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"). 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, or you can just keep it handy to enter it directly through a function later on.

Let's get started by loading `gcplyr`

```
library(gcplyr)
```

Data layouts

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

Time	A1	A2	A3	...	H11	H12
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_*` functions of `gcplyr`.

If your data is block-shaped, you'll use `read_blocks` and you can start in the next section.

If your data is wide-shaped, you'll use `read_wides` and you can skip down to the **Importing wide-shaped data** section.

In the unlikely event your data is already tidy, you'll use `read_tidys` and you can skip down to the **Importing tidy-shaped data** section.

Importing block-shaped data

To import block-shaped data, use the `read_blocks` function. `read_blocks` only requires a list of filenames (or relative file paths) and will return a list of data.frames (with each `data.frame` corresponding to a single block) 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 <- tempfile(
      pattern = paste(as.character(example_widedata$Time), "_", sep = ""),
      fileext = ".csv")
for (i in 1:length(temp_filenames)) {
  temp_filenames[i] <- strrsplit(temp_filenames[i], split = "\\\\")[[1]][
    length(strrsplit(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 = 9),
          rbind(
            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)
}
```

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

```
print_df(read.csv(temp_filenames[10], header = FALSE,
                  colClasses = "character"))
#>       1     2     3     4     5     6     7     8     9    10    11    12
#> A     0 2e-12     0 2e-12 2e-12     0     0 2e-12     0 2e-12 2e-12     0
#> B 2e-12 2e-12     0 2e-12 2e-12 2e-12 2e-12 2e-12     0 2e-12 2e-12 2e-12
#> C 2e-12 4e-12     0 2e-12     0 2e-12 2e-12 4e-12     0 2e-12     0 2e-12
#> D 2e-12 2e-12 4e-12 2e-12 2e-12 2e-12 2e-12 2e-12 4e-12 2e-12 2e-12     0
#> E 4e-12 2e-12 4e-12     0 2e-12     0 4e-12 2e-12 2e-12     0 2e-12     0
#> F     0 2e-12 2e-12     0     0     0     0 2e-12 2e-12     0     0     0
#> G 2e-12     0 2e-12 4e-12     0     0 2e-12     0 2e-12 4e-12     0     0
#> H 4e-12 4e-12 4e-12 4e-12     0 2e-12 2e-12 4e-12 4e-12 4e-12     0 2e-12
```

This would correspond to all the reads for a single plate taken at the very first timepoint. We can see that the first row contains column headers, and the first column contains row names. The absorbances look small here because R doesn't know that the first row is a header yet.

If we want to read these files into R, we simply provide `read_blocks` with the vector of file names, and save the result to some R object (here, `imported_blockdata`).

```
imported_blockdata <- read_blocks(files = temp_filenames)
```

Specifying the location of your block-shaped data

However, running `read_blocks` with only the filenames only works if the data in your block-shaped files starts in the first row and column (or has column names in the first row and/or rownames in the first column). If your data starts elsewhere, `read_blocks` needs to know what row/column to start reading on (if your data isn't the last thing in the file, `read_blocks` also needs to know where your data ends).

To show how this works, first let's create some example files where the data doesn't begin in the first row/column. In these example files, the plate reader saved the time that each plate was read in the 2nd row of the file, and started saving the data itself with a header in the 4th row.

Again, **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_filenames2`.

```
#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_filenames2 <-
  tempfile(pattern = paste(as.character(example_widedata$Time), "_2_", sep = ""),
                           fileext = ".csv")
for (i in 1:length(temp_filenames2)) {
  temp_filenames2[i] <- strsplit(temp_filenames2[i], split = "\\\\")[[1]][
    length(strsplit(temp_filenames2[i], split = "\\\\")[[1]])]
}
for (i in 1:length(temp_filenames2)) {
  write.table(
    cbind(
      matrix(c("", "", "", "", "A", "B", "C", "D", "E", "F", "G", "H"),
             nrow = 12),
      rbind(
        rep("", 12),
        matrix(c("Time", example_widedata$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_filenames2[i], quote = FALSE, row.names = FALSE, sep = ",",
    col.names = FALSE)
}
```

Let's take a look at one of the files:

```
print_df(read.csv(temp_filenames2[10], header = FALSE,
                  colClasses = "character"))
#>
#>    Time  8100
#>
#>       1     2     3     4     5     6     7     8     9    10    11    12
#> A     0 2e-12     0 2e-12 2e-12     0     0 2e-12     0 2e-12 2e-12     0
#> B 2e-12 2e-12     0 2e-12 2e-12 2e-12 2e-12 2e-12     0 2e-12 2e-12 2e-12
#> C 2e-12 4e-12     0 2e-12     0 2e-12 2e-12 4e-12     0 2e-12     0 2e-12
#> D 2e-12 2e-12 4e-12 2e-12 2e-12 2e-12 2e-12 2e-12 4e-12 2e-12 2e-12     0
#> E 4e-12 2e-12 4e-12     0 2e-12     0 4e-12 2e-12 2e-12     0 2e-12     0
#> F     0 2e-12 2e-12     0     0     0     0 2e-12 2e-12     0     0     0
#> G 2e-12     0 2e-12 4e-12     0     0 2e-12     0 2e-12 4e-12     0     0
#> H 4e-12 4e-12 4e-12 4e-12     0 2e-12 2e-12 4e-12 4e-12 4e-12     0 2e-12
```

In the above example, the column names are in row 4 and the rownames are in column 1. To specify that to `read_blocks`, we simply do:

```
#Now let's read it with read_blocks
imported_blockdata <- read_blocks(
  files = temp_filenames2,
  startrow = 4, startcol = 1)
```

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_blocks` will translate that to a number for you!

```
#Now let's read it with read_blocks
imported_blockdata <- read_blocks(
  files = temp_filenames2,
  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. `read_blocks` 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_filenames2[10], header = FALSE,
                  colClasses = "character"))
#>
#>    Time  8100
#>
#>       1     2     3     4     5     6     7     8     9    10    11    12
#> A     0 2e-12     0 2e-12 2e-12     0     0 2e-12     0 2e-12 2e-12     0
#> B 2e-12 2e-12     0 2e-12 2e-12 2e-12 2e-12 2e-12     0 2e-12 2e-12 2e-12
#> C 2e-12 4e-12     0 2e-12     0 2e-12 2e-12 4e-12     0 2e-12     0 2e-12
#> D 2e-12 2e-12 4e-12 2e-12 2e-12 2e-12 2e-12 2e-12 4e-12 2e-12 2e-12     0
#> E 4e-12 2e-12 4e-12     0 2e-12     0 4e-12 2e-12 2e-12     0 2e-12     0
#> F     0 2e-12 2e-12     0     0     0     0 2e-12 2e-12     0     0     0
#> G 2e-12     0 2e-12 4e-12     0     0 2e-12     0 2e-12 4e-12     0     0
#> H 4e-12 4e-12 4e-12 4e-12     0 2e-12 2e-12 4e-12 4e-12 4e-12     0 2e-12
```

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 `read_blocks` command:

```
#Reading the blockcurves files with metadata included
imported_blockdata <- read_blocks(
  files = temp_filenames2,
  startrow = 4, startcol = "A",
  metadata = list("time" = c(2, 3)))
```

You can see that 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 <- read_blocks(
  files = temp_filenames2,
  startrow = 4, startcol = "A",
  metadata = list("time" = c(2, "C")))
```

What to do next

Now that you've imported your block-shaped data, you'll need to transform it for later analyses. Skip the next section, **Importing wide-shaped data**, and instead jump 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 put the file name(s) in R, here we've stored the file name in `temp_filename`.

```
#This code just creates a wide-shaped example file
#Don't worry about how it works - when working with real growth
#curves data, this file would be created by the plate reader
temp_filename <- paste(tempfile(), ".csv", sep = "")
temp_filename <- strsplit(temp_filename, split = "\\\\")[[1]][
  length(strsplit(temp_filename, split = "\\\\")[[1]])]
write.csv(example_widedata, file = temp_filename, row.names = FALSE)
```

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

```
print_df(head(read.csv(temp_filename, header = FALSE),
              c(10, 4), row.names = FALSE))
#> Time A1    B1    C1
#>    0  0     0     0
#>  900  0     0     0
#> 1800  0     0     0
#> 2700  0     0     0
#> 3600  0     0     0
#> 4500  0 0.001     0
#> 5400  0 0.001     0
#> 6300  0 0.001     0
#> 7200  0 0.001 0.001
```

This would correspond to all the reads for a single plate taken across all timepoints. For instance, we can see that the first column contains the timepoint information, and each subsequent column corresponds to a well in the plate.

If we want to read these files into R, we simply provide `read_wides` with the file name, and save the result to some R object (here, `imported_widedata`).

```
#Now let's use read_wides to import our wide-shaped data
imported_widedata <- read_wides(files = temp_filename)
```

The resulting `data.frame` looks like this:

```
print_df(head(imported_widedata, c(10, 6)))
#> file26e82721437d    0 0     0     0     0
#> file26e82721437d  900 0     0     0     0
#> file26e82721437d 1800 0     0     0     0
#> file26e82721437d 2700 0     0     0     0
#> file26e82721437d 3600 0     0     0     0
#> file26e82721437d 4500 0 0.001     0 0.001
#> file26e82721437d 5400 0 0.001     0 0.001
#> file26e82721437d 6300 0 0.001     0 0.001
#> file26e82721437d 7200 0 0.001 0.001 0.001
#> file26e82721437d 8100 0 0.001 0.001 0.001
```

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.

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(temp_filename, temp_filename))
```

Specifying the location of your wide-shaped data

However, running `read_wides` with only the filename(s) only works if the data in your wide-shaped files starts in the first row and column (or has column names in the first row and/or rownames in the first column). If your data starts elsewhere, `read_wides` needs to know what row/column to start reading on (if your data isn't the last thing in the file, `read_wides` also needs to know where your data ends).

To show how this works, first let's create an example file where the data doesn't begin in the first row/column. In this example file, the plate reader started saving the data itself with a header in the 5th row.

Again, **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 name in `temp_filename2`.

