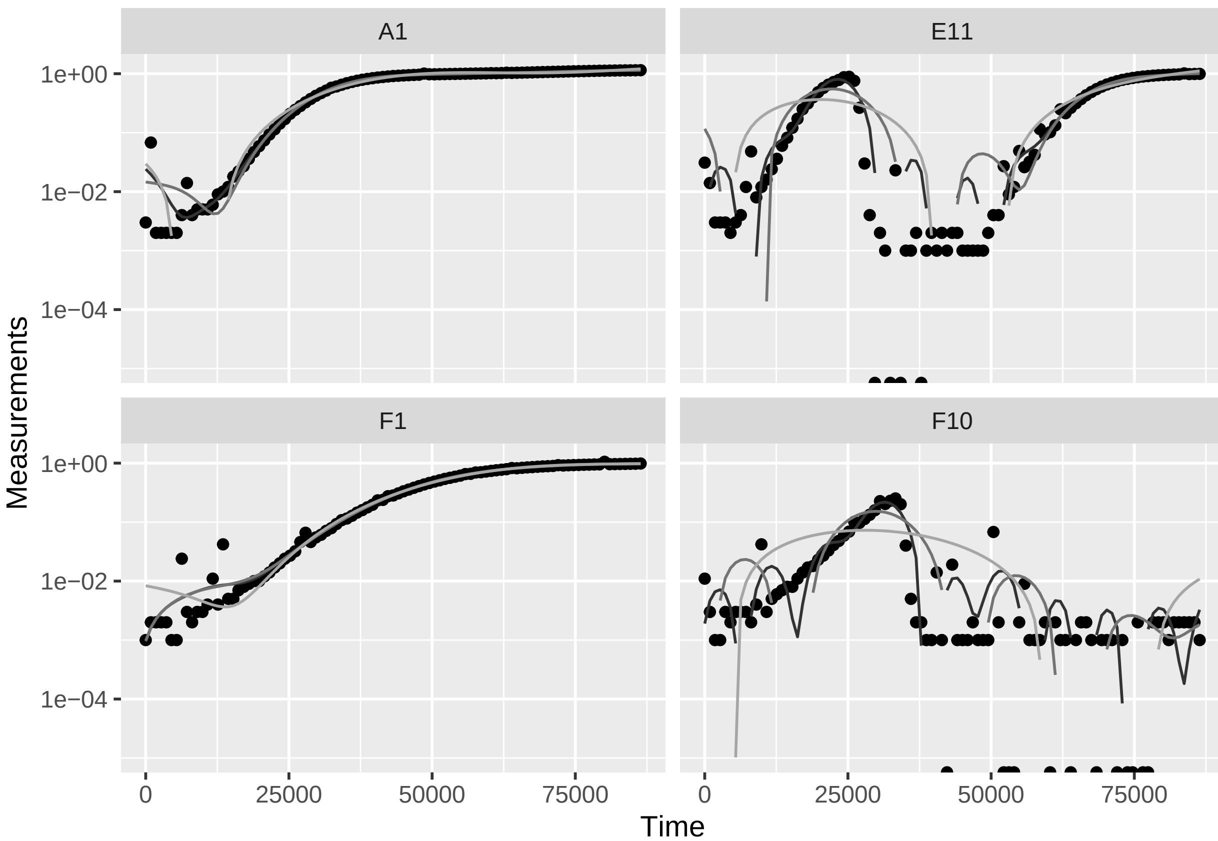
Here we can see that `loess` with smaller spans (darker lines) have smoothed the data somewhat but are still sensitive to outliers. However, `loess` with a larger span (lightest line) has introduced significant bias. To fix this, we might explore other smoothing methods, or combining `loess` with other smoothing methods.

Smoothing with GAM

For `gam`, the primary tuning parameter is the `k` argument. `gam` works by doing fits on subsets of the data and linking these fits together. `k` determines how many link points ("knots") it can use. If not specified, the default `k` value for smoothing a time series is 10, with **smaller values being more "smoothed"** (note this is opposite the trend with other smoothing methods). However, **unlike earlier methods, `k` values that are too large are also problematic**, as they will tend to 'overfit' the data. `k` cannot be larger than the number of data points, and should usually be substantially smaller than that. Here, we'll show `gam` smoothing with `k` values of 5, 10, and 20.

```
ex_dat_mrg$smoothed20 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "gam", subset_by = ex_dat_mrg$Well, k = 20)
ex_dat_mrg$smoothed10 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "gam", subset_by = ex_dat_mrg$Well, k = 10)
ex_dat_mrg$smoothed5 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "gam", subset_by = ex_dat_mrg$Well, k = 5)

#What does the smoothed data look like compared to the noisy original?
```

```r
#Lighter lines are smaller k and more "smoothed"
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time)) +
  geom_point(aes(y = Measurements)) +
  geom_line(aes(y = smoothed20), color = "gray20") +
  geom_line(aes(y = smoothed10), color = "gray45") +
  geom_line(aes(y = smoothed5), color = "gray65") +
  facet_wrap(~Well) +
  scale_y_continuous(trans = "log10")
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Warning in self$trans$transform(x): NaNs produced
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Warning in self$trans$transform(x): NaNs produced
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Warning in self$trans$transform(x): NaNs produced
#> Warning: Transformation introduced infinite values in continuous y-axis
```

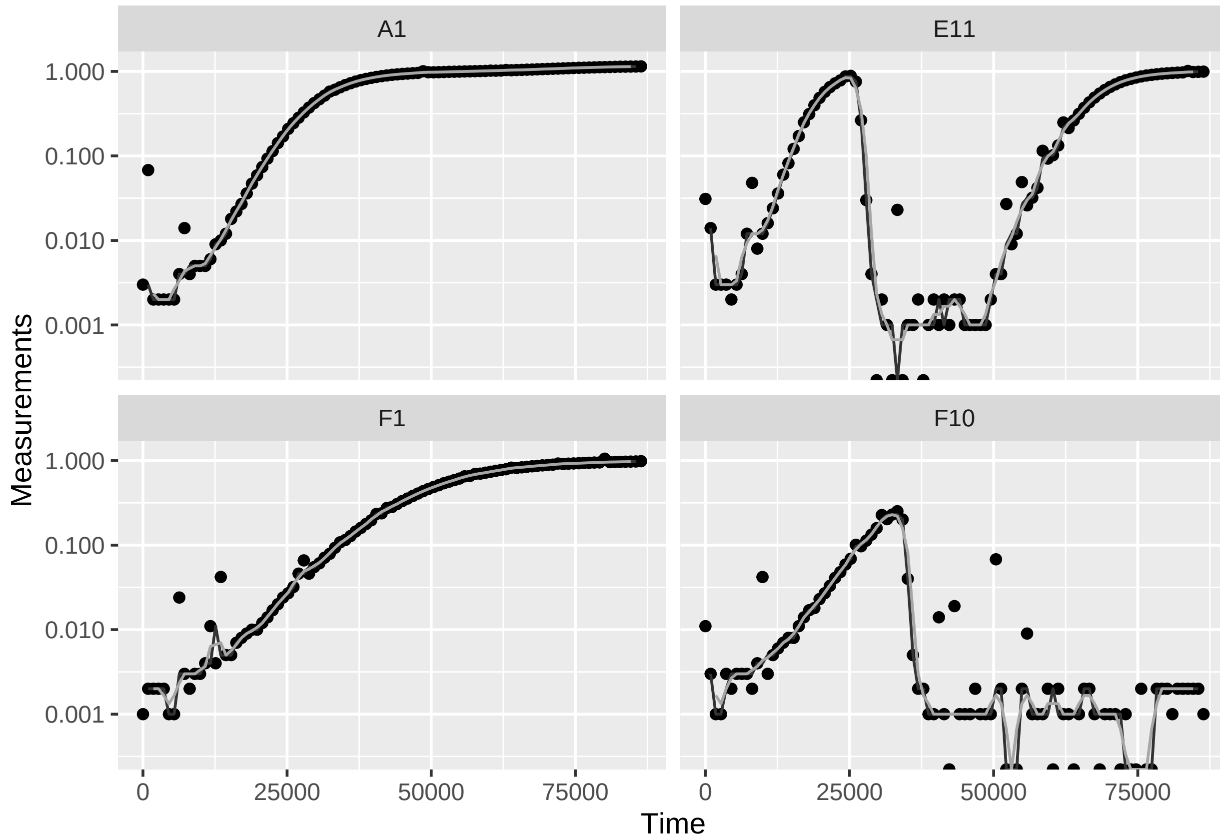


Here we can see that `gam` does alright when working with the no phage-added wells (A1 and F1): higher `k` values (darkest line) have smoothed the data but are still sensitive to those early outliers, while lower `k` values (lighter lines) have introduced significant bias. However, `gam` is struggling when phage have been added (E11 and F10). Across all the `k` values it has added many fluctuations and often dips into values of 0 or lower (plotted here as breaks in the line, since the log of numbers <= 0 are undefined). To fix this, we might explore other smoothing methods or combining `gam` with other smoothing methods.

Combining multiple smoothing methods

Often, combining multiple smoothing methods can provide improved results. For instance, `moving-median` is particularly good at removing outliers, but not very good at producing continuously smooth data. In contrast, `moving-average`, `loess`, and `gam` work better at producing continuously smooth data, but aren't as good at removing outliers. Here's an example using the strengths of both `moving-median` and `moving-average`:

```
ex_dat_mrg$smoothed_med3 <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$Measurements,
              sm_method = "moving-median", subset_by = ex_dat_mrg$Well,
              window_width_n = 3)
#Note that for the second round, we're using the first smoothing as input
ex_dat_mrg$smoothed <-
  smooth_data(x = ex_dat_mrg$Time, y = ex_dat_mrg$smoothed_med3,
              sm_method = "moving-average", subset_by = ex_dat_mrg$Well,
              window_width_n = 3)

#What does the smoothed data look like compared to the noisy original?
#The first round of smoothing with moving-median is plotted in lighter colors
#The second round of smoothing with moving-average is plotted in darker colors
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time)) +
  geom_point(aes(y = Measurements)) +
  geom_line(aes(y = smoothed_med3), color = "gray20") +
  geom_line(aes(y = smoothed), color = "gray65") +
  facet_wrap(~Well) +
  scale_y_continuous(trans = "log10")
#> Warning: Transformation introduced infinite values in continuous y-axis
#> Transformation introduced infinite values in continuous y-axis
#> Transformation introduced infinite values in continuous y-axis
#> Warning: Removed 2 row(s) containing missing values (geom_path).
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



Here we can see that the combination of minimal `moving-median` and `moving-average` smoothing has produced a curve that has most of the noise removed with minimal introduction of bias.

Processing data: calculating derivatives

In many cases, identifying features of a growth curve requires looking not only at the absorbance data over time, but the slope of the absorbance data over time. `gcplyr` includes a `calc_deriv` function that can be used to calculate the empirical derivative (slope) of absorbance data over time.

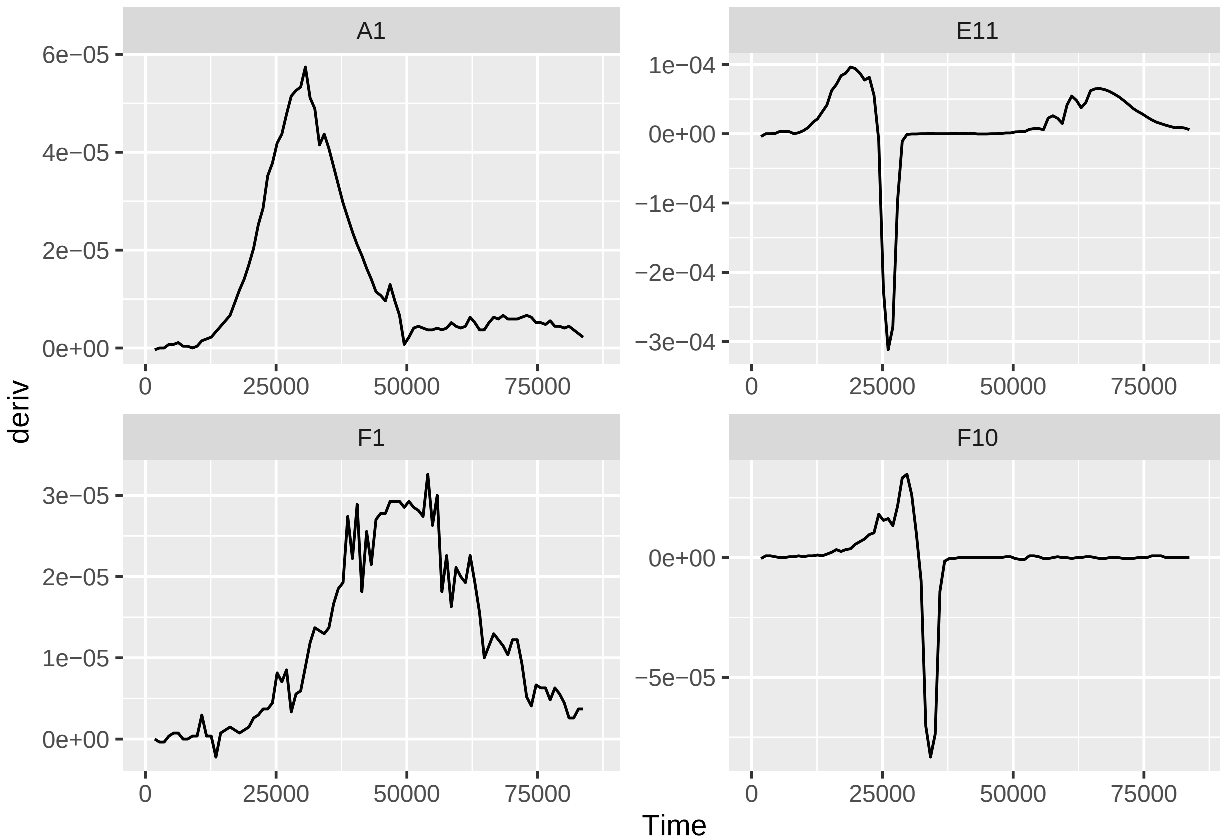
If you've previously smoothed your absorbance data, remember to use those smoothed values rather than the original values!

A simple derivative

To calculate a simple derivative (the slope of our original data) using `calc_deriv`, we simply have to provide the x and y values, along with a vector of `subset_by` values differentiating our unique growth curves (here, the different wells). (Note: if you're using `calc_deriv` within `dplyr::mutate`, there's no need to use the `subset_by` argument) Note that this is **not** the growth rate of the cells, but rather is a measure of how quickly the whole population was growing at each time point. This is useful for identifying events like population declines, or multiple rounds of growth.