```
#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_filename2 <- tempfile(fileext = ".csv")
temp_filename2 <- strsplit(temp_filename2, split = "\\\\")[[1]][
  length(strsplit(temp_filename2, split = "\\\\")[[1]])]
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 = temp_filename2,
          row.names = FALSE, col.names = FALSE, sep = ",")
```

Let's take a look at the file:

```
#Let's take a peek at what this file looks like
print_df(head(read.csv(temp_filename2, header = FALSE), c(10, 6)))
#> Experiment name Experiment_1
#>      Start date   2022-10-14
#>
#>
#>            Time           A1 B1 C1 D1    E1
#>               0            0  0  0  0     0
#>             900            0  0  0  0     0
#>            1800            0  0  0  0     0
#>            2700            0  0  0  0     0
#>            3600            0  0  0  0 0.001
```

Thus, we can see the data header is in row 5, and the data begins in row 6. To specify that to `read_wides`, we simply do (note that `header = TRUE` by default):

```
imported_widedata <- read_wides(files = temp_filename2,
                                startrow = 5)
print_df(head(imported_widedata, c(10, 6)))
#> file26e87f84123e    0 0     0     0     0
#> file26e87f84123e  900 0     0     0     0
#> file26e87f84123e 1800 0     0     0     0
#> file26e87f84123e 2700 0     0     0     0
#> file26e87f84123e 3600 0     0     0     0
#> file26e87f84123e 4500 0 0.001     0 0.001
#> file26e87f84123e 5400 0 0.001     0 0.001
#> file26e87f84123e 6300 0 0.001     0 0.001
#> file26e87f84123e 7200 0 0.001 0.001 0.001
#> file26e87f84123e 8100 0 0.001 0.001 0.001
```

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 we do so just to show this letter-style functionality).

```
imported_widedata <- read_wides(files = temp_filename2,
                                startrow = 5, 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, 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 = temp_filename2,
                                startrow = 5,
                                metadata = list("experiment_name" = c(1, 2),
                                                "start_date" = c(2, 2)))
print_df(head(imported_widedata, c(6, 3)))
#> file26e87f84123e Experiment_1 2022-10-14
#> file26e87f84123e Experiment_1 2022-10-14
#> file26e87f84123e Experiment_1 2022-10-14
#> file26e87f84123e Experiment_1 2022-10-14
#> file26e87f84123e Experiment_1 2022-10-14
#> file26e87f84123e Experiment_1 2022-10-14
```

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 = temp_filename2,
                                startrow = 5,
                                metadata = list("experiment_name" = c(1, "B"),
                                                "start_date" = c(2, "B")))
```

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 block-shaped to wide-shaped

If the data you've read into the R environment is block-shaped, you'll need to transform it from block-shaped to wide-shaped, and then wide-shaped to tidy-shaped. For the first step, you'll use `trans_block_to_wide`. All you need to do is provide `trans_block_to_wide` with the R object you saved when you used `read_blocks`.

```
imported_blocks_now_wide <- trans_block_to_wide(imported_blockdata)
#> Warning in trans_block_to_wide(imported_blockdata): Inferring nested_metadata to be
#> TRUE
```

Note that `trans_block_to_wide` automatically detected the metadata that `read_blocks` had pulled from our files, and has stored each piece of metadata as a column in our output file.

```
print(head(imported_blocks_now_wide, c(6, 12)), row.names = FALSE)
#>           block_name time A_1   A_2   A_3   A_4   A_5 A_6   A_7   A_8   A_9  A_10
#>     0_2_26e86a222d2c    0   0     0     0     0     0   0     0     0     0     0
#>   900_2_26e8794369c4  900   0     0     0     0     0   0     0     0     0     0
#>  1800_2_26e81ca649c7 1800   0     0     0     0     0   0     0     0     0     0
#>  2700_2_26e872aa5276 2700   0     0     0     0     0   0     0     0     0     0
#>   3600_2_26e83072650 3600   0 0e+00 0e+00 0e+00     0   0     0     0 0e+00     0
#>  4500_2_26e85cb42899 4500   0 0e+00 0e+00 0e+00 0e+00   0 0e+00 0e+00 0e+00 0e+00
```

Now that your block-shaped data has been transformed to wide-shaped data, you can use `trans_wide_to_tidy` (below) to further transform it into the tidy-shaped data we need for our analyses.

Transforming from wide-shaped to tidy-shaped

If the data you've read into the R environment 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 the R object 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_blocks_now_wide,
  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
#>  0_2_26e86a222d2c    0  A_1            0
#>  0_2_26e86a222d2c    0  A_2            0
#>  0_2_26e86a222d2c    0  A_3            0
#>  0_2_26e86a222d2c    0  A_4            0
#>  0_2_26e86a222d2c    0  A_5            0
#>  0_2_26e86a222d2c    0  A_6            0
```

Including design elements

Often during analysis of growth curve data, we'd like to incorporate information on the experimental design. For example, which bacteria are present in which wells, or which wells have received some treatment. `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_tidydesign`

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 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. **Don't worry how the below code works**, just imagine that you've created this file in Excel.

```
temp_filename <- tempfile(fileext = ".csv")
write.csv(
  file = temp_filename,
  x = matrix(rep(c("Treat 1", "Treat 2"), 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(temp_filename, header = FALSE,
                  colClasses = "character"))
#>         1       2       3       4       5       6       7       8       9      10      11      12
#> A Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
#> B Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
#> C Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
#> D Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
```

```
#> E Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
#> F Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
#> G Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
#> H Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 1 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2 Treat 2
```

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, read_blocks will automatically name it according to the file name.

```
my_design <- import_blockdesigns(files = temp_filename, block_names = "Treatment_numbers")
head(my_design, 20)
#>    Well Treatment_numbers
#> 1   A_1           Treat 1
#> 2   A_2           Treat 1
#> 3   A_3           Treat 1
#> 4   A_4           Treat 1
#> 5   A_5           Treat 1
#> 6   A_6           Treat 1
#> 7   A_7           Treat 2
#> 8   A_8           Treat 2
#> 9   A_9           Treat 2
#> 10 A_10           Treat 2
#> 11 A_11           Treat 2
#> 12 A_12           Treat 2
#> 13  B_1           Treat 1
#> 14  B_2           Treat 1
#> 15  B_3           Treat 1
#> 16  B_4           Treat 1
#> 17  B_5           Treat 1
#> 18  B_6           Treat 1
#> 19  B_7           Treat 2
#> 20  B_8           Treat 2
```

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 grown in 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 file. Again, **don't worry how the below code works**, just imagine that you've created this file in Excel.

```
temp_filename2 <- tempfile(fileext = ".csv")
write.csv(
  file = temp_filename2,
  x = matrix(rep(c("Strain A", "Strain B", "Strain C", "Strain D"), 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(temp_filename2, header = FALSE,
                  colClasses = "character"))
#>          1        2        3        4        5        6        7        8        9       10       11
#> A Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A
#> B Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A Strain A
#> C Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B
#> D Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B Strain B
#> E Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C
#> F Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C Strain C
#> G Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D
#> H Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D Strain D
```

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 import this design using `import_blockdesigns`. Since our two blocks contains 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, read_blocks will automatically name it according to the file name.

```
my_design <-
  import_blockdesigns(files = c(temp_filename, temp_filename2),
                      block_names = c("Treatment_numbers", "Strain_letters"))
head(my_design, 20)
#>    Well Treatment_numbers Strain_letters
#> 1   A_1           Treat 1       Strain A
#> 2   A_2           Treat 1       Strain A
#> 3   A_3           Treat 1       Strain A
#> 4   A_4           Treat 1       Strain A
#> 5   A_5           Treat 1       Strain A
#> 6   A_6           Treat 1       Strain A
#> 7   A_7           Treat 2       Strain A
#> 8   A_8           Treat 2       Strain A
#> 9   A_9           Treat 2       Strain A
#> 10 A_10           Treat 2       Strain A
#> 11 A_11           Treat 2       Strain A
#> 12 A_12           Treat 2       Strain A
#> 13  B_1           Treat 1       Strain A
#> 14  B_2           Treat 1       Strain A
#> 15  B_3           Treat 1       Strain A
#> 16  B_4           Treat 1       Strain A
#> 17  B_5           Treat 1       Strain A
#> 18  B_6           Treat 1       Strain A
#> 19  B_7           Treat 2       Strain A
#> 20  B_8           Treat 2       Strain A
```

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-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.

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 tidy-shaped design elements programmatically

If you don't have your experimental design information saved in a file, you can directly create such a `data.frame` using the `gcplyr` function `make_tidydesign`. `make_tidydesign` uses the spatial location of design elements in a multiwell plate as input arguments, but outputs a tidy-shaped `data.frame` that can be easily merged with your tidy-shaped data.

An example with a single design

Let's start with a simple example demonstrating the basic use of `make_tidydesign` (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

To generate a tidy-shaped design `data.frame` representing this information, we can use `make_tidydesign`:

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12,
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3",
      "Strain 4", "Strain 5", "Strain 6"),
    2:7,
```

```
    2:11,
    "123456",
    FALSE)
)
```

Now, what are each of the things we've specified for our "Bacteria" design component?

Well, make_tidydesign expects five things for each design component: * a vector containing the possible values * a vector containing all the rows these values should be applied to * a vector containing all the columns these values should be applied to * a string of the pattern itself within those rows and columns * a Boolean for whether this pattern should be filled byrow (defaults to TRUE)

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 byrow

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

What does the resulting `data.frame` look like?

```
head(my_design, 20)
#>    Well Bacteria
#> 1    A1     <NA>
#> 2    A2     <NA>
#> 3    A3     <NA>
#> 4    A4     <NA>
#> 5    A5     <NA>
#> 6    A6     <NA>
#> 7    A7     <NA>
#> 8    A8     <NA>
#> 9    A9     <NA>
#> 10  A10     <NA>
#> 11  A11     <NA>
#> 12  A12     <NA>
#> 13   B1     <NA>
#> 14   B2 Strain 1
#> 15   B3 Strain 1
#> 16   B4 Strain 1
#> 17   B5 Strain 1
#> 18   B6 Strain 1
#> 19   B7 Strain 1
#> 20   B8 Strain 1
```

A few notes on the pattern string

The fourth element of every argument passed to `make_tidydesign` is the string 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_tidydesign`, 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 our example above, we 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) and specify to `make_tidydesign` what letter you'd like the indices to start with. By default, the order goes from 1 to 9, then A to Z (uppercase), then a to z (lowercase). For instance, in the previous example, we could have done:

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "A",
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"),
    2:7,
    2:11,
    "ABCDEF",
    FALSE)
)
```

Or we could have done:

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"),
    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 <- make_tidydesign(
  nrows = 8, ncols = 12, pattern_split = ",",
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"),
    2:7,
    2:11,
    "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 names	Column 1	Column 2	Column 3	...	Column 11	Column 12
Row G	Blank	Media #1	Media #2	...	Media #10	Blank
Row H	Blank	Blank	Blank	...	Blank	Blank

We can generate that design by adding an additional argument to our make_tidydesign call.

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(c("Strain 1", "Strain 2", "Strain 3",
                    "Strain 4", "Strain 5", "Strain 6"),
                  2:7,
                  2:11,
                  "abcdef",
                  FALSE),
  Media = list(c("Media 1", "Media 2", "Media 3",
                 "Media 4", "Media 5", "Media 6",
                 "Media 7", "Media 8", "Media 9",
                 "Media 10", "Media 11", "Media 12"),
               2:7,
               2:11,
               "abcdefghij")
  )
head(my_design, 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 Strain 1 Media 1
#> 15   B3 Strain 1 Media 2
#> 16   B4 Strain 1 Media 3
#> 17   B5 Strain 1 Media 4
#> 18   B6 Strain 1 Media 5
#> 19   B7 Strain 1 Media 6
#> 20   B8 Strain 1 Media 7
```

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 <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Media = list(c("Media 1", "Media 2", "Media 3",
```

```
                "Media 4", "Media 5", "Media 6",
                "Media 7", "Media 8", "Media 9",
                "Media 10", "Media 11", "Media 12"),
              2:7,
              2:11,
              "abcde0ghij"),
  Bacteria = list(c("Strain 1", "Strain 2", "Strain 3",
                    "Strain 4", "Strain 5", "Strain 6"),
                  2:7,
                  2:11,
                  "abc0ef",
                  FALSE))
head(my_design, 20)
#>    Well   Media Bacteria
#> 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 Media 1 Strain 1
#> 15   B3 Media 2 Strain 1
#> 16   B4 Media 3 Strain 1
#> 17   B5 Media 4 Strain 1
#> 18   B6 Media 5 Strain 1
#> 19   B7    <NA> Strain 1
#> 20   B8 Media 7 Strain 1
```

Note that `make_tidydesign` is not limited to simple alternating patterns. The pattern string specified can be any pattern, which `make_tidydesign` will replicate sufficient times to cover the entire set of listed wells.

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Media = list(c("Media 1", "Media 2", "Media 3"),
               2:7,
               2:11,
               "aabbbc000abc"),
  Bacteria = list(c("Strain 1", "Strain 2"),
                  2:7,
                  2:11,
                  "abaaabbbab",
                  FALSE))
head(my_design, 20)
#>    Well   Media Bacteria
#> 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 Media 1 Strain 1
#> 15   B3 Media 1 Strain 2
#> 16   B4 Media 2 Strain 1
#> 17   B5 Media 2 Strain 1
#> 18   B6 Media 2 Strain 1
#> 19   B7 Media 3 Strain 1
#> 20   B8    <NA> Strain 2
```

`gcplyr` also includes an optional helper function for `make_tidydesign` 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 <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Media = make_designpattern(
    values = c("Media 1", "Media 2", "Media 3",
               "Media 4", "Media 5", "Media 6",
               "Media 7", "Media 8", "Media 9",
               "Media 10", "Media 11", "Media 12"),
    rows = 2:7, cols = 2:11, pattern = "abcde0ghij"),
  Bacteria = make_designpattern(
    values = c("Strain 1", "Strain 2", "Strain 3",
               "Strain 4", "Strain 5", "Strain 6"),
    rows = 2:7, cols = 2:11, pattern = "abc0ef",
    byrow = FALSE))
head(my_design, 20)
#>    Well   Media Bacteria
#> 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 Media 1 Strain 1
#> 15   B3 Media 2 Strain 1
#> 16   B4 Media 3 Strain 1
```

```
#> 17   B5 Media 4 Strain 1
#> 18   B6 Media 5 Strain 1
#> 19   B7    <NA> Strain 1
#> 20   B8 Media 7 Strain 1
```

Merging spectrophotometric and design data

Once we have both our design and data in the R environment, 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 `read_blocks` 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_tidydesign(
  pattern_split = ",", nrows = 8, ncols = 12,
  "Bacteria_strain" = make_designpattern(
    values = paste("Strain", 1:48),
    rows = 1:8, cols = 1:6,
    pattern = paste(1:48, collapse = ","),
    byrow = TRUE),
  "Bacteria_strain" = make_designpattern(
    values = paste("Strain", 1:48),
    rows = 1:8, cols = 7:12,
    pattern = paste(1:48, collapse = ","),
    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`:

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

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

How to pre-process, processing, and analyze your data

Once you have your spectrophotometric and design data merged, you're ready to move on to the next steps: pre-processing, processing, and analyzing.

There are a number of functions in `gcplyr` that can help pre-process, process, and analyze growth curves data. However, unlike the import and transformation steps we've done so far, different projects may require different analyses, and not all users will have the same analysis steps. The **Pre-processing**, **Processing**, and **Analyzing** sections of this document, therefore, are written to highlight the functions available for analysis in `gcplyr`, rather than prescribing a certain series of analysis steps.

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:

```
library(dplyr)
example_data_and_designs_filtered <-
  filter(example_data_and_designs, Well != "B1")
```

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(example_data_and_designs,
         Well != "B1", Bacteria_strain != "Strain 13")
```

Pre-processing: converting dates & times into numeric with lubridate

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 code works**, since it's just creating the output as if you had saved it from the plate reader.

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

Let's take a look at this data.frame. You'll notice that we've just modified the Time column from the `example_data_and_designs` file from the previous section.

```
head(example_data_and_designs)
#>      Time Well Measurements Bacteria_strain    Phage
#> 1 0:00:00   A1            0        Strain 1 No Phage
#> 2 0:00:00   B1            0        Strain 7 No Phage
#> 3 0:00:00   C1            0       Strain 13 No Phage
#> 4 0:00:00   D1            0       Strain 19 No Phage
#> 5 0:00:00   E1            0       Strain 25 No Phage
#> 6 0:00:00   F1            0       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). What are some of the time strings in the `Time` column?

```
unique(example_data_and_designs$Time)
#>  [1] "0:00:00"  "0:15:00"  "0:30:00"  "0:45:00"  "1:00:00"  "1:15:00"  "1:30:00"
#>  [8] "1:45:00"  "2:00:00"  "2:15:00"  "2:30:00"  "2:45:00"  "3:00:00"  "3:15:00"
#> [15] "3:30:00"  "3:45:00"  "4:00:00"  "4:15:00"  "4:30:00"  "4:45:00"  "5:00:00"
#> [22] "5:15:00"  "5:30:00"  "5:45:00"  "6:00:00"  "6:15:00"  "6:30:00"  "6:45:00"
#> [29] "7:00:00"  "7:15:00"  "7:30:00"  "7:45:00"  "8:00:00"  "8:15:00"  "8:30:00"
#> [36] "8:45:00"  "9:00:00"  "9:15:00"  "9:30:00"  "9:45:00"  "10:00:00" "10:15:00"
#> [43] "10:30:00" "10:45:00" "11:00:00" "11:15:00" "11:30:00" "11:45:00" "12:00:00"
#> [50] "12:15:00" "12:30:00" "12:45:00" "13:00:00" "13:15:00" "13:30:00" "13:45:00"
#> [57] "14:00:00" "14:15:00" "14:30:00" "14:45:00" "15:00:00" "15:15:00" "15:30:00"
#> [64] "15:45:00" "16:00:00" "16:15:00" "16:30:00" "16:45:00" "17:00:00" "17:15:00"
#> [71] "17:30:00" "17:45:00" "18:00:00" "18:15:00" "18:30:00" "18:45:00" "19:00:00"
#> [78] "19:15:00" "19:30:00" "19:45:00" "20:00:00" "20:15:00" "20:30:00" "20:45:00"
#> [85] "21:00:00" "21:15:00" "21:30:00" "21:45:00" "22:00:00" "22:15:00" "22:30:00"
#> [92] "22:45:00" "23:00:00" "23:15:00" "23:30:00" "23:45:00" "24:00:00"
```

Here we can see that we have timepoints every 15 minutes all the way up to 24 hours.

Let's use `lubridate` to convert this text back 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`:

```
library(lubridate)

example_data_and_designs$Time <- hms(example_data_and_designs$Time)

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

Great! `hms` has parsed the text for us. However, `hms`, `hm`, and `ms` produce a class specific to the `lubridate` package called a `period`. Unfortunately, `period`s don't work well with some of our downstream analysis steps, so we need to convert it to a pure numeric value. We can use the `lubridate` function `time_length` to do so. 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.

```
example_data_and_designs$Time <- time_length(example_data_and_designs$Time)

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

unique(example_data_and_designs$Time)
#>  [1]     0   900  1800  2700  3600  4500  5400  6300  7200  8100  9000  9900 10800
#> [14] 11700 12600 13500 14400 15300 16200 17100 18000 18900 19800 20700 21600 22500
#> [27] 23400 24300 25200 26100 27000 27900 28800 29700 30600 31500 32400 33300 34200
#> [40] 35100 36000 36900 37800 38700 39600 40500 41400 42300 43200 44100 45000 45900
#> [53] 46800 47700 48600 49500 50400 51300 52200 53100 54000 54900 55800 56700 57600
#> [66] 58500 59400 60300 61200 62100 63000 63900 64800 65700 66600 67500 68400 69300
#> [79] 70200 71100 72000 72900 73800 74700 75600 76500 77400 78300 79200 80100 81000
#> [92] 81900 82800 83700 84600 85500 86400
```

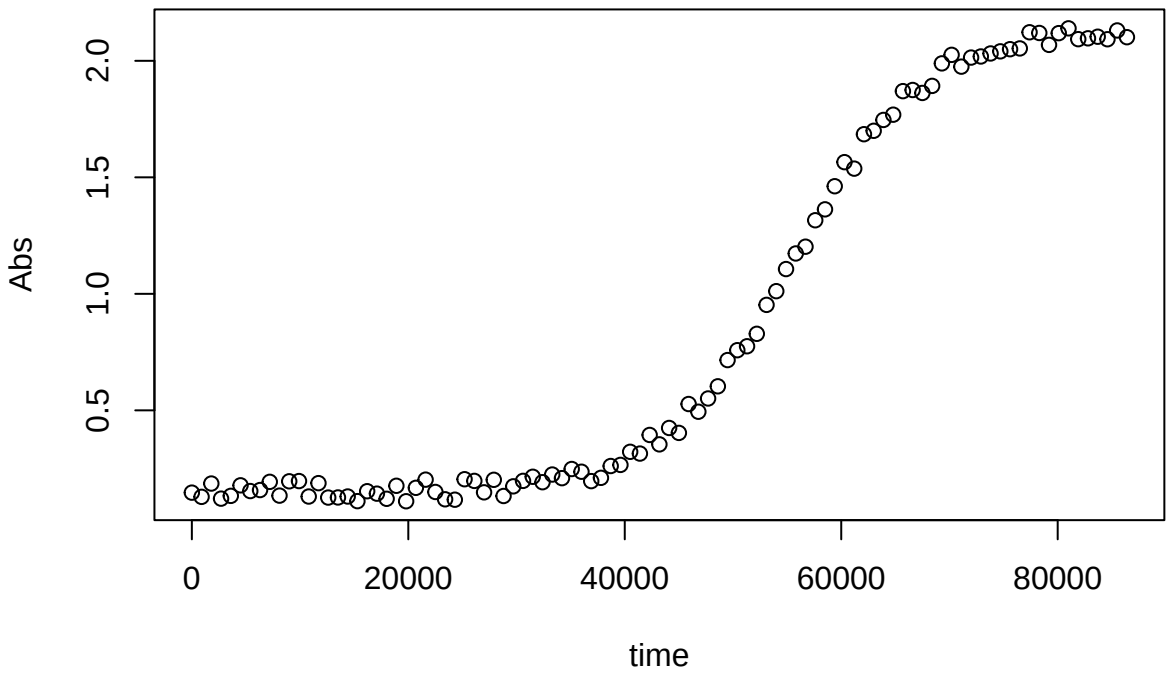
And now we can see that we've gotten our nice numeric `Time` values back! So we can proceed with the next steps of the analysis.