```
ex_dat_mrg$deriv <-
  calc_deriv(x = ex_dat_mrg$Time, y = ex_dat_mrg$smoothed,
             subset_by = ex_dat_mrg$Well)
```

```
#Now let's plot the derivative
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv)) +
  geom_line() +
  facet_wrap(~Well, scales = "free")
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```



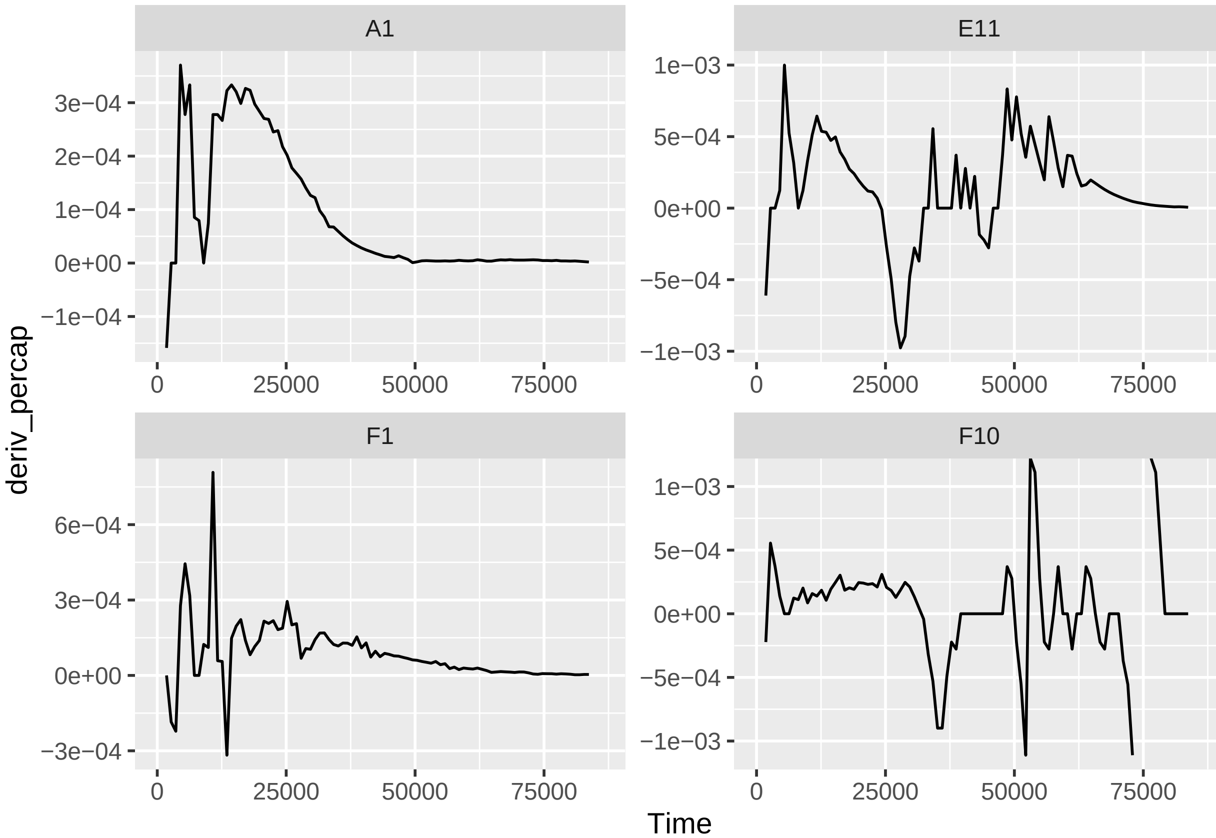
Here we can clearly see when the slope of the total population was increasing the fastest, and when it declines in the phage-added wells. But we can also see something surprising in Well A1 that may not have been immediately apparent visually: there is a second, slower, burst of growth later on. Such a pattern is common in bacterial growth curves and is called *diauxic growth*. Additionally, we can see in Well E11 when the bacteria start to grow again following near-extinction by phages, presumably after evolving resistance to the phage.

Per-capita derivative

If we want to calculate the growth rate of the cells, we need to use `calc_deriv` to return the **per-capita** derivative. Just as before, provide the x and y values, along with a vector of `subset_by` values (as needed), but now set `percapita = TRUE`. Note that in this case, you are required to specify a blank value, i.e. the value of your `Measurements` that corresponds to a population density of 0. If your data have already been normalized, simply add `blank = 0`.

```
ex_dat_mrg$deriv_percap <-
  calc_deriv(x = ex_dat_mrg$Time, y = ex_dat_mrg$smoothed,
```

```
             subset_by = ex_dat_mrg$Well, percapita = TRUE, blank = 0)

#Now let's plot the per-capita derivative
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv_percap)) +
  geom_line() +
  facet_wrap(~Well, scales = "free")
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```



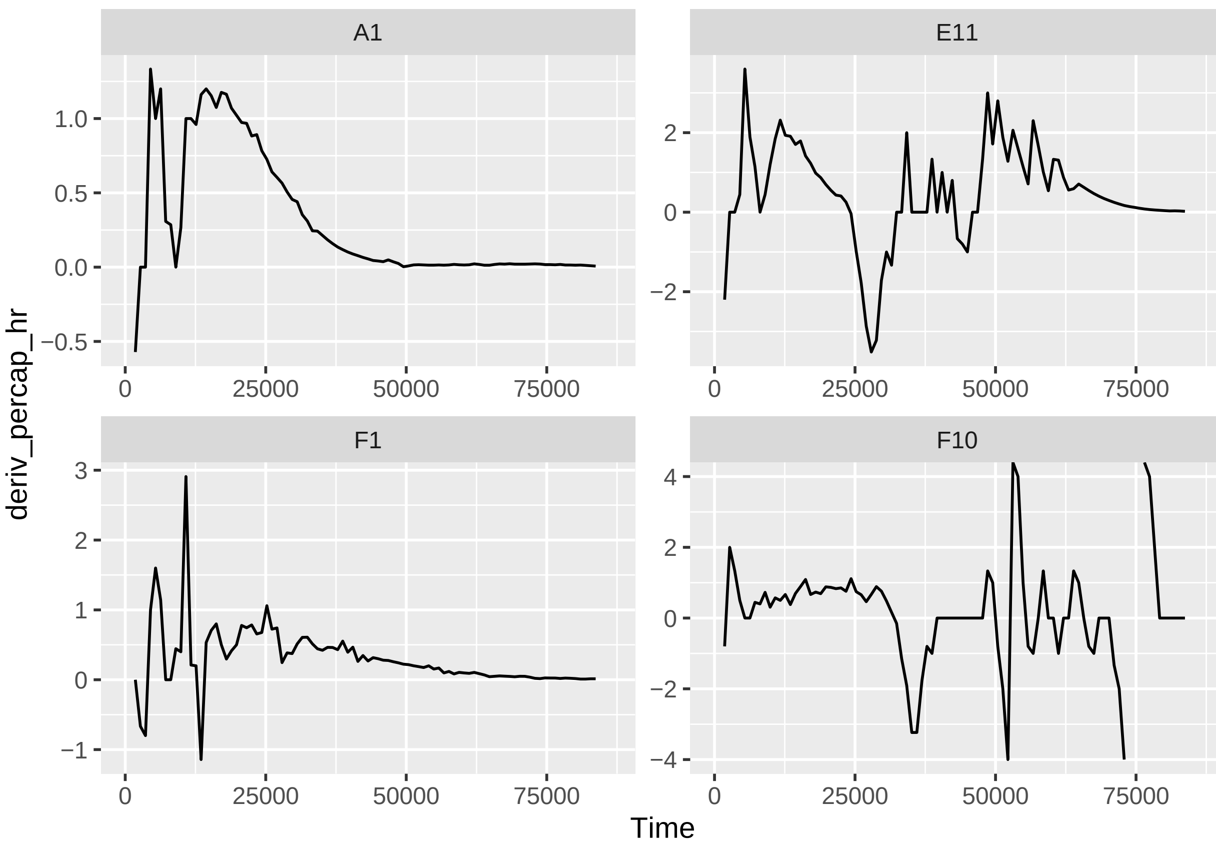
Here we can see that, in Well A1, the per-capita growth rate peaked much earlier in the time-series than might appear from the density dynamics or non per-capita derivative. We can also see that there was clearly a lag phase at the beginning before the bacteria started growing rapidly.

However, the other wells seem to have a lot of noise obscuring their per-capita growth rates. What happened? Why hasn't our smoothing been sufficient? As I explore later, per-capita growth rates can be strongly affected by even small noise at very low densities, something that can be excluded simply by only analyzing per-capita growth when densities are above some minimum value.

Changing the derivative units

To convert your x-axis (time) units in your derivative calculations to a different unit, use the `x_scale` argument. Simply specify the ratio of your x units to the desired units. For instance, in our example data `x` is the number of *seconds* since the growth curve began. What if we wanted growth rate in *per-hour*? There are 3600 seconds in an hour, so we set `x_scale = 3600`

```
ex_dat_mrg$deriv_percap_hr <-
  calc_deriv(x = ex_dat_mrg$Time, y = ex_dat_mrg$smoothed,
             subset_by = ex_dat_mrg$Well, percapita = TRUE, blank = 0,
             x_scale = 3600)

#Now let's plot the derivative in units of Abs/hour
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv_percap_hr)) +
  geom_line() +
  facet_wrap(~Well, scales = "free")
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```



Now we can see the bacterial growth rate in more-understandable units: peak growth rates are often around 1-2 times/hour (when ignoring the points that seem likely to be noise).

Analyzing data with summarize

Ultimately, analyzing growth curves requires summarizing the entire time series of data by some metric or metrics. For instance, we may calculate metrics like:

- the maximum density
- the total area under the curve
- the lag time (approximated as the time from the start until maximum per-capita growth rate is achieved)

- the maximum per-capita growth rate
- the density when a diauxic shift occurs
- the time until diauxic shift occurs
- the peak per-capita growth rate after a diauxic shift
- the peak density before a decline from phage predation
- the time when bacteria drop below some density because of phage predation

`gcplyr` contains a number of functions that make it easier to carry out these calculations. Additionally, `gcplyr` functions are flexible enough that you can use them in designing your own metric calculations. The following sections highlight general-use `gcplyr` functions and provide examples to calculate the common metrics above.

But first, we need to familiarize ourselves with the `dplyr` package and its functions `group_by` and `summarize`. Why? Because the upcoming `gcplyr` functions need to be used *within* `dplyr::summarize`. **If you're already familiar with `dplyr`, feel free to skip the primer in the next section.** If you're not familiar yet, don't worry! Continue to the next section, where I provide a primer that will teach you all you need to know on using `group_by` and `summarize` for `gcplyr`.

A brief primer on dplyr

The `R` package `dplyr` provides a "grammar of data manipulation" that is useful for a broad array of data analysis tasks (in fact, `dplyr` is the direct inspiration for the name of this package!) For our purposes, we're going to focus on two particular functions: `group_by` and `summarize` (also available as `summarise`).

The `group_by` functions in `dplyr` allow users to group the rows of their `data.frame`'s into groups. Then, `summarize` will carry out user-specified calculations on *each* group independently, producing a new `data.frame` where each group is a single row.

For growth curves, this means we will:

1. `group_by` our data so that every well is a group
2. `summarize` each well with calculations like maximum density or area under the curve

Since `dplyr` will drop columns that the data aren't grouped by and that aren't summarized, we will typically want to list all of our design columns for `group_by`, along with the plate name and well. Make sure you're *not* grouping by Time, Absorbance, or anything else that varies *within* a well, since if you do `dplyr` will group timepoints within a well separately.

In the next section, I provide a simple example of how the `max` function is used with `group_by` and `summarize` to calculate lag time and the maximum per-capita growth rate. If you want to learn more, `dplyr` has extensive documentation and examples of its own online. Feel free to explore them as desired, but this primer and the coming example should be sufficient to use the remaining `gcplyr` functions, which (as a reminder) have to be used *within* `summarize` to work correctly.

Summarizing with simple base functions: maximum and minimum density

One of the most common steps is calculating global maxima and minima of data. For instance, with bacterial growth, maximum density is one of the most commonly measured traits. Here, we'll show how to find it using the built-in `max` function.

First, we need to group our data. `group_by` simply requires the `data.frame` to be grouped, and the names of the columns we want to group by.

```
#First, drop unneeded columns (optional)
ex_dat_mrg <- dplyr::select(ex_dat_mrg,
                            Time, Well, Measurements, Bacteria_strain, Phage,
                            smoothed, deriv, deriv_percap, deriv_percap_hr)
#Then, carry out grouping
grouped_ex_dat_mrg <- group_by(ex_dat_mrg, Bacteria_strain, Phage, Well)

head(grouped_ex_dat_mrg)
#> # A tibble: 6 x 9
#> # Groups:   Bacteria_strain, Phage, Well [6]
#>    Time Well  Measurements Bacteria_strain Phage    smoothed deriv deriv_pe~1 deriv~2
#>   <dbl> <fct>        <dbl> <chr>           <chr>       <dbl> <dbl>      <dbl>   <dbl>
#> 1     0 A1           0.003 Strain 1        No Phage       NA    NA         NA      NA
#> 2     0 B1           0.001 Strain 7        No Phage       NA    NA         NA      NA
#> 3     0 C1           0.002 Strain 13       No Phage       NA    NA         NA      NA
#> 4     0 D1           0.002 Strain 19       No Phage       NA    NA         NA      NA
#> 5     0 E1           0.002 Strain 25       No Phage       NA    NA         NA      NA
#> 6     0 F1           0.001 Strain 31       No Phage       NA    NA         NA      NA
#> # ... with abbreviated variable names 1: deriv_percap, 2: deriv_percap_hr
```

Notice that this hasn't changed anything about our `data.frame`, but `R` now knows what the groups are. Then, we run `summarize`, specifying:

1. the name of the variable we want results saved to
2. the function that calculates the summarized results

For instance, in the code below we've calculated the maximum of the `smoothed` column, and saved it in a column named `max_dens` (note that we need to specify `na.rm = TRUE` to tell `max` to ignore all `NA` values). We've saved the output from `summarize` to a new `data.frame`: `ex_dat_mrg_sum`, short for example_data_merged_summarized.

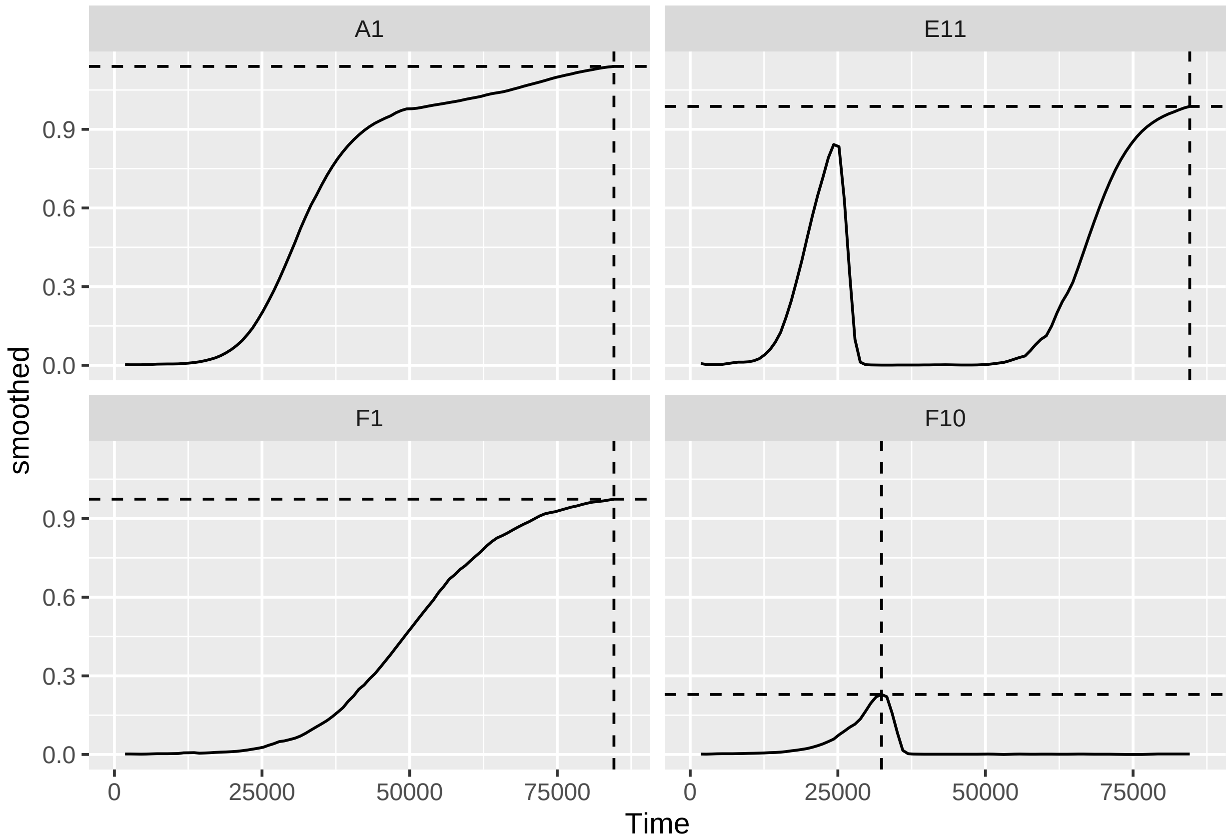
```
ex_dat_mrg_sum <- summarize(grouped_ex_dat_mrg,
                            max_dens = max(smoothed, na.rm = TRUE))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(ex_dat_mrg_sum)
#> # A tibble: 6 x 4
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  max_dens
#>   <chr>           <chr>       <fct>    <dbl>
#> 1 Strain 1        No Phage    A1       1.14
#> 2 Strain 1        Phage Added A7       0.453
#> 3 Strain 10       No Phage    B4       1.16
#> 4 Strain 10       Phage Added B10      0.959
#> 5 Strain 11       No Phage    B5       1.17
#> 6 Strain 11       Phage Added B11      1.02
```