Pre-processing: smoothing

Oftentimes, growth curve data produced by a plate reader will be noisy, and some degree of smoothing before analysis is necessary to reduce this noise and improve the accuracy of analyses. `gcplyr` has a `smooth_data` function that can carry out such smoothing.

First, let's add some noise to the example data we've been working with:

```
#First let's add some simulated noise to our example data
example_data_and_designs$Measurements <-
  example_data_and_designs$Measurements +
  runif(nrow(example_data_and_designs), min = 0.1, max = 0.2)

#What does this noisy data look like?
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$Measurements[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "Abs")
```



Now, we can see how our smoothing works. `smooth_data` has four different smoothing algorithms to choose from: moving average, moving median, loess, and gam. Moving average and moving median are simple smoothing algorithms that primarily act to reduce the effects of outliers on the data. loess and gam are both spline-fitting approaches that smooth data. loess uses polynomial-like curves, which produce curves with smoothly changing derivatives, but can in some cases create curvature artifacts not present in the original data. gam uses additive curves with less smoothly changing derivatives, but tends to better avoid the creation of curvature artifacts.

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.

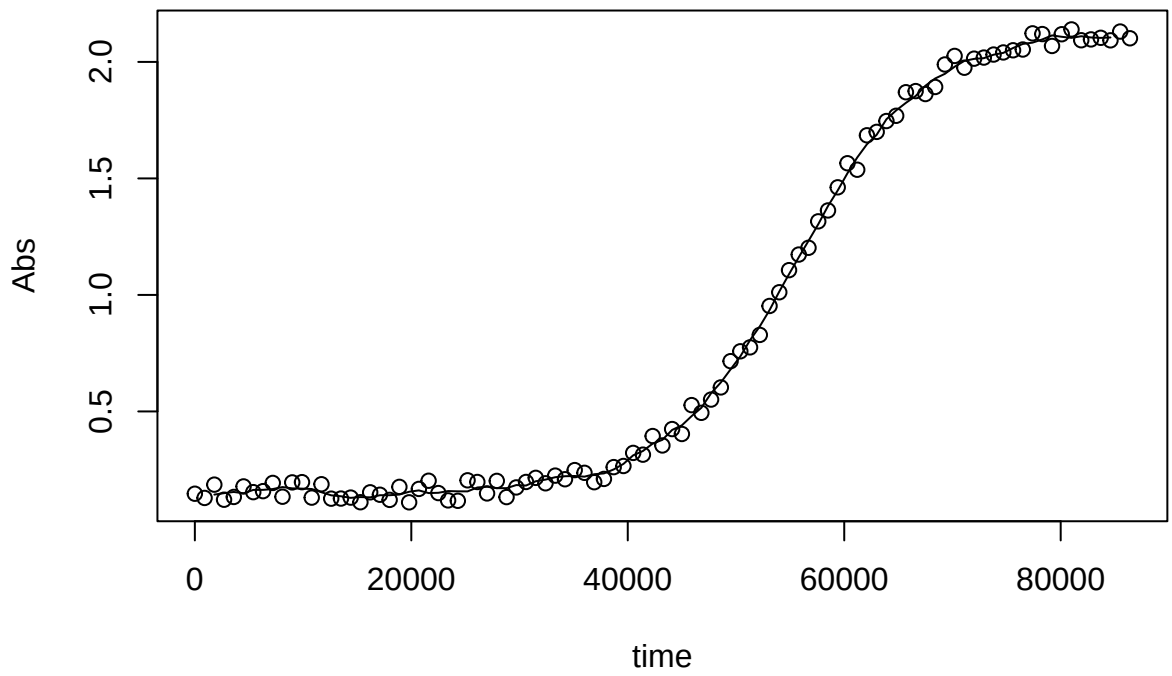
Since your dataframe likely includes data from multiple wells (or even plates), we'll want to only smooth within each of those subsets. You can specify the groupings using the `subset_by` argument, which should be a vector as long as y, whose unique values denote the subset groups. (Note: if you're using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

A note on tuning parameters: All four smoothing algorithms require a tuning parameter that controls how "smoothed" the data are.

- For `moving-average` and `moving-median`, this is the `window_width_n` parameter, which controls how wide the moving windows used to calculate the median and average is.
- For `loess`, this is primarily determined by the `span` argument, which can be passed to `smooth_data` via the `...` argument.
- For `gam`, see `mgcv::gam` for details, where tuning would require passing `formula` and `data` to `smooth_data` via the `...` argument, and altered tuning parameters (e.g. `k`, `sp`, `bs`) would be included in `formula`.

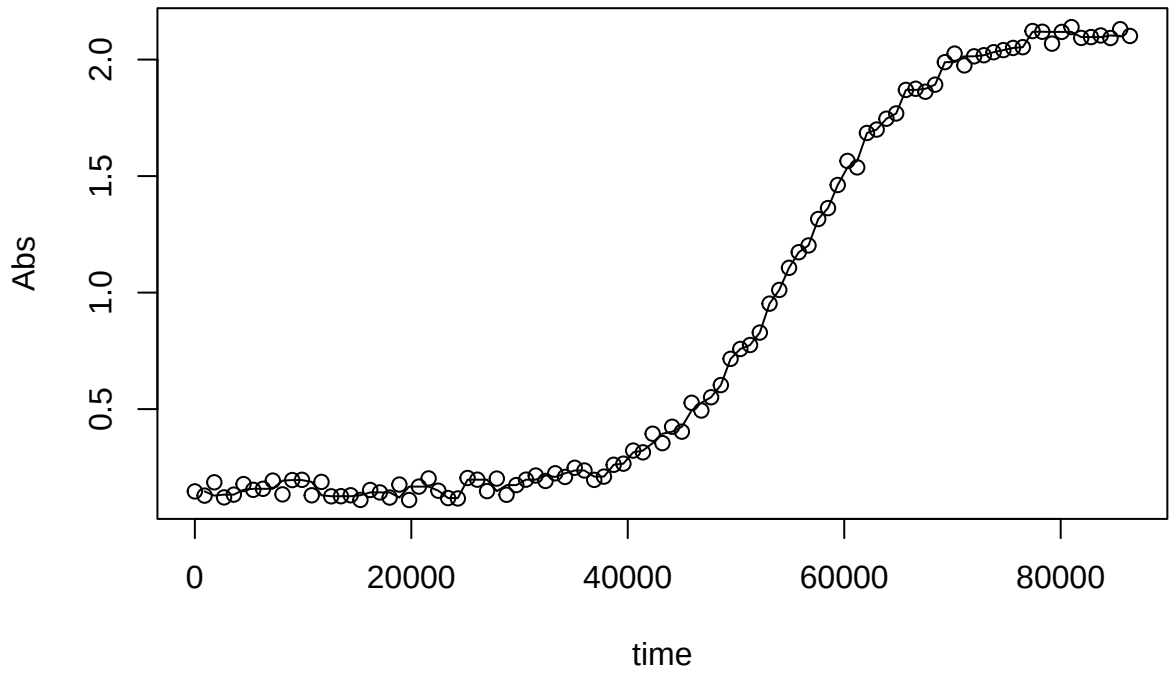
Smoothing with moving-average

```
example_data_and_designs$smoothed <-
  smooth_data(x = example_data_and_designs$Time,
              y = example_data_and_designs$Measurements,
              method = "moving-average",
              subset_by = example_data_and_designs$Well,
              window_width_n = 5)

#What does the smoothed data look like compared to the noisy original?
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$Measurements[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "Abs")
lines(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$smoothed[
       example_data_and_designs$Well == "A2"])
```



Smoothing with moving-median

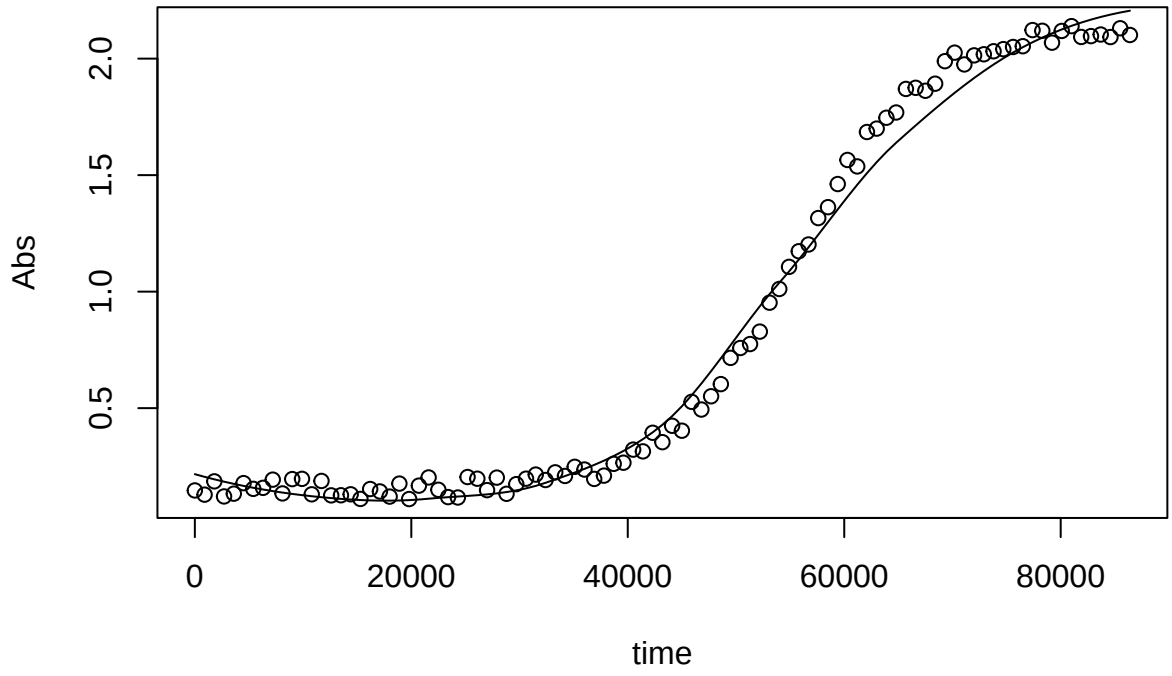
```
example_data_and_designs$smoothed <-
  smooth_data(x = example_data_and_designs$Time,
              y = example_data_and_designs$Measurements,
              method = "moving-median",
              subset_by = example_data_and_designs$Well,
              window_width_n = 3)

#What does the smoothed data look like compared to the noisy original?
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$Measurements[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "Abs")
lines(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$smoothed[
       example_data_and_designs$Well == "A2"])
```