If you want additional characteristics, you simply add them to the summarize. For instance, if we want the time when the maximum density occurs, you just add that as a second argument. In this case, we use the `which.max` function, which returns the index of the maximum value, to get the index of the `Time` when the maximum occurs, and save it to a column titled `max_time`:

```
ex_dat_mrg_sum <- summarize(grouped_ex_dat_mrg,
                            max_dens = max(smoothed, na.rm = TRUE),
                            max_time = Time[which.max(smoothed)])
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(ex_dat_mrg_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  max_dens max_time
#>   <chr>           <chr>       <fct>    <dbl>    <dbl>
#> 1 Strain 1        No Phage    A1       1.14     84600
#> 2 Strain 1        Phage Added A7       0.453    30600
#> 3 Strain 10       No Phage    B4       1.16     78300
#> 4 Strain 10       Phage Added B10      0.959    30600
#> 5 Strain 11       No Phage    B5       1.17     65700
#> 6 Strain 11       Phage Added B11      1.02     84600
```

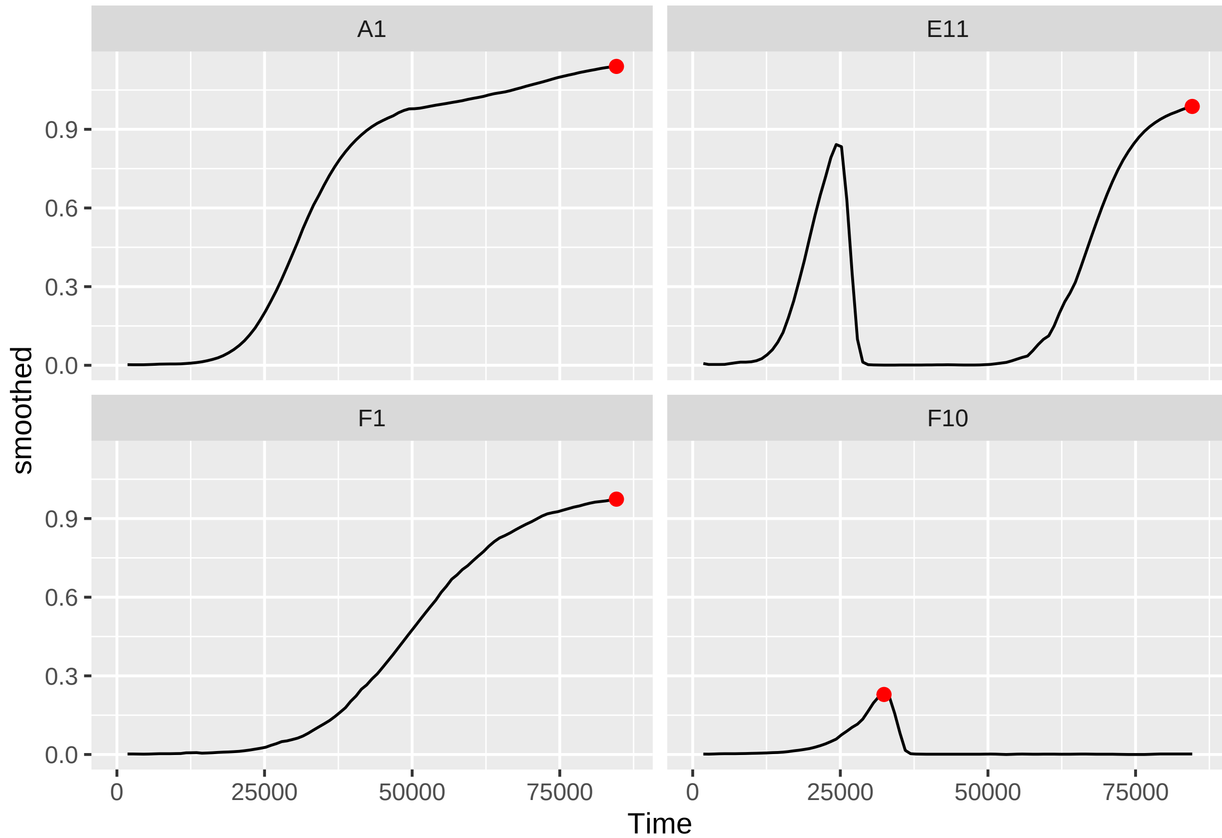
And we can quite easily plot such summarized values as a horizontal line or vertical line on top of our original growth curves data with the `geom_hline` or `geom_vline` functions:

```
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well) +
  geom_hline(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(yintercept = max_dens), lty = 2) +
  geom_vline(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(xintercept = max_time), lty = 2)
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



Alternatively, we could plot these summary points as a point:

```
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well) +
  geom_point(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(x = max_time, y = max_dens),
             size = 2, color = "red")
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



Summarizing with simple gcplyr functions: area under the curve

One common metric of growth curves is the total area under the curve. `gcplyr` has an `auc` function to easily calculate this area. Just like `min` and `max`, it needs to be used inside `summarize` on a `data.frame` that has been grouped.

To use `auc`, simply specify the x and y data whose area-under-the-curve you want to calculate. Here, we calculate the area-under-the-curve of the `smoothed` column and save it to a column titled `auc`.

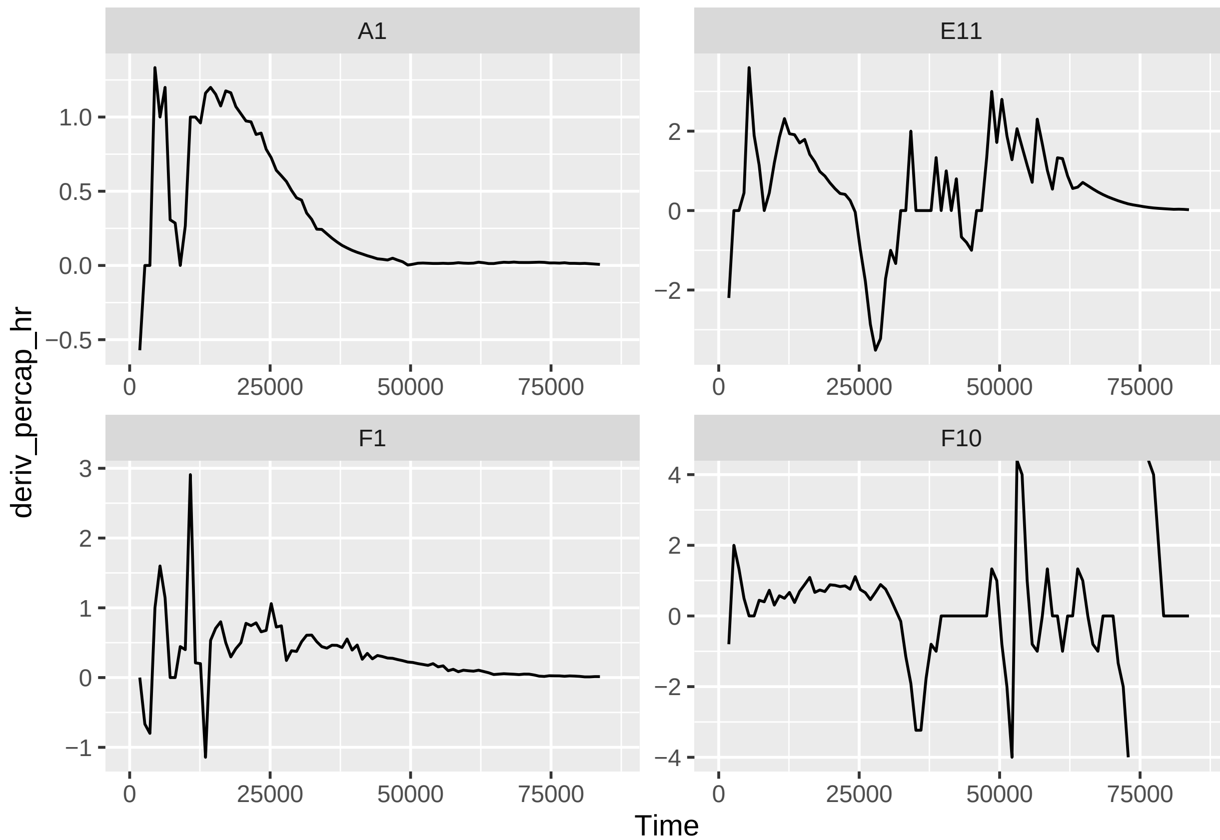
```
ex_dat_mrg_sum <-
  summarize(grouped_ex_dat_mrg,
            auc = auc(x = Time, y = smoothed))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(ex_dat_mrg_sum)
#> # A tibble: 6 x 4
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well     auc
#>   <chr>           <chr>       <fct>  <dbl>
#> 1 Strain 1        No Phage    A1    54952.
#> 2 Strain 1        Phage Added A7     3846
#> 3 Strain 10       No Phage    B4    69766.
#> 4 Strain 10       Phage Added B10   20743.
#> 5 Strain 11       No Phage    B5    71456.
#> 6 Strain 11       Phage Added B11   26149.
```

Summarizing on subsets: maximum growth rate

Sometimes, we need to provide limits on the data passed to our simple functions. We can demonstrate this in the process of calculating one of the most common metrics we want to identify: the maximum per-capita growth rate

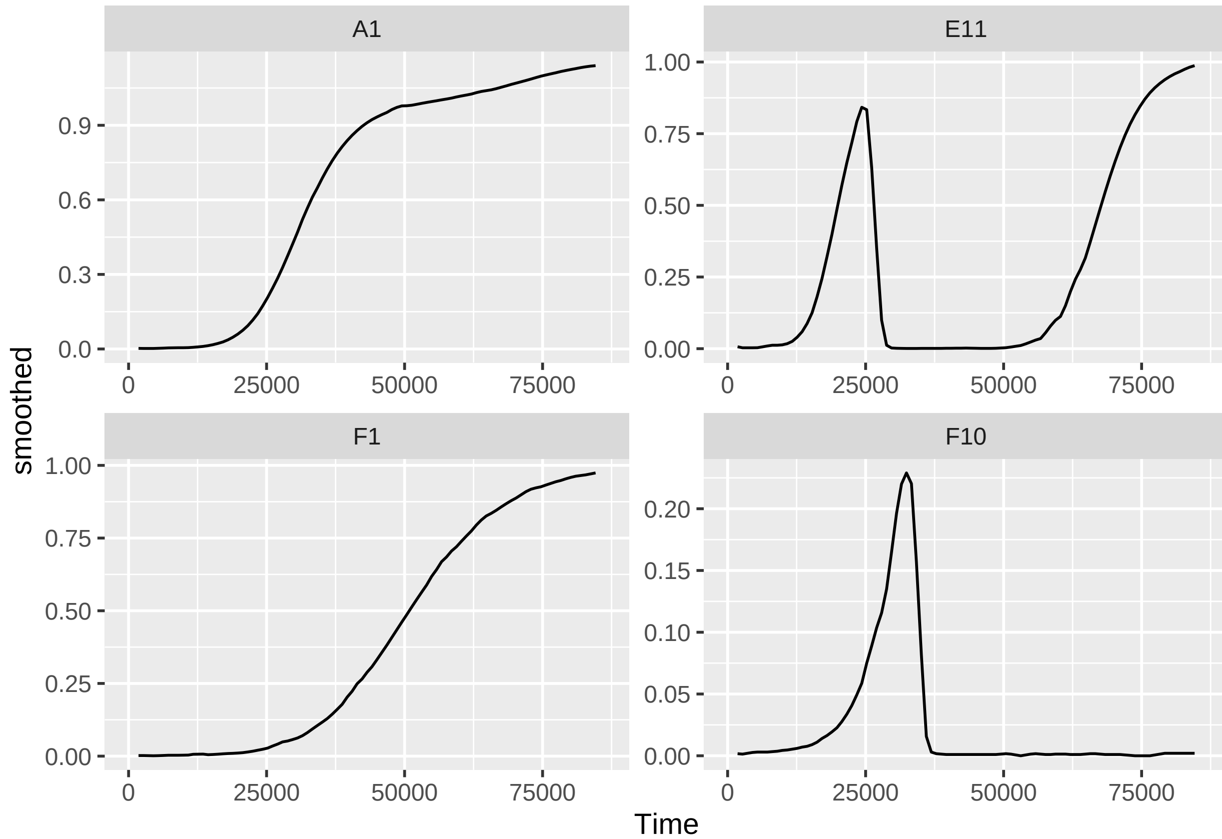
Let's look again at our smoothed per-capita growth rates:

```
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv_percap_hr)) +
  geom_line() +
  facet_wrap(~Well, scales = "free")
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```



Hmmm, there's a lot of noise in these plots, what's going on? We can begin to understand if we also look at our smoothed density values:

```
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well, scales = "free")
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



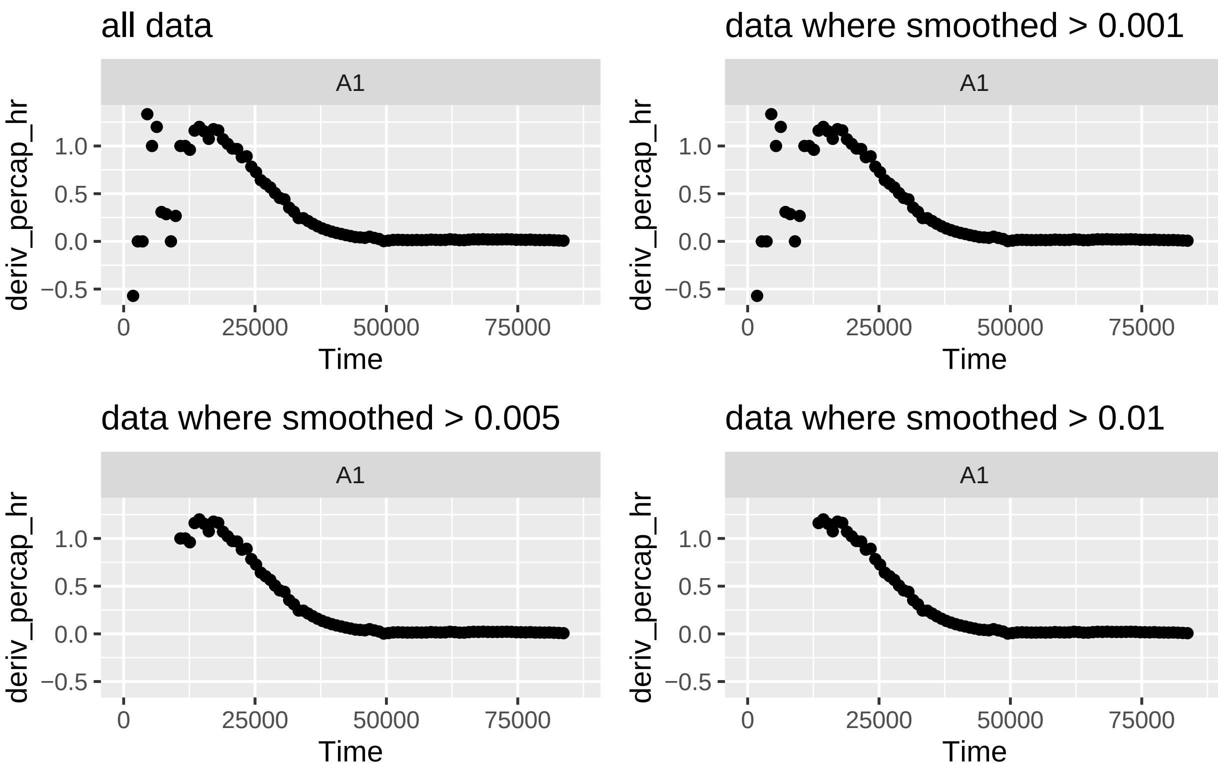
If we compare these plots with the previous ones, we can begin to see that most of the noise is arising when the bacterial populations are very small. Indeed, **this is common with per-capita growth rates, which are very sensitive to noise at low densities**. What can we do about it? We can simply exclude all the values when the *density* is really low.

Let's plot our per-capita growth rate data at different cutoffs for the minimum *density* of bacteria. Even though these are smoothed values, we'll use points here, since it better showcases where data are being excluded:

```
for (my_well in sample_wells) {
  #Title
  title <- cowplot::ggdraw() +
    cowplot::draw_label(paste("Well", my_well),
                        fontface = "bold", x = 0, hjust = 0) +
    theme(plot.margin = margin(0, 0, 0, 7))

  #Save x and y limits for all plots so they're all on the same axes
  xdat <- dplyr::filter(ex_dat_mrg, Well == my_well)$Time
  ydat <- dplyr::filter(ex_dat_mrg, Well == my_well)$deriv_percap_hr
  xlims <- c(min(xdat[is.finite(xdat)], na.rm = TRUE),
             max(xdat[is.finite(xdat)], na.rm = TRUE))
  ylims <- c(min(ydat[is.finite(ydat)], na.rm = TRUE),
             max(ydat[is.finite(ydat)], na.rm = TRUE))

  #Plot unfiltered data
  p1 <- ggplot(data = dplyr::filter(ex_dat_mrg, Well == my_well),
               aes(x = Time, y = deriv_percap_hr)) +
```

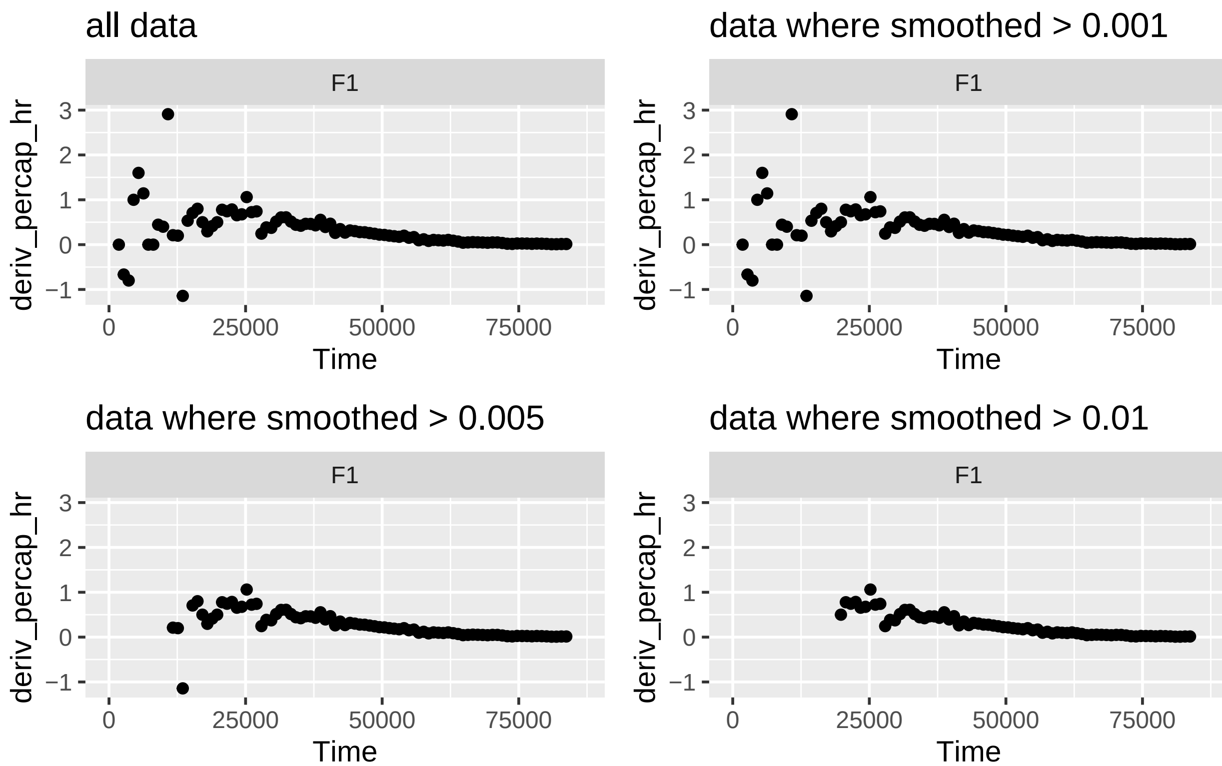
```
    geom_point() + facet_wrap(~Well, scales = "free") +
    ggtitle("all data") +
    xlim(xlims[1], xlims[2]) + ylim(ylims[1], ylims[2])

  #Plot data with filters for density
  p2 <- ggplot(data = dplyr::filter(ex_dat_mrg,
                                    Well == my_well, smoothed > 0.001),
               aes(x = Time, y = deriv_percap_hr)) +
    geom_point() + facet_wrap(~Well, scales = "free") +
    ggtitle("data where smoothed > 0.001") +
    xlim(xlims[1], xlims[2]) + ylim(ylims[1], ylims[2])
  p3 <- ggplot(data = dplyr::filter(ex_dat_mrg,
                                    Well == my_well, smoothed > 0.005),
               aes(x = Time, y = deriv_percap_hr)) +
    geom_point() + facet_wrap(~Well, scales = "free") +
    ggtitle("data where smoothed > 0.005") +
    xlim(xlims[1], xlims[2]) + ylim(ylims[1], ylims[2])
  p4 <- ggplot(data = dplyr::filter(ex_dat_mrg,
                                    Well == my_well, smoothed > 0.01),
               aes(x = Time, y = deriv_percap_hr)) +
    geom_point() + facet_wrap(~Well, scales = "free") +
    ggtitle("data where smoothed > 0.01") +
    xlim(xlims[1], xlims[2]) + ylim(ylims[1], ylims[2])

  print(cowplot::plot_grid(title, cowplot::plot_grid(p1, p2, p3, p4, ncol = 2),
                           ncol = 1, rel_heights = c(0.1, 1)))
}
#> Warning: Removed 5 rows containing missing values (geom_point).
#> Warning: Removed 1 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
#> Warning: Removed 5 rows containing missing values (geom_point).
#> Warning: Removed 1 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
```

Well A1



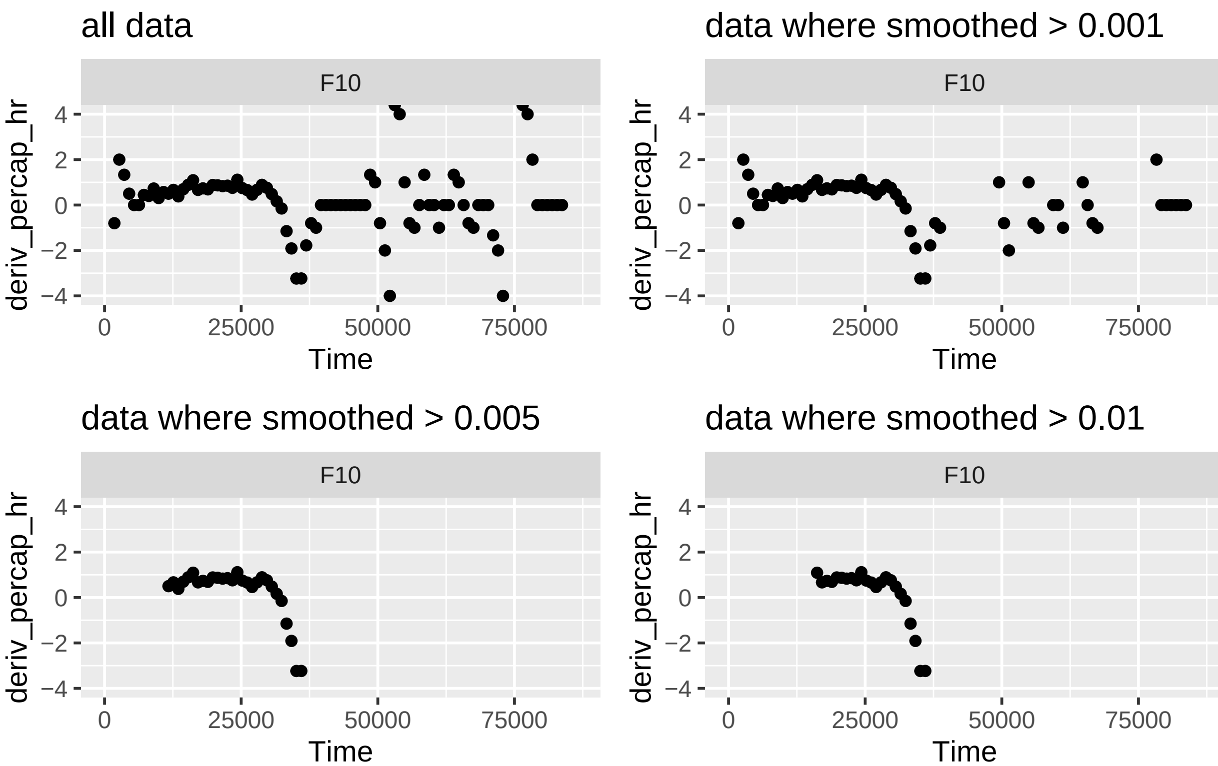

```
#> Warning: Removed 8 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
```

Well F1

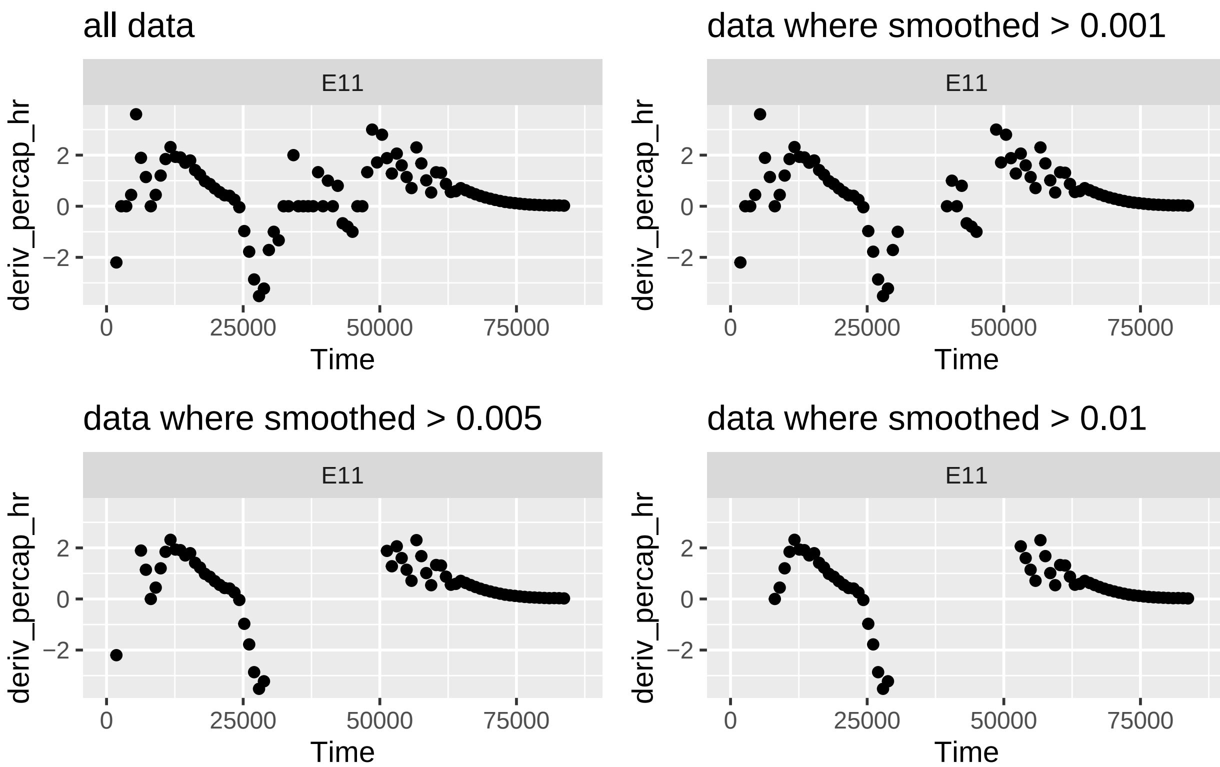



```
#> Warning: Removed 5 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
#> Removed 1 rows containing missing values (geom_point).
```

Well F10


all data
data where smoothed > 0.001
data where smoothed > 0.005
data where smoothed > 0.01
F10
deriv_percap_hr
4
2
0
−2
−4
0
25000
50000
75000
Time

Well E11

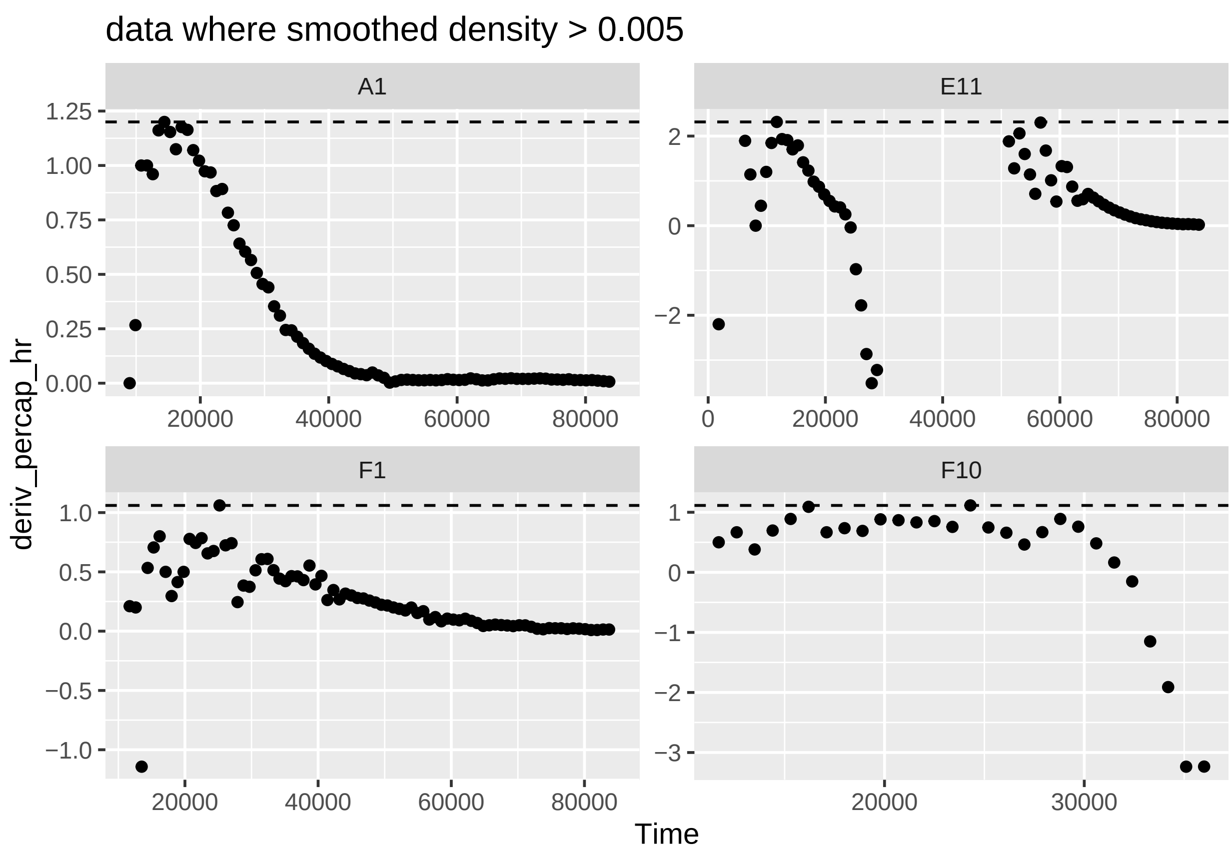


We can see with a cutoff of 0.001, much of the noise still remains. However, once we use a cutoff of 0.005, they are all basically gone, and no high growth rate values are affected by 0.005 vs 0.01. If we checked this pattern for all the wells (as you should in your own analyses), we would see a similar result. **Now, let's calculate the maximum growth rate of just the subset** of data points where OD is above 0.005. We can specify that subset directly in the summarize command:

```
ex_dat_mrg_sum <-
  summarize(grouped_ex_dat_mrg,
            max_growth_rate = max(deriv_percap_hr[smoothed > 0.005],
                                  na.rm = TRUE))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(ex_dat_mrg_sum)
#> # A tibble: 6 x 4
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  max_growth_rate
#>   <chr>           <chr>       <fct>           <dbl>
#> 1 Strain 1        No Phage    A1               1.20
#> 2 Strain 1        Phage Added A7               2.71
#> 3 Strain 10       No Phage    B4               2.84
#> 4 Strain 10       Phage Added B10              3.48
#> 5 Strain 11       No Phage    B5               2.22
#> 6 Strain 11       Phage Added B11              3.36
```

And now we can visualize our findings:

```r
ggplot(data = dplyr::filter(ex_dat_mrg,
                            Well %in% sample_wells, smoothed >= 0.005),
       aes(x = Time, y = deriv_percap_hr)) +
  geom_point() +
  facet_wrap(~Well, scales = "free") +
  ggtitle("data where smoothed density > 0.005") +
  geom_hline(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(yintercept = max_growth_rate), lty = 2)
#> Warning: Removed 3 rows containing missing values (geom_point).
```



Finding local extrema: peak density, maximum growth rate, lag time, and diauxic shifts

We've previously shown how you can use `max` and `min` to find the global maxima and minima in data. However, what about *local* maxima or minima? That is, peaks and valleys that are obvious to the eye but aren't the highest or smallest values in the entire time series. In this section, we'll show how you can use the `gcplyr` functions `first_peak` and `find_local_extrema` to find points that are local maxima or minima in your data.