Smoothing with LOESS

```
example_data_and_designs$smoothed <-
  smooth_data(x = example_data_and_designs$Time,
              y = example_data_and_designs$Measurements,
```

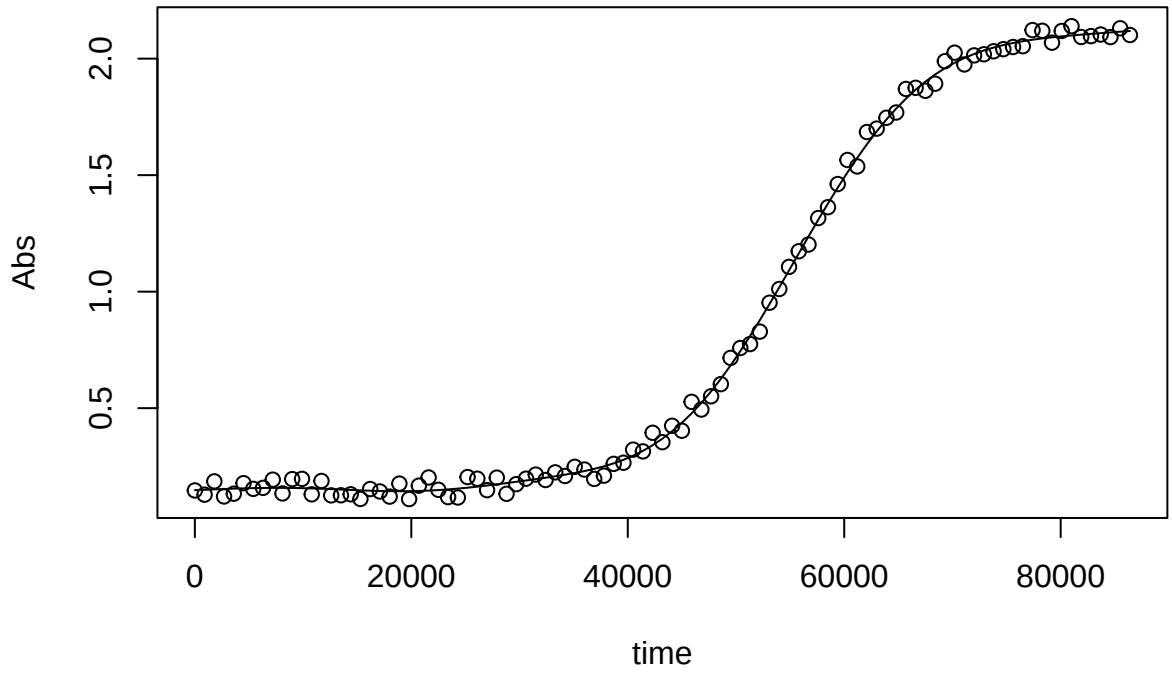
```
              method = "loess",
              subset_by = example_data_and_designs$Well)

#What does the smoothed data look like compared to the noisy original?
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$Measurements[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "Abs")
lines(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$smoothed[
       example_data_and_designs$Well == "A2"])
```



Smoothing with GAM

```
example_data_and_designs$smoothed <-
  smooth_data(x = example_data_and_designs$Time,
              y = example_data_and_designs$Measurements,
              method = "gam",
              subset_by = example_data_and_designs$Well)

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

```
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$Measurements[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "Abs")
lines(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$smoothed[
       example_data_and_designs$Well == "A2"])
```



Processing data: calculating derivatives

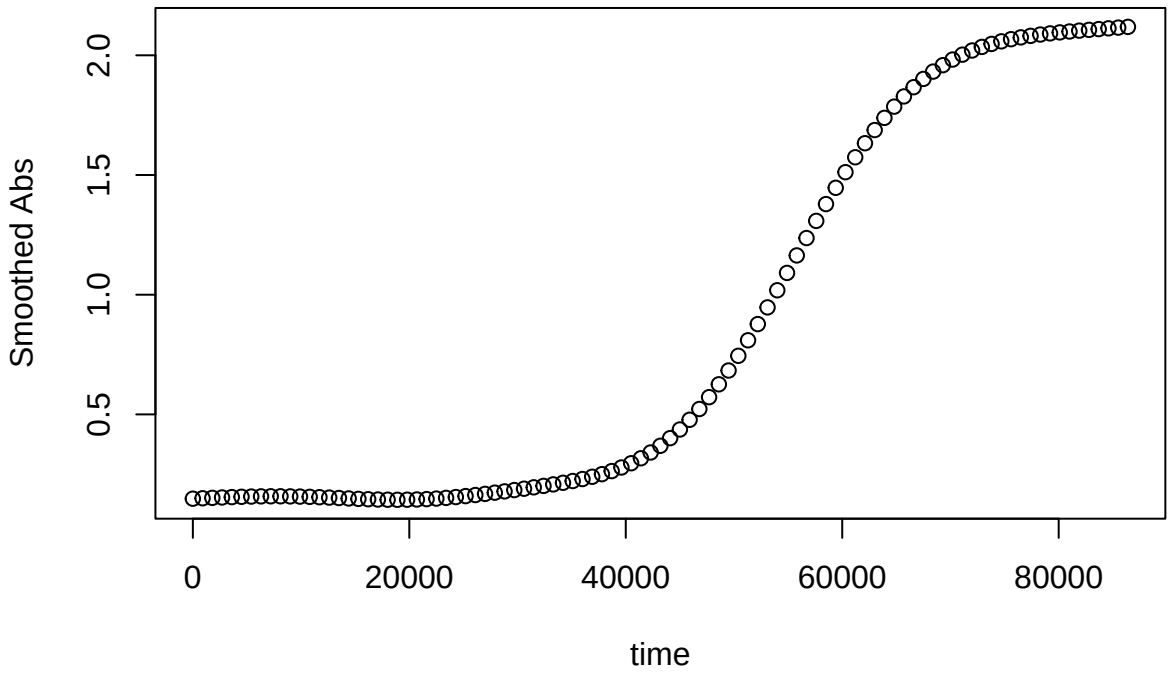
In many cases, identifying features of a growth curve requires looking not only at the absorbance data over time, but the slope of the absorbance data over time. `gcplyr` includes a `calc_deriv` function that can be used to calculate the empirical derivative (slope) of absorbance data over time.

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

Here's the smoothed absorbance data we'll be getting the derivatives of:

```
#Let's plot the smoothed absorbance to remind ourselves what it looks like
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
```

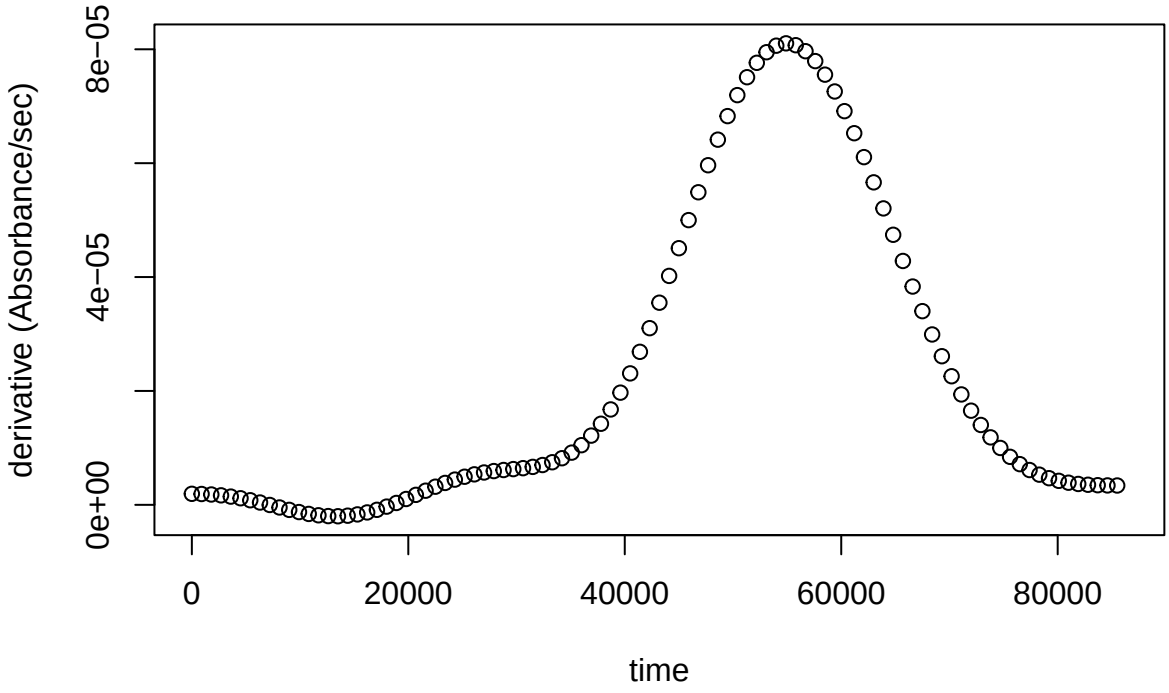
```
    example_data_and_designs$smoothed[
      example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "Smoothed Abs")
```



A simple derivative

To calculate a simple derivative 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)

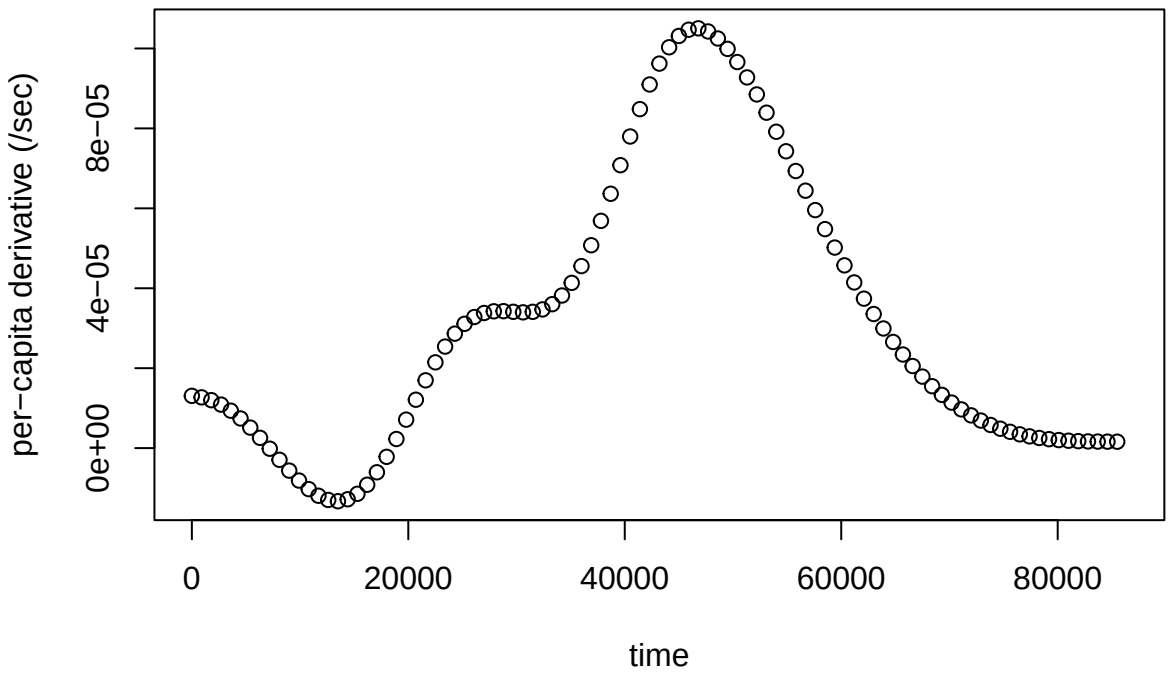
```
example_data_and_designs$deriv <-
  calc_deriv(x = example_data_and_designs$Time,
             y = example_data_and_designs$smoothed,
             subset_by = example_data_and_designs$Well)