Finding the first peak: peak density, maximum growth rate, and lag time

One particular special case we're often interested in is the first peak in some set of data. For instance, when bacteria are grown with phages, the density they reach before they start declining due to phage predation (a measure of their susceptibility to the phage)? Alternatively, in the previous section we found the global maximum per-capita growth rate, but some of these maxima happened after near-extinction and recovery. What if we wanted to find the peak growth rate before near-extinction?

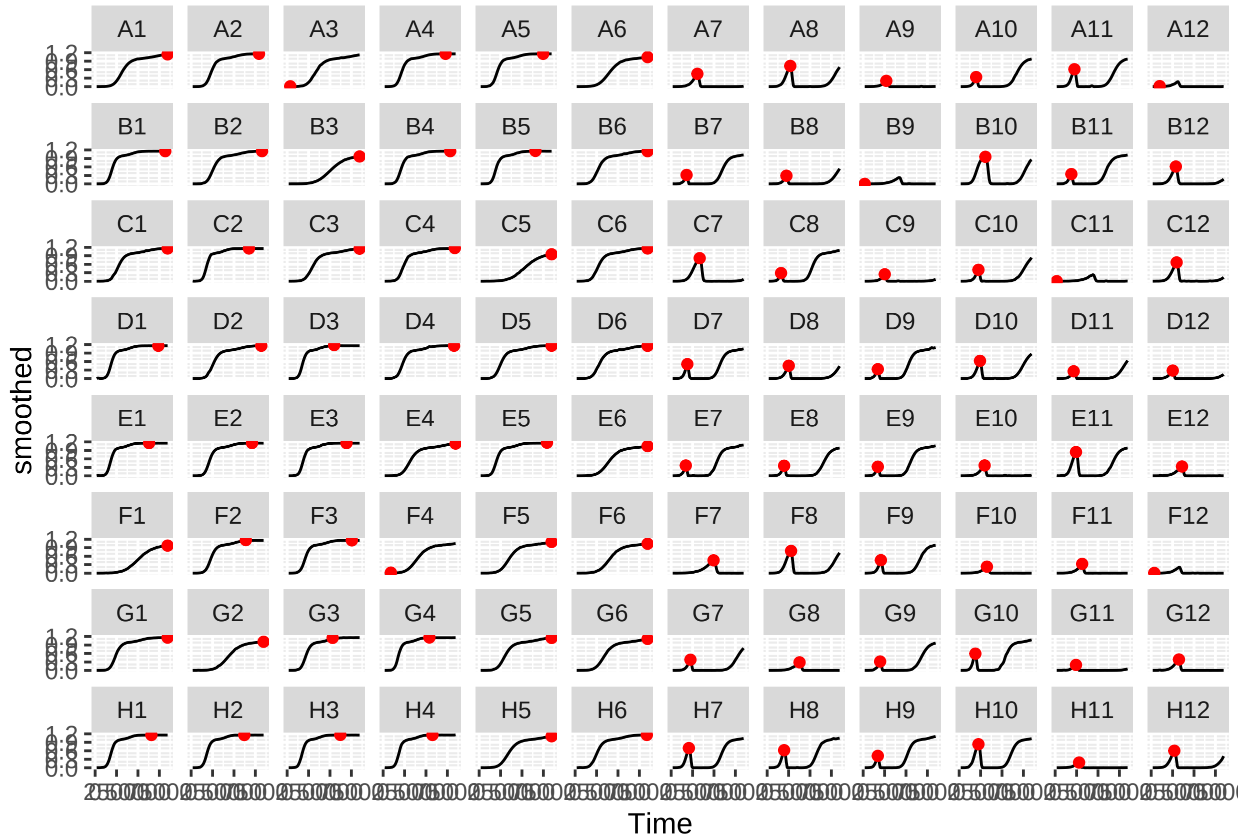
Peak density Let's start with the former example: finding the peak of density.

To identify the first peak, we can use the `gcplyr` function `first_peak`. `first_peak` simply requires the `y` data you want to identify the peak in. In this case, that's `smoothed`. We also need to specify whether we want the function to `return` the `index` of the first peak, the `x` value of the peak, or the `y` value of the peak. We'll get the `x` and `y` values, saving them in columns `first_peak_x` and `first_peak_y`, respectively. (Note that if you want the x-value, you have to provide the x values to `first_peak`). As usual, `first_peak` needs to be used inside of a `summarize` command on data that has already been grouped.

```
ex_dat_mrg_sum <-
  summarize(grouped_ex_dat_mrg,
            first_peak_x = first_peak(x = Time, y = smoothed, return = "x"),
            first_peak_y = first_peak(y = smoothed, return = "y"))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.

head(ex_dat_mrg_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  first_peak_x first_peak_y
#>   <chr>           <chr>       <fct>        <dbl>        <dbl>
#> 1 Strain 1        No Phage    A1           84600        1.14
#> 2 Strain 1        Phage Added A7           30600        0.453
#> 3 Strain 10       No Phage    B4           78300        1.16
#> 4 Strain 10       Phage Added B10          30600        0.959
#> 5 Strain 11       No Phage    B5           65700        1.17
#> 6 Strain 11       Phage Added B11          18900        0.348
```

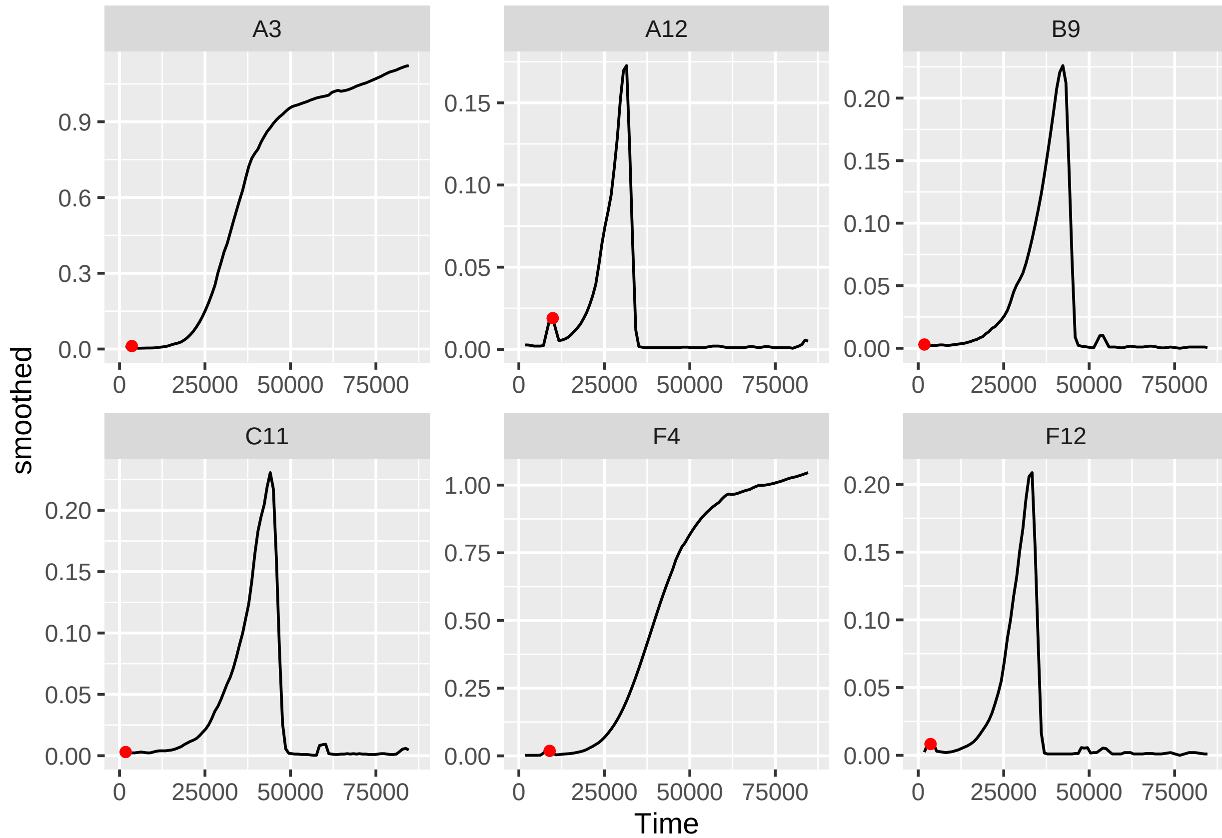
Let's plot these points on all the wells to confirm they are what we expect:

```
ggplot(data = ex_dat_mrg, aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well, nrow = 8, ncol = 12) +
  geom_point(data = ex_dat_mrg_sum,
             aes(x = first_peak_x, y = first_peak_y),
             color = "red", size = 1.5)
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



Hmmm, in most of the wells `first_peak` worked perfectly well. However, a few of the wells aren't quite what we'd expect. Let's take a closer look at them:

```
wells_tocheck <- c("A3", "A12", "B9", "C11", "F4", "F12")
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% wells_tocheck),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well, scales = "free") +
  geom_point(data = dplyr::filter(ex_dat_mrg_sum, Well %in% wells_tocheck),
             aes(x = first_peak_x, y = first_peak_y),
             color = "red", size = 1.5)
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



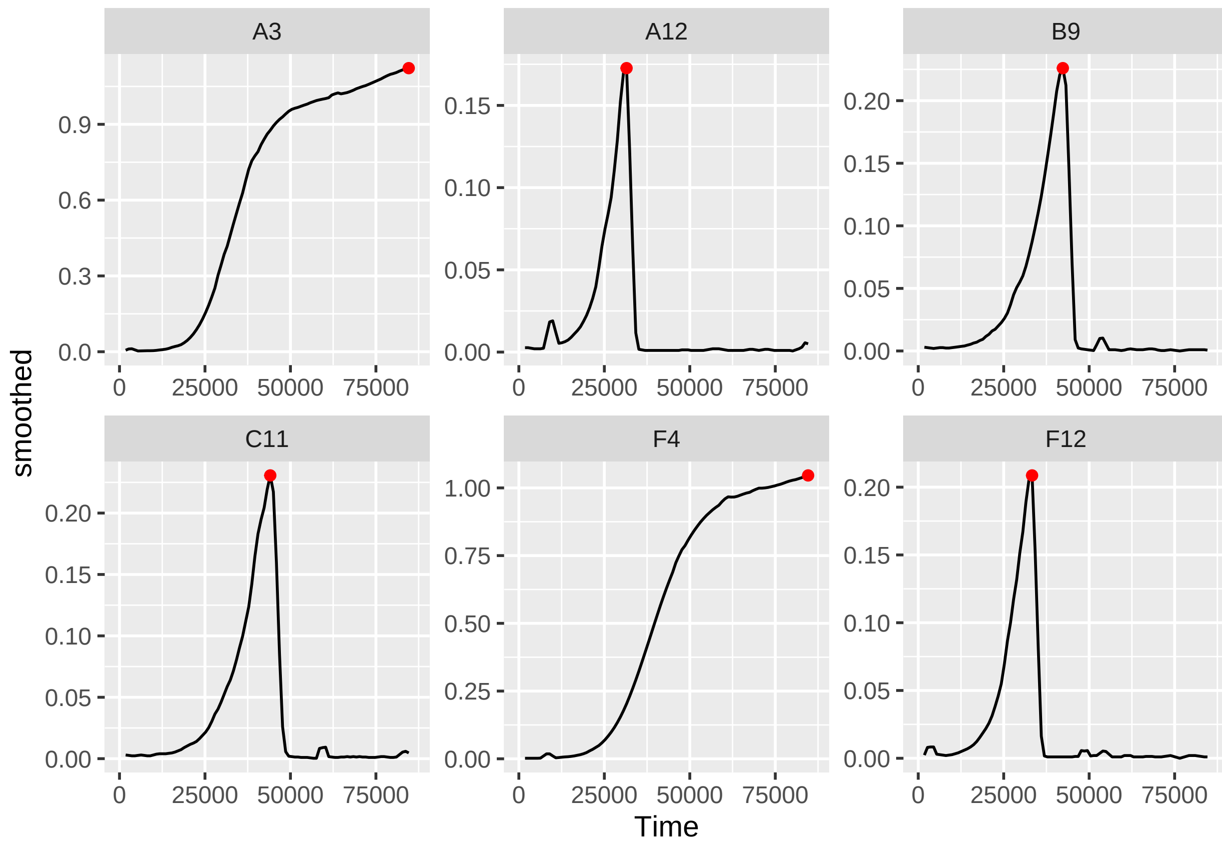
Now we can see what's going on. In these wells, `first_peak` seems to have 'gotten stuck' on some earlier smaller peaks. Just like in smoothing, **peak-finding also has tuning parameters**. For `first_peak` and `find_local_extrema`, these are `width_limit_n`, `width_limit` and `height_limit`:

- `width_limit` determines the width of the window used to search for peaks and valleys, in units of `x`
- `width_limit_n` determines the width of the window, in units of number of data points
- `height_limit` determines the shortest peak or shallowest valley the window will cross, in units of `y`

If we want `first_peak` to be less sensitive to local peaks, we can increase these parameters (the default setting is `width_limit_n` equal to 20% of the length of `y`, but `width_limit` is a better approach since it works in units of seconds). Let's try that:

```
ex_dat_mrg_sum <-
  summarize(grouped_ex_dat_mrg,
            first_peak_x = first_peak(x = Time, y = smoothed, return = "x",
                                      width_limit = 35000),
            first_peak_y = first_peak(x = Time, y = smoothed, return = "y",
                                      width_limit = 35000))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.

ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% wells_tocheck),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well, scales = "free") +
  geom_point(data = dplyr::filter(ex_dat_mrg_sum, Well %in% wells_tocheck),
```

```
             aes(x = first_peak_x, y = first_peak_y),
             color = "red", size = 1.5)
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```

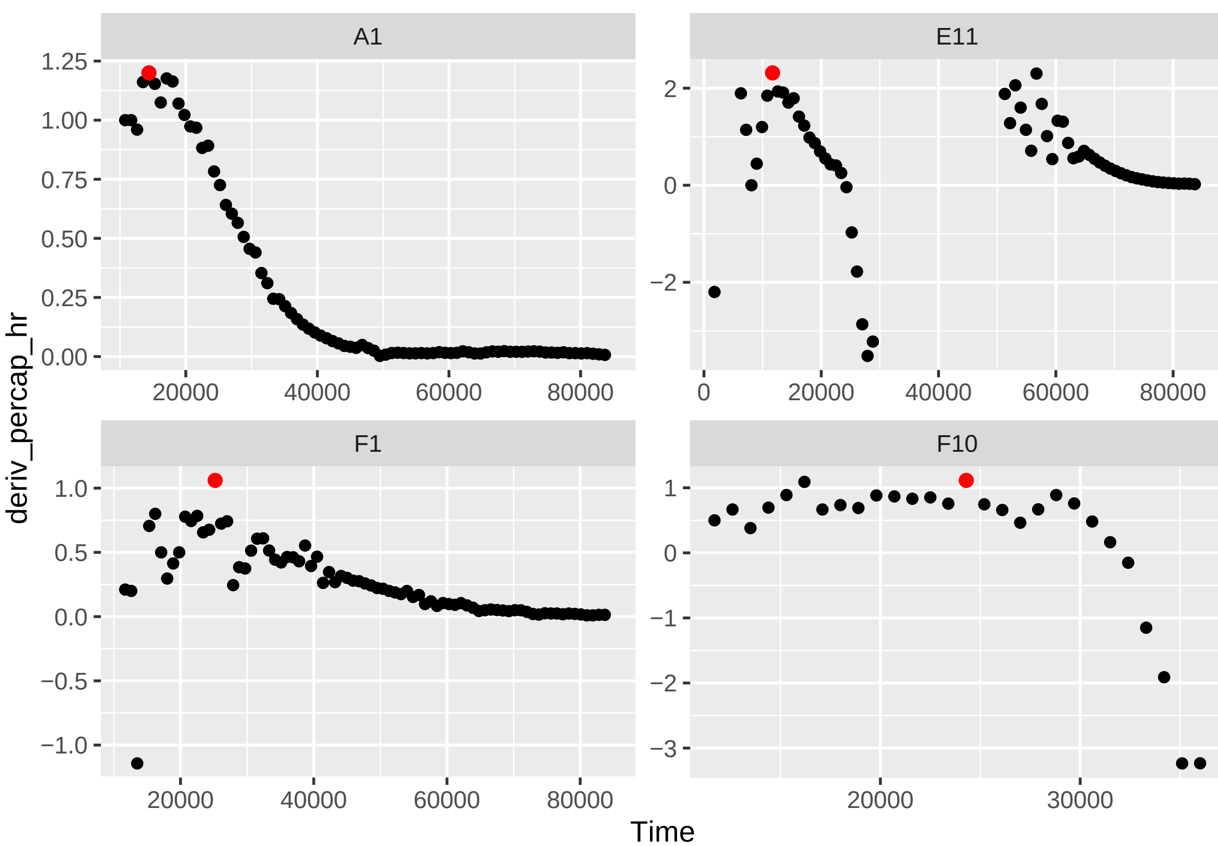


That worked great!

Maximum growth rate and lag time Now let's look at the other example: using `first_peak` to find the first peak in per-capita growth rate to find both the maximum growth rate and the lag time. As we did earlier, we'll limit our analyses to data where `smoothed > 0.005`, and visualize using points (even though this is smoothed):

```
ex_dat_mrg_sum <-
  summarize(grouped_ex_dat_mrg,
            max_growth_rate = first_peak(x = Time[smoothed > 0.005],
                                         y = deriv_percap_hr[smoothed > 0.005],
                                         return = "y", width_limit = 35000),
            lag_time = first_peak(x = Time[smoothed > 0.005],
                                  y = deriv_percap_hr[smoothed > 0.005],
                                  return = "x", width_limit = 35000))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.

head(ex_dat_mrg_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
```

```
#>   Bacteria_strain Phage        Well  max_growth_rate lag_time
#>   <chr>           <chr>        <fct>           <dbl>    <dbl>
#> 1 Strain 1        No Phage     A1               1.20    14400
#> 2 Strain 1        Phage Added A7               1.76    16200
#> 3 Strain 10       No Phage     B4               2.84    10800
#> 4 Strain 10       Phage Added B10              2.14    10800
#> 5 Strain 11       No Phage     B5               2.22    11700
#> 6 Strain 11       Phage Added B11              3.36     9000

ggplot(data = dplyr::filter(ex_dat_mrg,
                            Well %in% sample_wells, smoothed > 0.005),
       aes(x = Time, y = deriv_percap_hr)) +
  geom_point() +
  facet_wrap(~Well, scales = "free") +
  geom_point(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(x = lag_time, y = max_growth_rate),
             color = "red", size = 2)
#> Warning: Removed 3 rows containing missing values (geom_point).
```



Here we can see that in Well E11, `first_peak` has identified the peak growth rate at the beginning of the dynamics, and not the one that occurs later on. This means that our `lag_time` value will actually reflect what we want it to.

But what if you want to find an extrema that's *not* the first peak? In the next section, we'll learn how to use `find_local_extrema` to identify all kinds of local extrema.

Finding any kind of local extrema: diauxic shifts

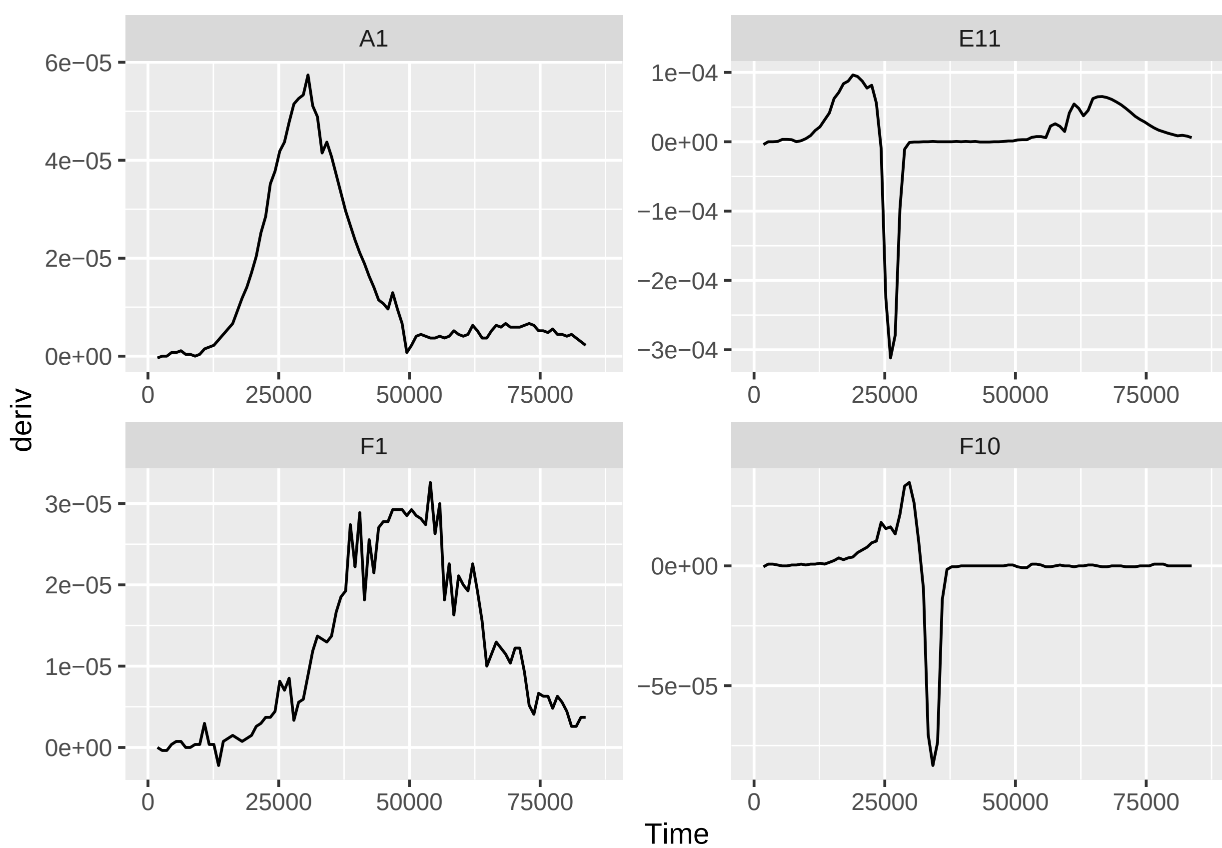
We've seen how `first_peak` can be used to identify the first peak in data. But what about other kinds of local extrema? The first minimum? The *second* peak?

In order to identify these kinds of extrema, we can use the more-general function `find_local_extrema`. In fact, `first_peak` is really just a special case of `find_local_extrema`. Just like `first_peak`, `find_local_extrema` only requires a vector of y data in which to find the local extrema, and can return the `index`, `x` value, or `y` of the extrema it finds.

Unlike `first_peak`, `find_local_extrema` returns a vector containing *all* of the local extrema found under the given settings. Users can alter which kinds of local extrema are reported using the arguments `return_maxima`, `return_minima`, and `return_endpoints`. However, `find_local_extrema` will always return a vector of all the extrema found, so users must use brackets to select which one they want `summarize` to save.

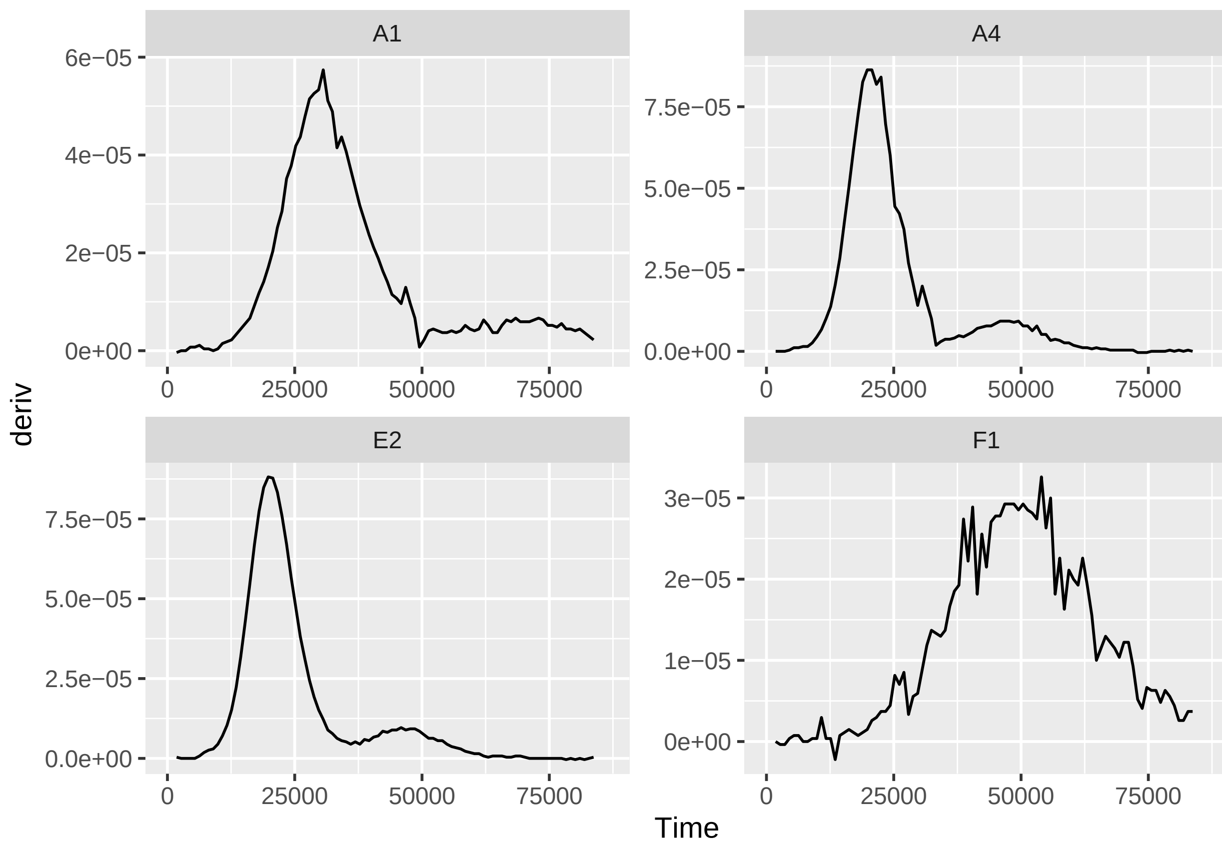
Let's dig into an example: identifying diauxic shifts. To refresh your memory on what we saw in the section A simple derivative, here's a plot of the derivative of some of the wells over time.

```
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv)) +
  geom_line() +
  facet_wrap(~Well, scales = "free")
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```



In fact, if we look at some more of the wells with no phage added, we'll see a similar pattern repeatedly.

```r
sample_wells <- c("A1", "A4", "E2", "F1")
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv)) +
  geom_line() +
  facet_wrap(~Well, scales = "free")
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```

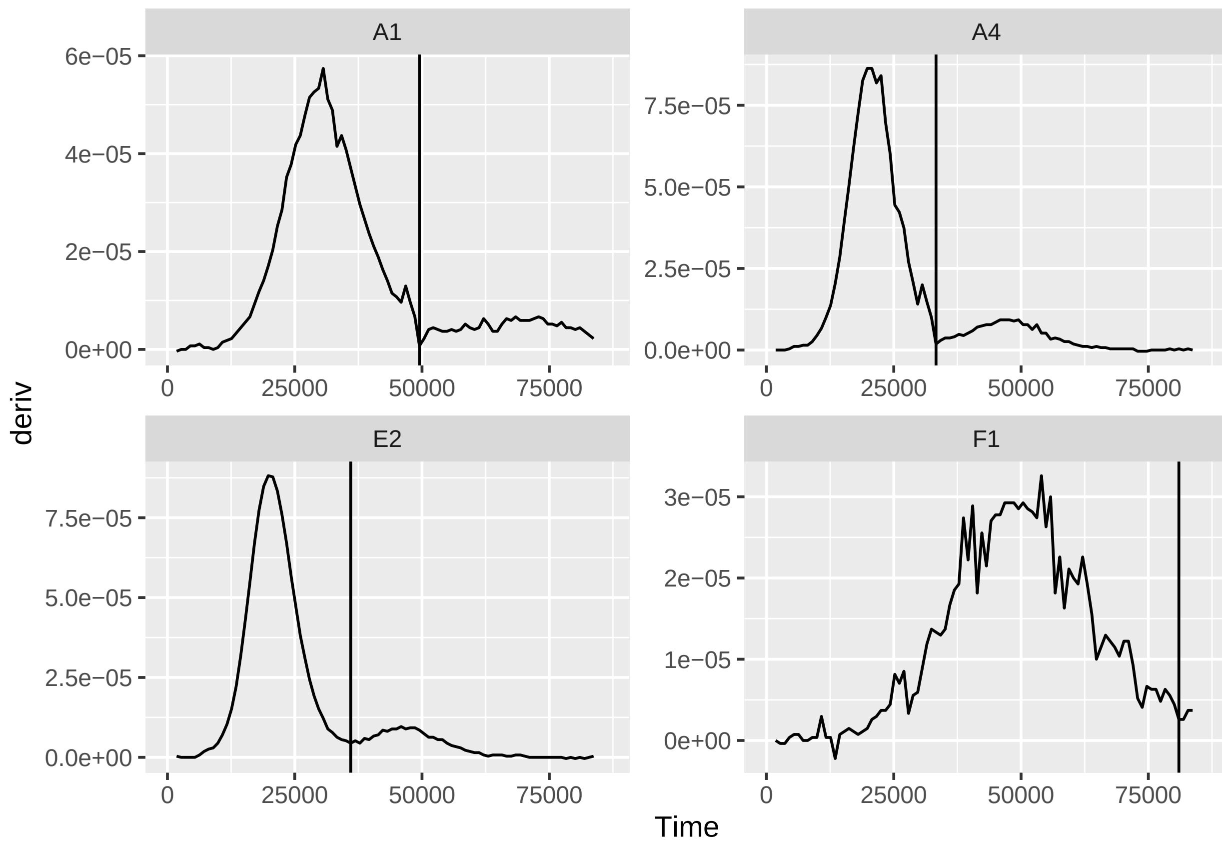


This second, slower, burst of growth after the first wave of growth is common in bacterial growth curves and is called *diauxic growth*.

How could we identify the time when the bacteria switch from their first burst of growth to their second? We can find the first minima (that isn't just the start) in the `deriv` values. To do so, we specify to `find_local_extrema` that we want `return = "x"` and we don't want maxima returned:

```r
ex_dat_mrg_sum <-
  summarize(
    grouped_ex_dat_mrg,
    diauxie_time = find_local_extrema(x = Time, y = deriv, return = "x",
                                      return_maxima = FALSE, return_minima = TRUE,
                                      width_limit_n = 39)[2])
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.