#Now let's plot the derivative
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$deriv[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "derivative (Absorbance/sec)")
```



Per-capita derivative

`calc_deriv` can also 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`

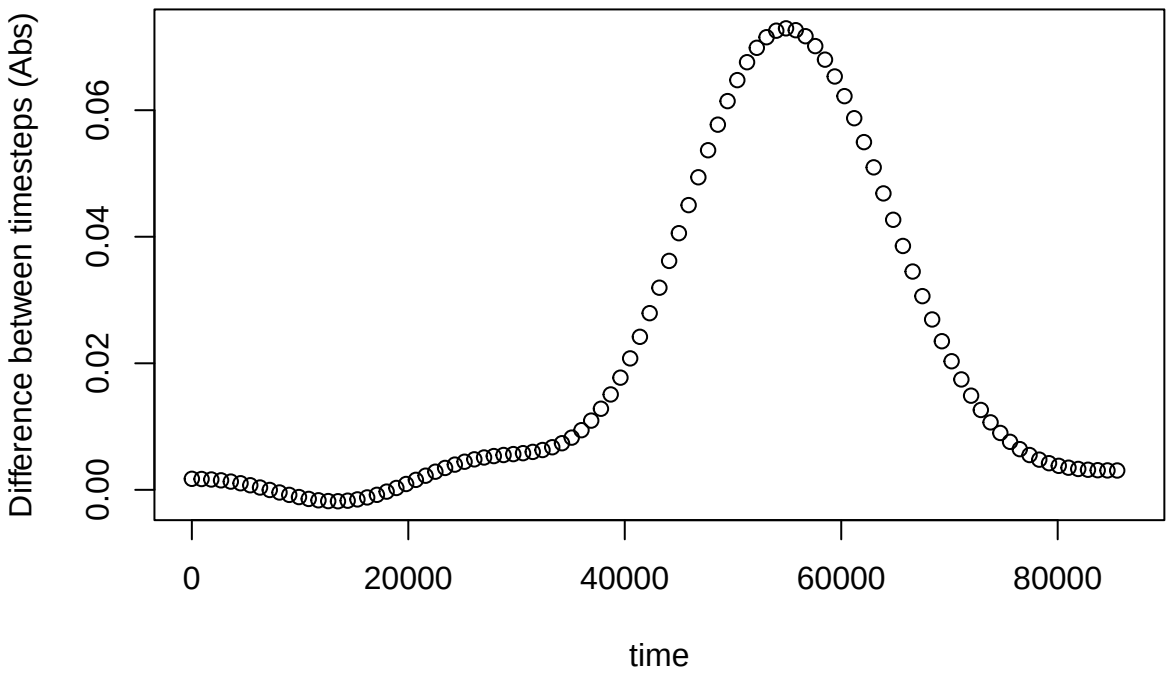
```
example_data_and_designs$deriv_percap <-
  calc_deriv(x = example_data_and_designs$Time,
             y = example_data_and_designs$smoothed,
             subset_by = example_data_and_designs$Well,
             percapita = TRUE)

#Now let's plot the per-capita derivative
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$deriv_percap[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "per-capita derivative (/sec)")
```



Finite differences

If, instead of derivatives, you simply want the difference between each subsequent y value, you can set `scale_x = NA` (in which case, you also don't need to provide the x values). (This looks very similar to our original derivative plot because in the example data all timepoints are equally spaced)

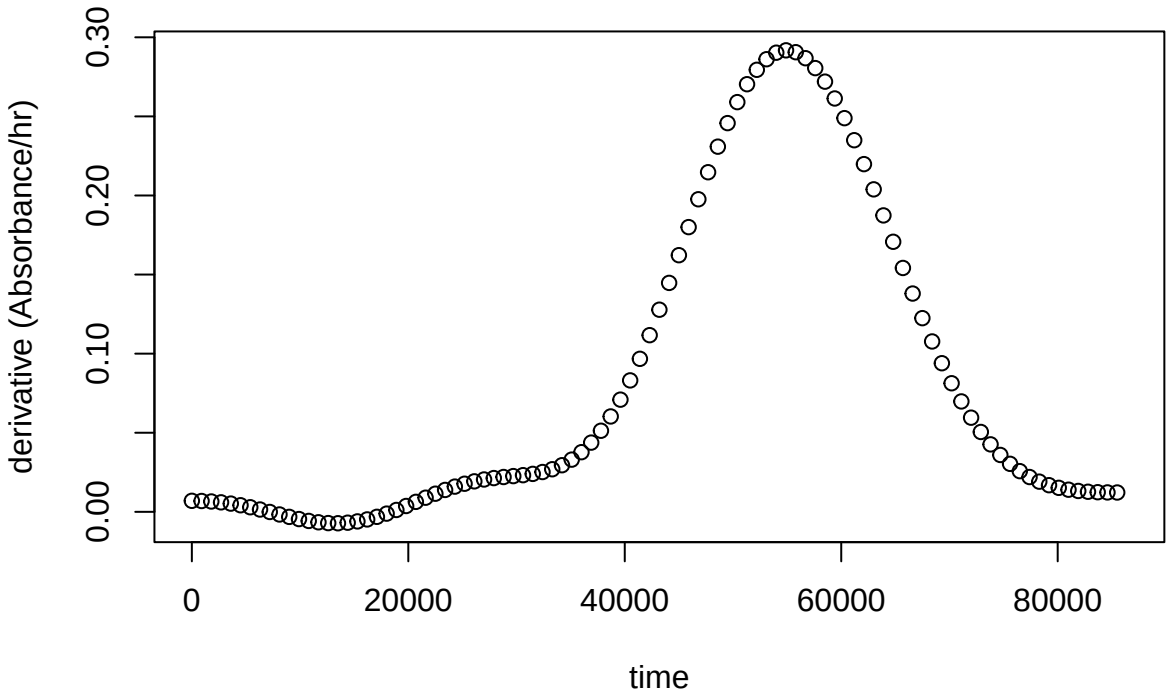
```
example_data_and_designs$difference <-
  calc_deriv(y = example_data_and_designs$smoothed,
             subset_by = example_data_and_designs$Well,
             x_scale = NA)

#Now let's plot the finite differences
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$difference[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "Difference between timesteps (Abs)")
```



Changing the derivative units

Finally, if you want your derivative in units different from those that x is provided in, you can specify the ratio of your x units to the desired units with `x_scale` as well. 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`

```
example_data_and_designs$deriv_hr <-
  calc_deriv(x = example_data_and_designs$Time,
             y = example_data_and_designs$smoothed,
             subset_by = example_data_and_designs$Well,
             x_scale = 3600)

#Now let's plot the derivative in units of Abs/hour
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$deriv_hr[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "derivative (Absorbance/hr)")
```



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 the maximum density, maximum per-capita growth rate, or total area under the curve. `gcplyr` contains a number of functions to assist with these calculations.

However, before we can explore how to use those functions, we need to familiarize ourselves with the `dplyr` 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 next section.** If you're not familiar yet, don't worry! Continue to the next section, where we provide a primer on using `group_by` and `summarize` that will teach you all you need to know for.

A brief primer on dplyr: grouping and summarize

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 `group_by` our data so that every well is a group, and then we'll `summarize` each well with calculations like maximum density or area under the curve.

Let's work through an example. 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.

```
library(dplyr)
grouped_example_data_and_designs <-
  group_by(example_data_and_designs,
           Bacteria_strain, Phage, Well)
```

Since `dplyr` will drop any columns that the data aren't grouped by, we will typically want to list all of our design columns, and 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.

Then, we run `summarize`, specifying the name of the summarized column and the function that calculates the summary output. For instance, in the code below we've calculated the minimum smoothed absorbance each well reached at any point in its growth.

```
example_data_and_designs_sum <-
  summarize(grouped_example_data_and_designs,
            min_abs = min(smoothed))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(example_data_and_designs_sum)
#> # A tibble: 6 x 4
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  min_abs
#>   <chr>           <chr>       <chr>   <dbl>
#> 1 Strain 1        No Phage    A1     0.152
#> 2 Strain 1        Phage Added A7     0.151
#> 3 Strain 10       No Phage    B4     0.124
#> 4 Strain 10       Phage Added B10    0.134
#> 5 Strain 11       No Phage    B5     0.132
#> 6 Strain 11       Phage Added B11    0.0741
```

If you want additional characteristics, you simply add them to the summarize. For instance, we could get the maximum of the per-capita growth rate (note that na.rm is needed to tell `max` to ignore `NA` values):