#Plot data with vertical line at detected diauxie
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv)) +
```

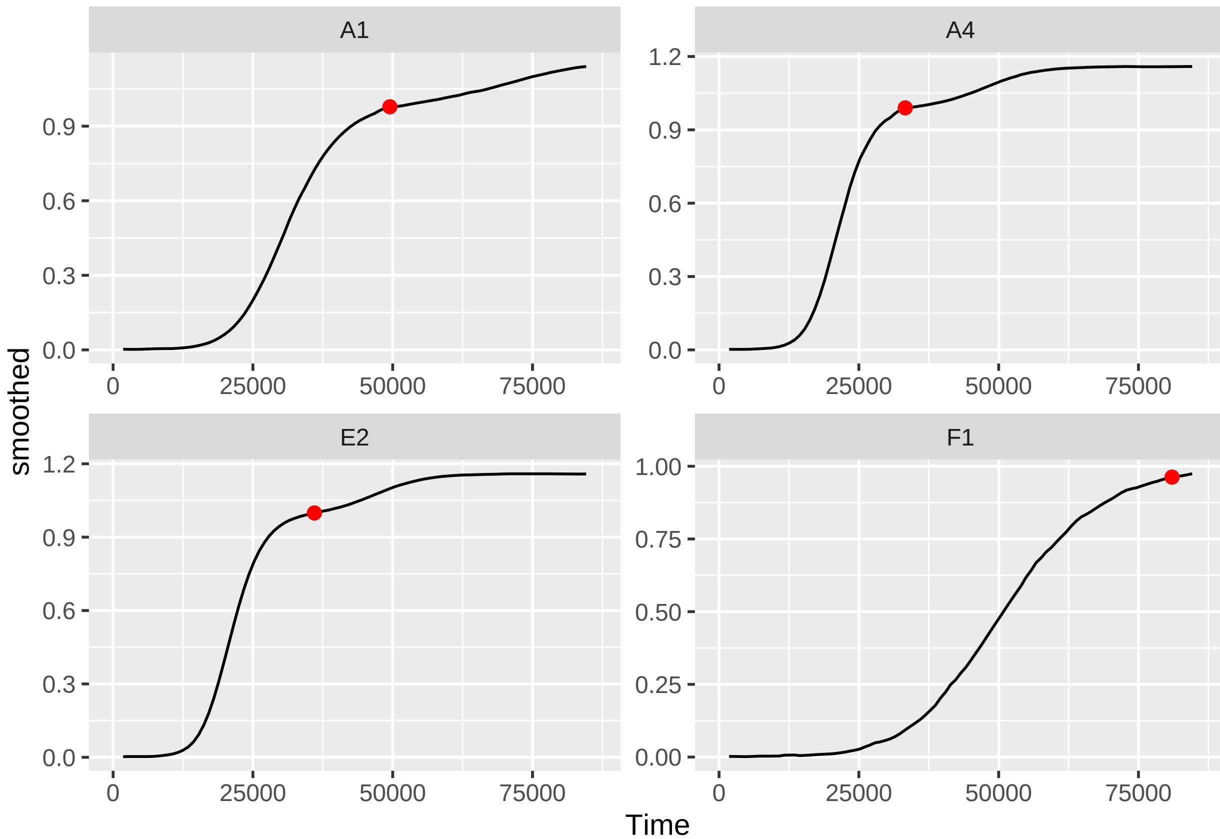
```
  geom_line() +
  facet_wrap(~Well, scales = "free") +
  geom_vline(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(xintercept = diauxie_time))
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```



Now that we've found the point where the bacteria switch, we could quite easily find the density where that happens. To make it easier to follow, we'll save the *index* where the diauxic shift occurs to a column titled `diauxie_idx`. To get that, we simply run `find_local_extrema` with `return = "index"`. Then, we can get the `smoothed` value at that index:

```
ex_dat_mrg_sum <-
  summarize(
    grouped_ex_dat_mrg,
    diauxie_time = find_local_extrema(x = Time, y = deriv, return = "x",
                                      return_maxima = FALSE, return_minima = TRUE,
                                      width_limit_n = 39)[2],
    diauxie_idx = find_local_extrema(x = Time, y = deriv, return = "index",
                                      return_maxima = FALSE, return_minima = TRUE,
                                      width_limit_n = 39)[2],
    diauxie_dens = smoothed[diauxie_idx])
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.

#Plot data with a point at the moment of diauxic shift
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
```

```
      aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well, scales = "free") +
  geom_point(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(x = diauxie_time, y = diauxie_dens),
             size = 2, color = "red")
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



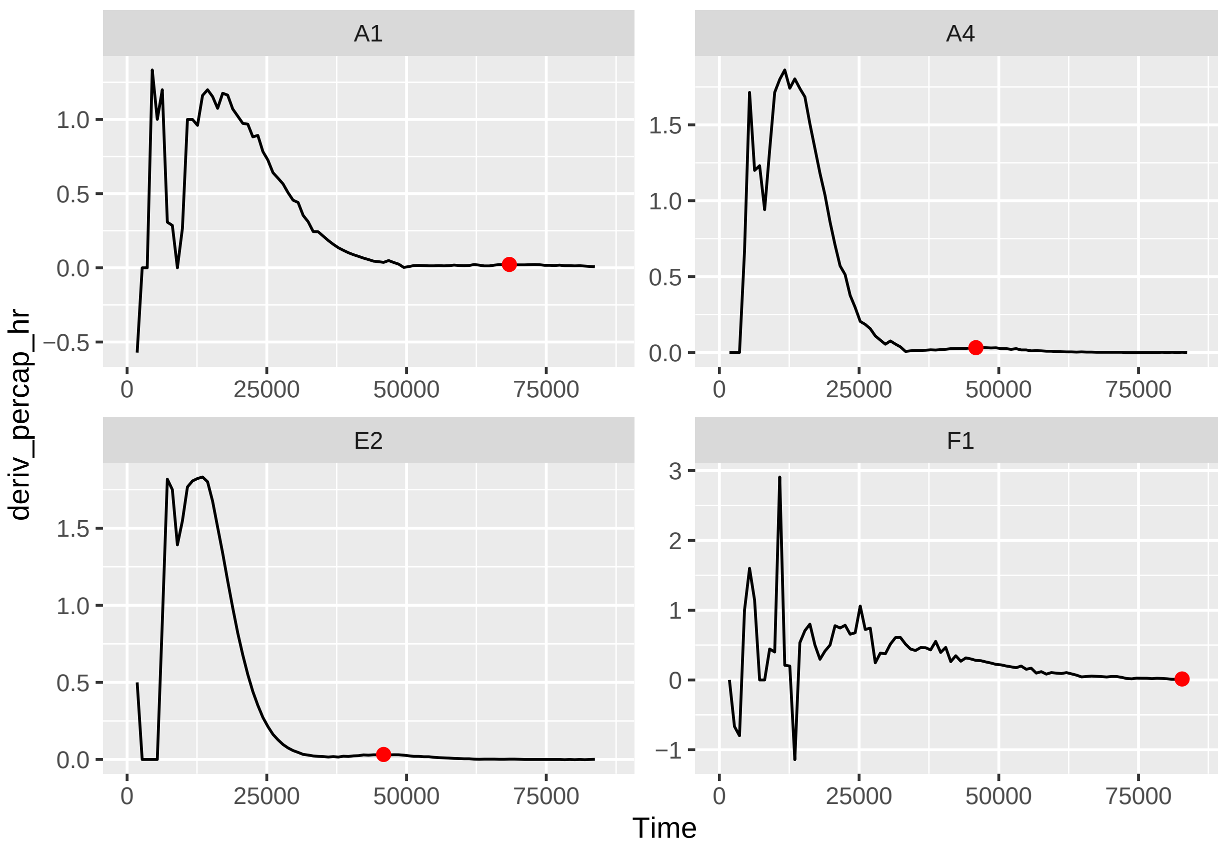
Something that was hard to see on the density plot has now been easily quantified and can be visualized exactly where the shift occurs.

Combining subsets and local extrema: diauxic growth rate

In the previous section we identified when the bacteria shifted into their second burst of growth. Can we find out what the peak per-capita growth rate was during that second burst? Yes, we just have to put together some of the things we've learned already. In particular, we're going to combine our use of `find_local_extrema`, `max`, and subsets to find the `max(deriv_percap_hr)` during the times after the diauxic shift:

```
ex_dat_mrg_sum <-
  summarize(
    grouped_ex_dat_mrg,
    diauxie_time = find_local_extrema(x = Time, y = deriv, return = "x",
                                      return_maxima = FALSE, return_minima = TRUE,
                                      width_limit_n = 39)[2],
```

```
    diauxie_percap = max(deriv_percap_hr[Time >= diauxie_time], na.rm = TRUE),
    diauxie_percap_time =
      Time[Time >= diauxie_time][
        which.max(deriv_percap_hr[Time >= diauxie_time])]
    )
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.

#Plot data with a point at the moment of peak diauxic growth rate
ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = deriv_percap_hr)) +
  geom_line() +
  facet_wrap(~Well, scales = "free") +
  geom_point(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(x = diauxie_percap_time, y = diauxie_percap),
             size = 2, color = "red")
#> Warning: Removed 5 row(s) containing missing values (geom_path).
```



Finding threshold-crossings: extinction time and time to density

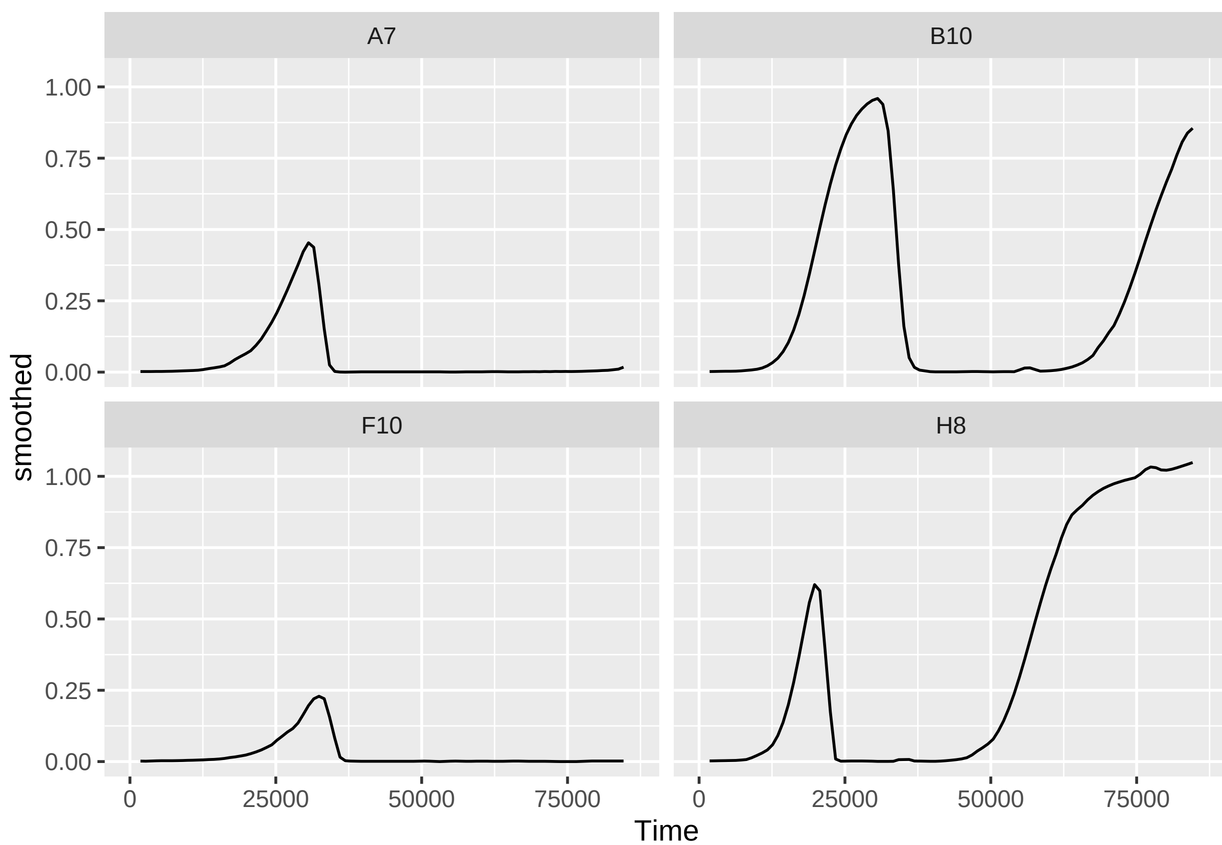
We've previously shown how you can find local and global extrema in data, but what if you just want to find when the data passes some threshold value? In this section, we'll show how you can use the `gcplyr` functions `first_below` and `find_threshold_crosses` to find the points when your data crosses user-defined thresholds.

Finding the first point below a threshold: extinction time

One common case of threshold-crossing we might be interested in is the first point that our data falls below some threshold density. For instance, when bacteria are grown with phages, the amount of time it takes before the bacterial population falls below some threshold can be a proxy metric for how sensitive the bacteria are to that phage.

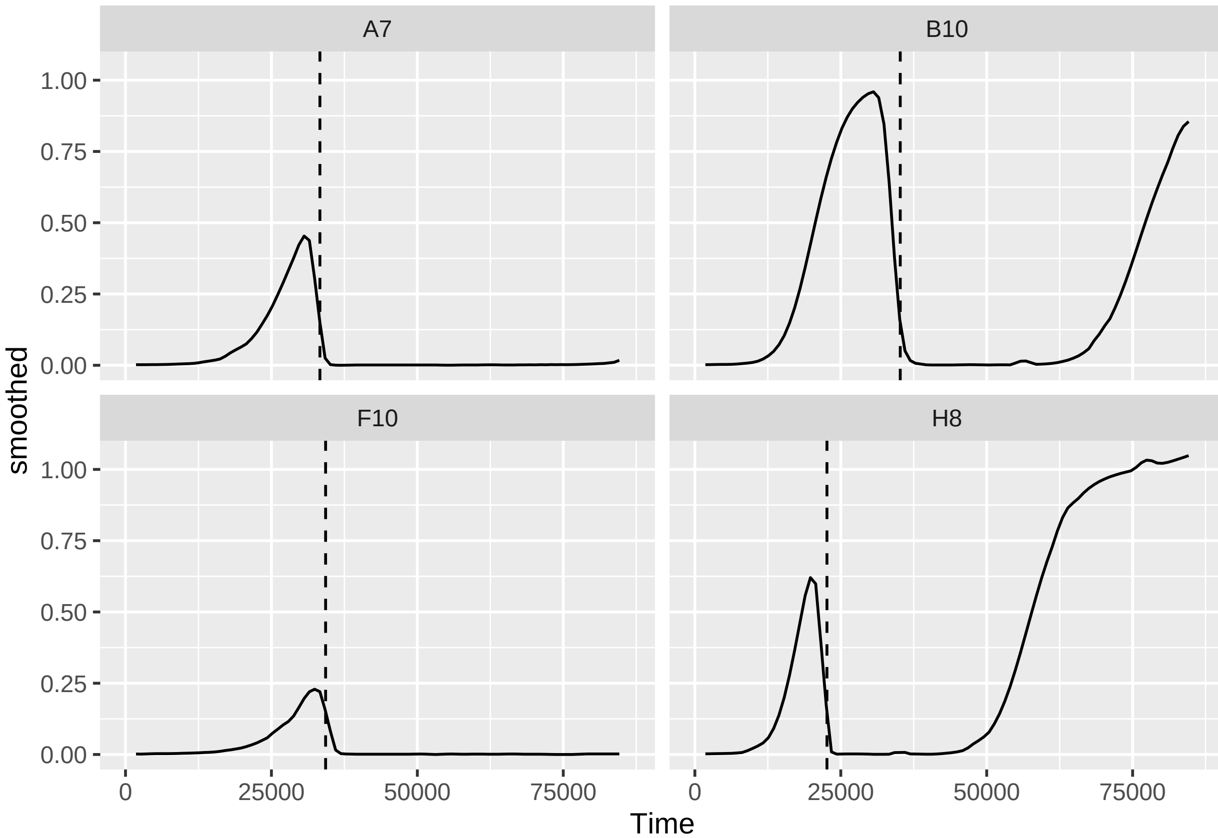
Let's take a look at the *smoothed* absorbance values in some example wells with both bacteria and phages:

```
sample_wells <- c("A7", "B10", "F10", "H8")

ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well)
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



Ok great. Now let's suppose that I think that an absorbance of 0.15 is a good threshold for extinction in my experiment. How could we use `first_below` to calculate the time when that first occurs across all our different wells? Well, primarily, `first_below` simply needs our `x` and `y` values, the `threshold` we want to use, as well as whether we want it to `return` the `index` of the first point below the threshold, or the `x` value of that point (since we care about the time it happened here, we'll do the latter). Additionally, we'll specify that we don't care if the startpoint is below the threshold: we only care when the data goes from above to below it.

```
ex_dat_mrg_sum <-
  summarize(
    grouped_ex_dat_mrg,
    extin_time = first_below(x = Time, y = smoothed, threshold = 0.15,
                             return = "x", return_endpoints = FALSE))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(ex_dat_mrg_sum)
#> # A tibble: 6 x 4
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  extin_time
#>   <chr>           <chr>       <fct>      <dbl>
#> 1 Strain 1        No Phage    A1            NA
#> 2 Strain 1        Phage Added A7        33307.
#> 3 Strain 10       No Phage    B4            NA
#> 4 Strain 10       Phage Added B10       35187.
#> 5 Strain 11       No Phage    B5            NA
#> 6 Strain 11       Phage Added B11       20445.

ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well) +
  geom_vline(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(xintercept = extin_time), lty = 2)
#> Warning: Removed 4 row(s) containing missing values (geom_path).
```



All the phage-added wells have a time when the bacteria drop below that threshold, and the plot clearly shows that it's right where we'd expect it.