```
example_data_and_designs_sum <-
  summarize(grouped_example_data_and_designs,
            min_abs = min(smoothed),
            max_percap_deriv = max(deriv_percap, na.rm = TRUE))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(example_data_and_designs_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  min_abs max_percap_deriv
#>   <chr>           <chr>       <chr>   <dbl>            <dbl>
#> 1 Strain 1        No Phage    A1     0.152         0.0000594
#> 2 Strain 1        Phage Added A7     0.151         0.0000607
#> 3 Strain 10       No Phage    B4     0.124         0.000134
#> 4 Strain 10       Phage Added B10    0.134         0.000129
#> 5 Strain 11       No Phage    B5     0.132         0.000151
#> 6 Strain 11       Phage Added B11    0.0741        0.000172
```

That's all you need to know for now! 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 should be sufficient to use the remaining `gcplyr` functions, which have to be used *within* `summarize` to work correctly.

Finding local extrema

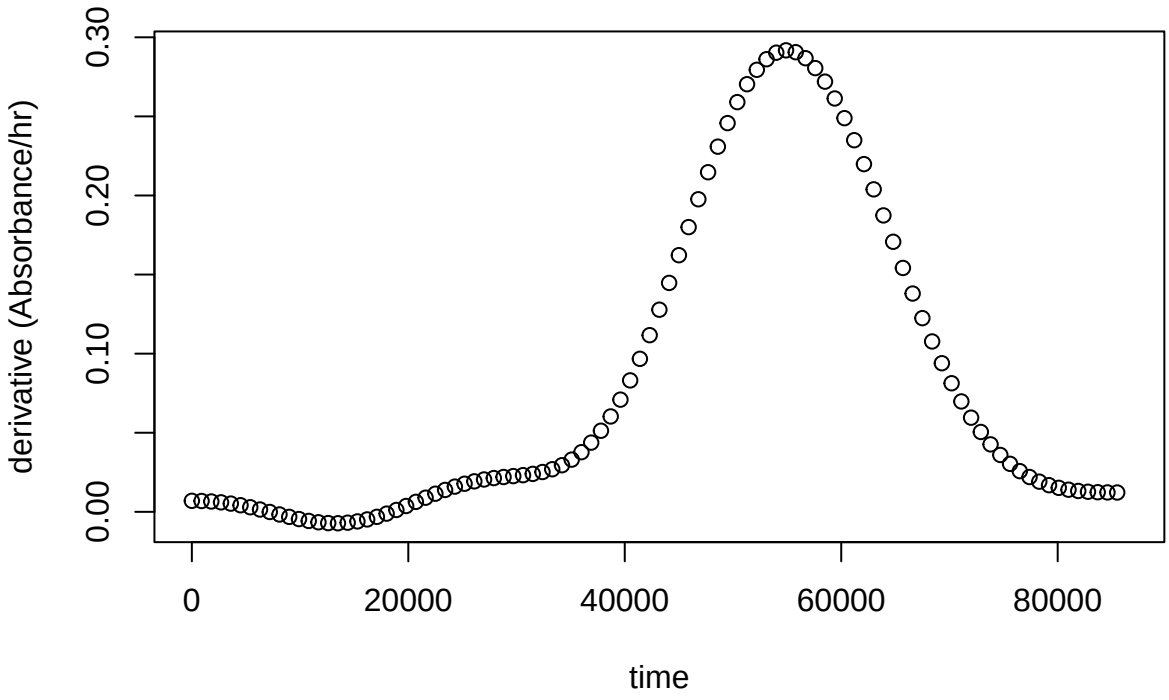
One common analysis step is identifying peaks and valleys in growth curve data, whether it be in the original absorbance data, or in one of the derivatives in the curve. `gcplyr` has several functions to facilitate identifying these local extrema.

A common use-case: the first peak

One of the main peaks or valleys users are interested in identifying is the first peak. For instance, in absorbance data, the first peak could be the maximum absorbance reached before the population begins to decline as a result of phages or antibiotics. Whereas in derivative data, the first peak could show the maximum growth rate of the bacteria.

To identify the first peak, use `first_peak`. `first_peak` simply requires the y data you want to identify the peak in. Let's use the derivative we calculated in the previous section, since it has a clear peak we might want to identify.

```
#Let's plot the derivative in units of Abs/hour again
plot(example_data_and_designs$Time[
  example_data_and_designs$Well == "A2"],
     example_data_and_designs$deriv_hr[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "derivative (Absorbance/hr)")
```



Now let's identify the peak in our data. First, we'll group our data using `dplyr::group_by`, then use `first_peak` inside our summarize command. (Remember to load dplyr with `library(dplyr)` if you haven't already)

```
example_data_and_designs_grouped <-
  group_by(example_data_and_designs,
           Bacteria_strain, Phage, Well)
example_data_and_designs_sum <-
  summarize(example_data_and_designs_grouped,
            first_peak_index = first_peak(deriv_hr))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(example_data_and_designs_sum)
#> # A tibble: 6 x 4
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  first_peak_index
#>   <chr>           <chr>       <chr>            <dbl>
#> 1 Strain 1        No Phage    A1                   1
#> 2 Strain 1        Phage Added A7                  15
#> 3 Strain 10       No Phage    B4                   1
#> 4 Strain 10       Phage Added B10                  1
#> 5 Strain 11       No Phage    B5                  46
#> 6 Strain 11       Phage Added B11                  1
```

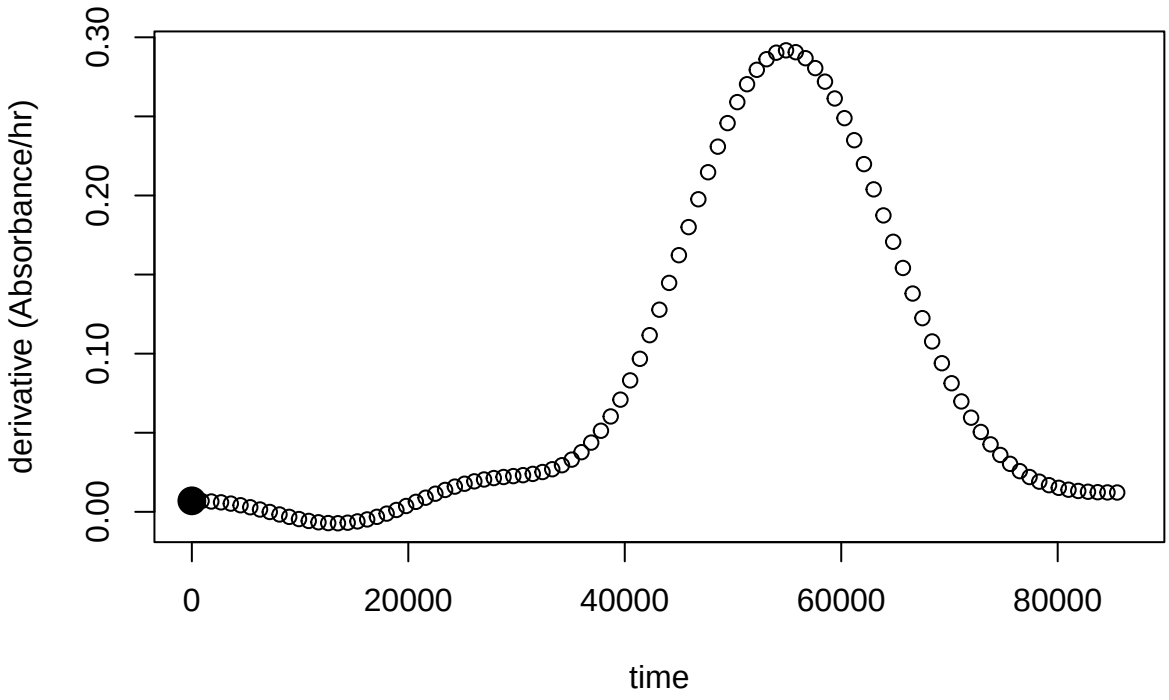
By default, `first_peak` returns the index of the timepoint where the first peak is located *within* the group. If you want the x or y of the first peak, simply set `return = "x"` or `return = "y"`. Note that if `return =`

"x", you must specify the x values to `first_peak`

```
example_data_and_designs_sum <-
  summarize(example_data_and_designs_grouped,
            first_peak_x = first_peak(deriv_hr, x = Time, return = "x"),
            first_peak_y = first_peak(deriv_hr, return = "y"))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(example_data_and_designs_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  first_peak_x first_peak_y
#>   <chr>           <chr>       <chr>        <dbl>        <dbl>
#> 1 Strain 1        No Phage    A1               0      0.00322
#> 2 Strain 1        Phage Added A7           12600      0.00386
#> 3 Strain 10       No Phage    B4               0      0.0184
#> 4 Strain 10       Phage Added B10              0      0.0229
#> 5 Strain 11       No Phage    B5           40500      0.382
#> 6 Strain 11       Phage Added B11              0      0.0607
```

And now that we have x and y values, we can plot them to confirm that `first_peak` finds what we expect.

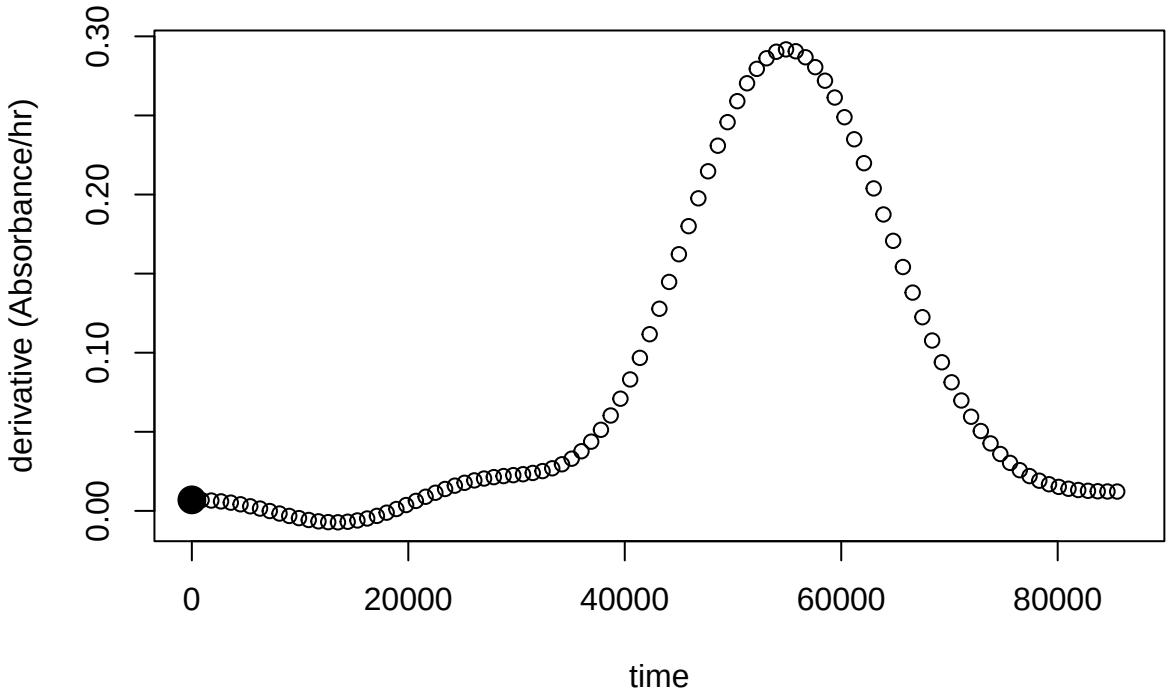
```
plot(example_data_and_designs$Time[
       example_data_and_designs$Well == "A2"],
     example_data_and_designs$deriv_hr[
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "derivative (Absorbance/hr)")
points(x = example_data_and_designs_sum$first_peak_x[
             example_data_and_designs_sum$Well == "A2"],
  y = example_data_and_designs_sum$first_peak_y[
        example_data_and_designs_sum$Well == "A2"],
  pch = 16, cex = 2)
```



Here we can see that `first_peak` has found *a* peak, but perhaps not the large one we're primarily interested in. If we want `first_peak` to be less sensitive to local peaks, we can increase the `width_limit_n` argument (which defaults to 20% of the length of `y`, in this case `= 19`).