Finding any kind of threshold-crossing: time to density

We've seen how `first_below` can be used to identify the first point some data crosses below a threshold. But what about other kinds of threshold-crossing events? The first point it passes above a threshold? The first point it's ever below a threshold, including at the start?

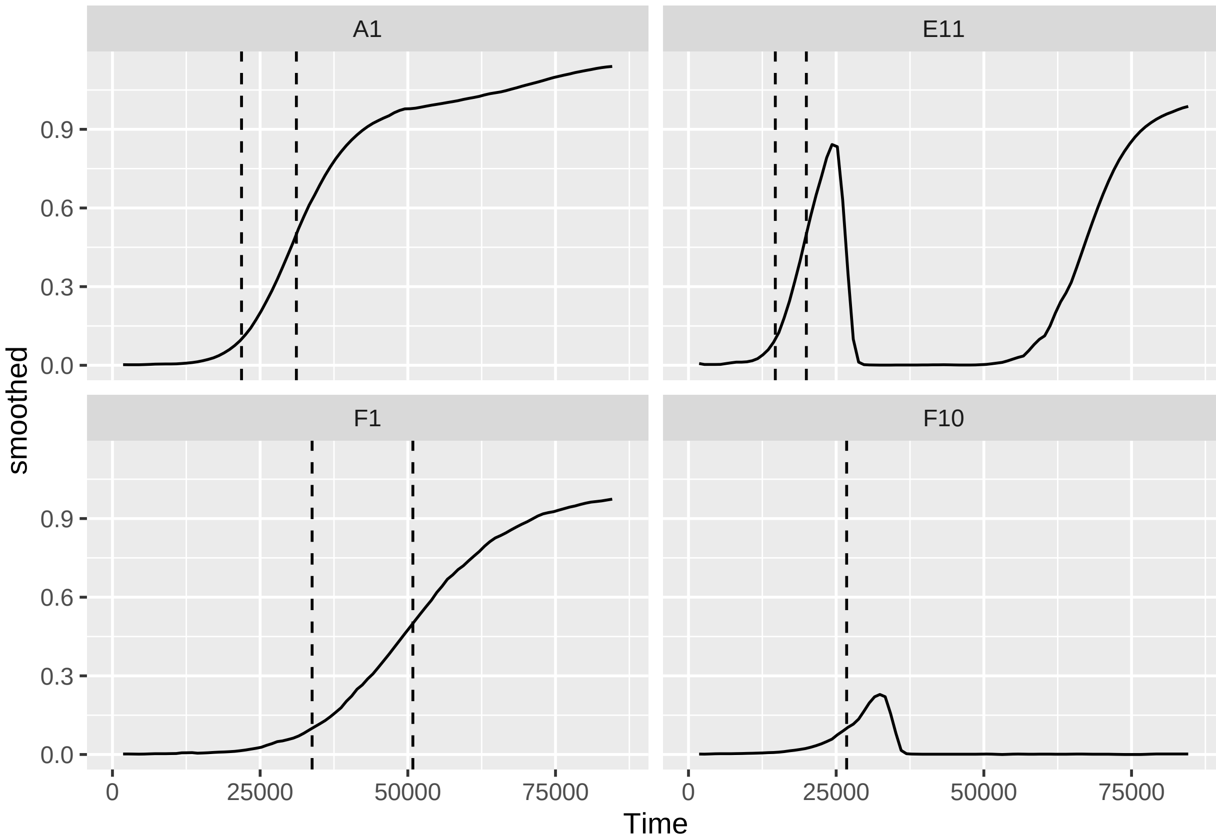
In order to identify these kinds of extrema, we can use the more-general function `find_threshold_crosses`. In fact, `first_below` is really just a special case of `find_threshold_crosses`. Just like `first_below`, `find_threshold_crosses` only requires a `threshold` and a vector of `y` data in which to find the threshold crosses, and can return the `index` or `x` value of the crossing events it finds.

However, unlike `first_below`, `find_threshold_crosses` returns a vector containing *all* of the threshold crossings found under the given settings. Users can alter which kinds of threshold crossings are reported using the arguments `return_rising`, `return_falling`, and `return_endpoints`. However, `find_threshold_crosses` will always return a vector of all the extrema found, so users must use brackets to select which one they want `summarize` to save.

Let's dig into an example: identifying the first time the bacteria reach some density, including if they start at that density

```
sample_wells <- c("A1", "F1", "F10", "E11")
ex_dat_mrg_sum <-
  summarize(
    grouped_ex_dat_mrg,
```

```
    time_to_01 = find_threshold_crosses(x = Time, y = smoothed,
                                        threshold = 0.1, return = "x",
                                        return_endpoints = TRUE,
                                        return_falling = FALSE)[1],
    time_to_05 = find_threshold_crosses(x = Time, y = smoothed,
                                        threshold = 0.5, return = "x",
                                        return_endpoints = TRUE,
                                        return_falling = FALSE)[1])
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(ex_dat_mrg_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  time_to_01 time_to_05
#>   <chr>           <chr>       <fct>      <dbl>      <dbl>
#> 1 Strain 1        No Phage    A1        21851.     31134.
#> 2 Strain 1        Phage Added A7        21855.        NA
#> 3 Strain 10       No Phage    B4        15178.     20629.
#> 4 Strain 10       Phage Added B10       15196.     20627.
#> 5 Strain 11       No Phage    B5        14434.     19326.
#> 6 Strain 11       Phage Added B11       14440.     59796.

ggplot(data = dplyr::filter(ex_dat_mrg, Well %in% sample_wells),
       aes(x = Time, y = smoothed)) +
  geom_line() +
  facet_wrap(~Well) +
  geom_vline(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(xintercept = time_to_01), lty = 2) +
  geom_vline(data = dplyr::filter(ex_dat_mrg_sum, Well %in% sample_wells),
             aes(xintercept = time_to_05), lty = 2)
#> Warning: Removed 4 row(s) containing missing values (geom_path).
#> Warning: Removed 1 rows containing missing values (geom_vline).
```



As we can see, `find_threshold_crosses` has returned the times when the bacteria reached those densities. We can see that some bacteria (e.g. those in Wells A7 and F10) never reached 0.5, so they have an `NA` value for `time_to_05`. By comparing the times it took each strain to reach an absorbance of 0.1, we could learn something about how soon the bacteria started growing and how quickly they grew.

Combining growth curves data with other data

As you approach the end of your growth curves analyses, you have likely summarized the dynamics of your growth curves into one or a few metrics. At this point, you may wish to pull in other sources of data to compare to your growth curves metrics. Just like merging multiple growth curves data frames together, this can be achieved with `merge_dfs`.

Let's use the `ex_dat_mrg_sum` from an earlier section, where we've summarized our growth curves using area-under-the-curve (although this approach would work with any number of summarized metrics).

```
ex_dat_mrg_sum <-
  summarize(grouped_ex_dat_mrg, auc = auc(x = Time, y = smoothed))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
```

Now imagine that, separately, we've measured the resistance of each of these bacteria to antibiotics, and we want to know if there's any relationship between the antibiotic resistance of the bacteria and their growth.

We're just going to focus on the bacterial growth in the absence of phage, so let's use `dplyr::filter` to remove the phage added rows.

```
ex_dat_mrg_sum <- dplyr::filter(ex_dat_mrg_sum, Phage == "No Phage")
head(ex_dat_mrg_sum)
#> # A tibble: 6 x 4
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage    Well     auc
#>   <chr>           <chr>    <fct>  <dbl>
#> 1 Strain 1        No Phage A1    54952.
#> 2 Strain 10       No Phage B4    69766.
#> 3 Strain 11       No Phage B5    71456.
#> 4 Strain 12       No Phage B6    61346.
#> 5 Strain 13       No Phage C1    61170.
#> 6 Strain 14       No Phage C2    73824.
```

Now, let's generate some mock antibiotic resistance data. The file containing the antibiotic resistance data should have the bacterial strain names under the same header `Bacterial_strain`, so that merge_dfs knows to match those two columns. We'll put whether or not the strain is resistant to the antibiotic under the `Antibiotic_resis` column, with a TRUE for resistance, and FALSE for sensitivity. **Don't worry exactly how this code works**, since it's just simulating data that you would have collected in the lab.

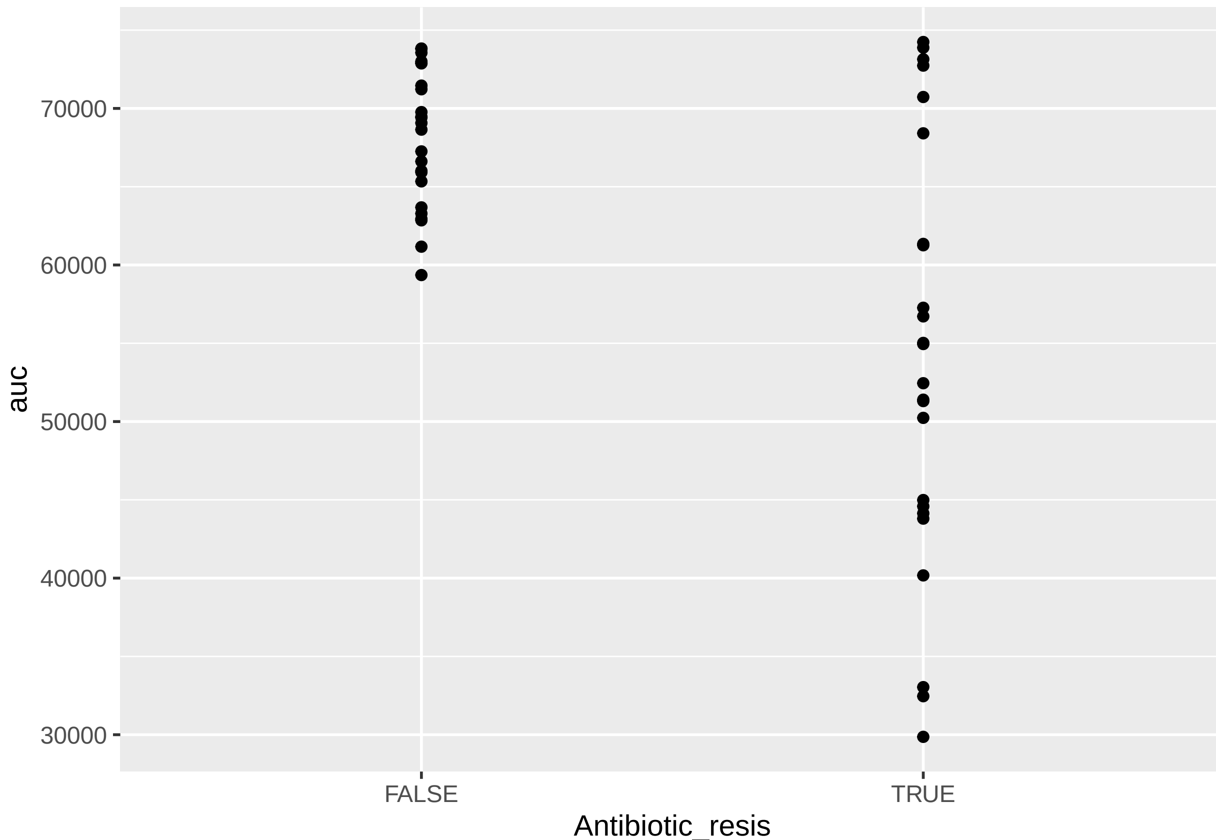
```
set.seed(123)
antibiotic_dat <-
  data.frame(Bacteria_strain = paste("Strain", 1:48),
             Antibiotic_resis =
               ex_dat_mrg_sum$auc[
                 match(paste("Strain", 1:48),
                       ex_dat_mrg_sum$Bacteria_strain)] *
               runif(48, 0.5, 1.5) < mean(ex_dat_mrg_sum$auc))

head(antibiotic_dat)
#>   Bacteria_strain Antibiotic_resis
#> 1        Strain 1             TRUE
#> 2        Strain 2            FALSE
#> 3        Strain 3             TRUE
#> 4        Strain 4            FALSE
#> 5        Strain 5            FALSE
#> 6        Strain 6             TRUE
```

Great, now we merge our two data frames.

```
growth_and_antibiotics <-
  merge_dfs(ex_dat_mrg_sum, antibiotic_dat)
#> Joining, by = "Bacteria_strain"
head(growth_and_antibiotics)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage    Well     auc Antibiotic_resis
#>   <chr>           <chr>    <fct>  <dbl> <lgl>
#> 1 Strain 1        No Phage A1    54952. TRUE
#> 2 Strain 10       No Phage B4    69766. FALSE
#> 3 Strain 11       No Phage B5    71456. FALSE
#> 4 Strain 12       No Phage B6    61346. TRUE
#> 5 Strain 13       No Phage C1    61170. FALSE
#> 6 Strain 14       No Phage C2    73824. FALSE
```

And now let's see if there's a relationship!

```
ggplot(data = growth_and_antibiotics,
       aes(x = Antibiotic_resis, y = auc)) +
  geom_point()
```



There is! We can see that the antibiotic resistant strains (`TRUE`) have a smaller area-under-the-curve than the antibiotic sensitive strains (`FALSE`) (although, to be fair, I did simulate the data so we'd get that result).

Other growth curve analysis packages

A number of other `R` packages besides `gcplyr` facilitate analysis of growth curves data.

There are, broadly speaking, two ways to analyze growth curves data:

1. directly quantify attributes of the growth dynamics
2. fit the growth dynamics with a mathematical model, then extract parameters from the fitted model

While `gcplyr` focuses on manipulation of growth curves data and the first analysis approach (direct quantification of growth curves dynamics), many other `R` packages focus on fitting growth dynamics with a mathematical model.

Generally, fitting growth dynamics with a model has greater power to accurately quantify the underlying traits. However, it also takes much more effort to be rigorous when fitting data with a model. You have to carefully choose a model whose assumptions your data meet. You also have to evaluate the fits to ensure that the optimization algorithms arrived on reasonable solutions.

A number of `R` packages implement fitting-style approaches, which I list here for readers to explore on their own. At some point in the future, I hope to incorporate more direct examples of how to use tidy-shaped data imported and manipulated by `gcplyr` with these packages.

- `growthcurver`
- `QurvE`
- `AUDIT` (including `growr` and `mtpview1`)
- `growthrates`
- `drc`
- `opm`
- `grofit`
- `R-Biolog`
- `growthmodels`
- `cellGrowth`
- `grofit`
- `GCAT`
- `CarboLogR`
- `biogrowth`

Additionally, one `R` package doesn't implement fitting-style approaches, but does contain useful functionality for plate-reader data analysis:

- `plater`