```
example_data_and_designs_sum <-
  summarize(example_data_and_designs_grouped,
            first_peak_x = first_peak(deriv_hr, x = Time, return = "x",
                                      width_limit_n = 39),
            first_peak_y = first_peak(deriv_hr, return = "y",
                                      width_limit_n = 39))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(example_data_and_designs_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  first_peak_x first_peak_y
#>   <chr>           <chr>       <chr>        <dbl>        <dbl>
#> 1 Strain 1        No Phage    A1               0      0.00322
#> 2 Strain 1        Phage Added A7           12600      0.00386
#> 3 Strain 10       No Phage    B4               0      0.0184
#> 4 Strain 10       Phage Added B10              0      0.0229
#> 5 Strain 11       No Phage    B5           40500      0.382
#> 6 Strain 11       Phage Added B11              0      0.0607
plot(example_data_and_designs$Time[
       example_data_and_designs$Well == "A2"],
     example_data_and_designs$deriv_hr[
```

```
       example_data_and_designs$Well == "A2"],
  xlab = "time", ylab = "derivative (Absorbance/hr)")
points(x = example_data_and_designs_sum$first_peak_x[
            example_data_and_designs_sum$Well == "A2"],
  y = example_data_and_designs_sum$first_peak_y[
        example_data_and_designs_sum$Well == "A2"],
  pch = 16, cex = 2)
```



In the next section, we'll learn how to use `find_local_extrema` to identify other kinds of local extrema, not just the first peak.

Finding any kind of local extrema

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

In order to identify these kinds of extrema, we can use the more-general function `find_local_extrema`. `find_local_extrema` works very similarly to `first_peak`, but with a few additional options that users can specify to get exactly the kinds of peaks and valleys they want.

Just like `first_peak`, `find_local_extrema` only requires a vector of y data in which to find the local extrema, and will return the index of the extrema *within* the current group. By altering the `return` argument to `return = "x"` or `return = "y"`, find_local_extrema will return x and y values rather than indices.

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 should use brackets to specify which one they want.

For instance, here's an example where we've used `find_local_extrema` to identify the first peak in the data *that includes endpoints*:

```
example_data_and_designs_sum <-
  summarize(example_data_and_designs_grouped,
            first_peak_x = find_local_extrema(
              y = deriv_hr, x = Time, return = "x",
              return_maxima = TRUE, return_minima = FALSE,
              return_endpoints = TRUE, width_limit_n = 39)[1],
            first_peak_y = find_local_extrema(
              y = deriv_hr, return = "y",
              return_maxima = TRUE, return_minima = FALSE,
              return_endpoints = TRUE, width_limit_n = 39)[1])
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(example_data_and_designs_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well  first_peak_x first_peak_y
#>   <chr>           <chr>       <chr>        <dbl>        <dbl>
#> 1 Strain 1        No Phage    A1               0      0.00322
#> 2 Strain 1        Phage Added A7           12600      0.00386
#> 3 Strain 10       No Phage    B4               0      0.0184
#> 4 Strain 10       Phage Added B10              0      0.0229
#> 5 Strain 11       No Phage    B5           40500      0.382
#> 6 Strain 11       Phage Added B11              0      0.0607
```

Additionally, note that with `find_local_extrema`, users must specify at least one of the tuning parameters: `width_limit_n` or `height_limit`. These parameters control how sensitive the function is to smaller local peaks and valleys. `width_limit_n` is the number of data points wide the algorithm will search at each step, meaning that a smaller `width_limit_n` will be more sensitive to narrow peaks and valleys. `height_limit` (in units of `y`) limits the depth of the peaks and valleys the algorithm will search over at each step, meaning that a smaller `height_limit` will be more sensitive to shallow peaks and valleys.

Threshold identification

[This section to-be-written]

Area under the curve

One other common metric of growth curves is the total area under the curve. `gcplyr` has an `auc` function to easily calculate this area. Just like `first_peak` and `find_local_extrema`, it needs to be used inside of a `data.frame` that has been grouped and is being summarized using `dplyr`.

To use `auc`, simply specify the x and y data you are interested in calculating the area-under-the-curve of. Note that you can also specify a subset of the x-range to calculate the area of, in cases where you do not want the area under the curve from the beginning to the end of your time series.

Here, we calculate the area-under-the-curve of the density data, as well as the area-under-the-curve beginning after 3 hours (10800 seconds)

```
example_data_and_designs_sum <-
  summarize(example_data_and_designs_grouped,
            auc = auc(x = Time, y = smoothed),
            auc_after3hrs = auc(x = Time, y = smoothed, xlim = c(10800, NA)))
#> `summarise()` has grouped output by 'Bacteria_strain', 'Phage'. You can override
#> using the `.groups` argument.
head(example_data_and_designs_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage       Well      auc auc_after3hrs
#>   <chr>           <chr>       <chr>   <dbl>         <dbl>
#> 1 Strain 1        No Phage    A1     22183.        20502.
#> 2 Strain 1        Phage Added A7     21316.        19671.
#> 3 Strain 10       No Phage    B4     96121.        94563.
#> 4 Strain 10       Phage Added B10    95597.        93918.
#> 5 Strain 11       No Phage    B5    104527.       102989.
#> 6 Strain 11       Phage Added B11    40390.        38701.
```

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 `example_data_and_designs_sum` from the previous section, where we've summarized our growth curves using area-under-the-curve (although this approach would work with any number of summarized metrics). 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.

```
example_data_and_designs_sum <-
  dplyr::filter(example_data_and_designs_sum, Phage == "No Phage")
head(example_data_and_designs_sum)
#> # A tibble: 6 x 5
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage    Well      auc auc_after3hrs
#>   <chr>           <chr>    <chr>   <dbl>         <dbl>
#> 1 Strain 1        No Phage A1     22183.        20502.
#> 2 Strain 10       No Phage B4     96121.        94563.
#> 3 Strain 11       No Phage B5    104527.       102989.
#> 4 Strain 12       No Phage B6     47355.        45891.
#> 5 Strain 13       No Phage C1     46037.        44407.
#> 6 Strain 14       No Phage C2    116988.       115399.
```

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.

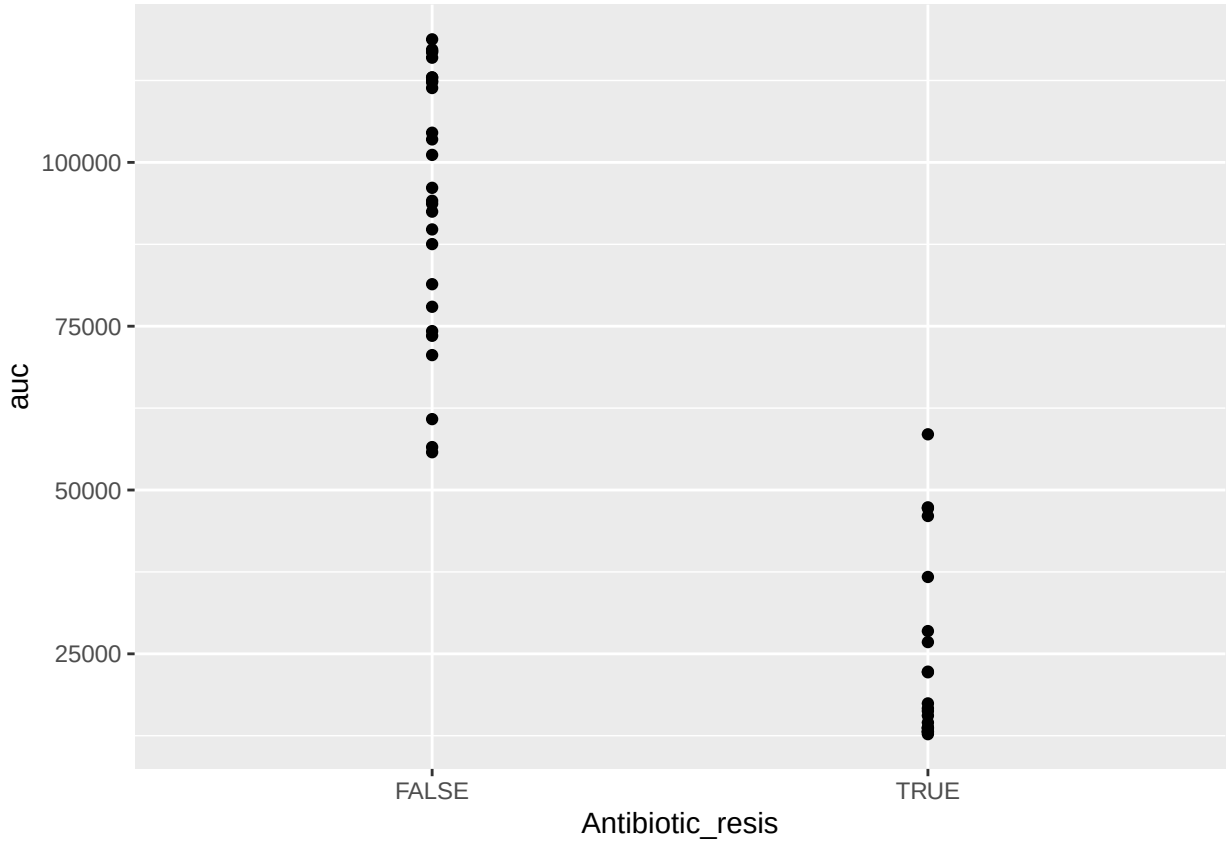
```
set.seed(123)
antibiotic_dat <-
  data.frame(
    Bacteria_strain = paste("Strain", 1:48),
    Antibiotic_resis =
      example_data_and_designs_sum$auc[
        match(paste("Strain", 1:48),
              example_data_and_designs_sum$Bacteria_strain)] *
      runif(48, 0.5, 1.5) < mean(example_data_and_designs_sum$auc))

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(example_data_and_designs_sum,
                                    antibiotic_dat)
#> Joining, by = "Bacteria_strain"
head(growth_and_antibiotics)
#> # A tibble: 6 x 6
#> # Groups:   Bacteria_strain, Phage [6]
#>   Bacteria_strain Phage    Well      auc auc_after3hrs Antibiotic_resis
#>   <chr>           <chr>    <chr>   <dbl>         <dbl> <lgl>
#> 1 Strain 1        No Phage A1     22183.        20502. TRUE
#> 2 Strain 10       No Phage B4     96121.        94563. FALSE
#> 3 Strain 11       No Phage B5    104527.       102989. FALSE
#> 4 Strain 12       No Phage B6     47355.        45891. TRUE
#> 5 Strain 13       No Phage C1     46037.        44407. TRUE
#> 6 Strain 14       No Phage C2    116988.       115399. FALSE
```

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

```
library(ggplot2)
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, we did simulate the data so we'd get that result).