gcplyr-workflow

Mike Blazanin

Table of 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 |
| 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 can simply read it using the built-in R function read.table.

## 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.

### The simplest 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] <- strsplit(temp\_filenames[i], split = "\\\\")[[1]][  
 length(strsplit(temp\_filenames[i], split = "\\\\")[[1]])]  
}  
for (i in 1:length(temp\_filenames)) {  
 write.table(  
 cbind(matrix(c("", "A", "B", "C", "D", "E", "F", "G", "H"), nrow = 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.

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")

Additionally, some plate readers might output growth curve data in a block shape but in a single file. For instance, the file may contain the block from lines 1 - 8, then an empty line, then the next block from lines 10 - 17, etc. Since read\_blocks is vectorized on most of its input arguments, including startrow, startcol, endrow, and endcol, such a layout can be specified by passing a vector of startrows and endrows to read\_blocks:

imported\_blockdata <- read\_blocks(  
 files = "example\_file.csv",  
 startrow = c(1, 10, 19, 28, 37, 46, 55),  
 endrow = c(8, 17, 26, 35, 44, 53, 62))

### 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).

### The simplest 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.

#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)))  
#> file3ad023da1525 0 0 0 0 0  
#> file3ad023da1525 900 0 0 0 0  
#> file3ad023da1525 1800 0 0 0 0  
#> file3ad023da1525 2700 0 0 0 0  
#> file3ad023da1525 3600 0 0 0 0  
#> file3ad023da1525 4500 0 0.001 0 0.001  
#> file3ad023da1525 5400 0 0.001 0 0.001  
#> file3ad023da1525 6300 0 0.001 0 0.001  
#> file3ad023da1525 7200 0 0.001 0.001 0.001  
#> file3ad023da1525 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 2022-03-22   
#>   
#>   
#> 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)))  
#> file3ad033db7741 0 0 0 0 0  
#> file3ad033db7741 900 0 0 0 0  
#> file3ad033db7741 1800 0 0 0 0  
#> file3ad033db7741 2700 0 0 0 0  
#> file3ad033db7741 3600 0 0 0 0  
#> file3ad033db7741 4500 0 0.001 0 0.001  
#> file3ad033db7741 5400 0 0.001 0 0.001  
#> file3ad033db7741 6300 0 0.001 0 0.001  
#> file3ad033db7741 7200 0 0.001 0.001 0.001  
#> file3ad033db7741 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)))  
#> file3ad033db7741 Experiment\_1 2022-03-22  
#> file3ad033db7741 Experiment\_1 2022-03-22  
#> file3ad033db7741 Experiment\_1 2022-03-22  
#> file3ad033db7741 Experiment\_1 2022-03-22  
#> file3ad033db7741 Experiment\_1 2022-03-22  
#> file3ad033db7741 Experiment\_1 2022-03-22

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.

# 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.frames are the best shape for analyses and required by gcplyr.

You can transform your data.frames 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\_3ad03711877 0 0 0 0 0 0 0 0 0 0 0  
#> 900\_2\_3ad0c5319a0 900 0 0 0 0 0 0 0 0 0 0  
#> 1800\_2\_3ad0aab74c8 1800 0 0 0 0 0 0 0 0 0 0  
#> 2700\_2\_3ad0953686e 2700 0 0 0 0 0 0 0 0 0 0  
#> 3600\_2\_3ad041e26eca 3600 0 0 0 0 0 0 0 0 0 0  
#> 4500\_2\_3ad047466a53 4500 0 0 0 0 0 0 0 0 0 0

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\_3ad03711877 0 A\_1 0  
#> 0\_2\_3ad03711877 0 A\_2 0  
#> 0\_2\_3ad03711877 0 A\_3 0  
#> 0\_2\_3ad03711877 0 A\_4 0  
#> 0\_2\_3ad03711877 0 A\_5 0  
#> 0\_2\_3ad03711877 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 tidy-shaped files using read\_table functions and merged with previously-imported data 2. Design elements can be generated programmatically using make\_tidydesign

## Reading design elements from files

Just like spectrophotometric data, design elements that are saved in tidy-shaped tabular data files can be read using the read\_table function.

Once these design elements have been read into the R environment, you can merge them with your data. See the next section for details.

## 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 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"))

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

# Analyzing data

Once you have your spectrophotometric and design data merged, you’re ready to begin analyzing your data.

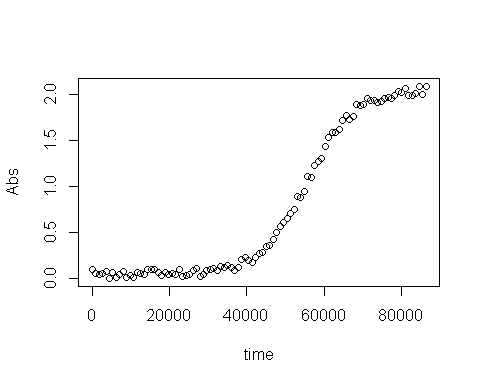
There are a number of functions in gcplyr that can help analyze growth curves data. However, unlike the import and transformation steps we’ve done so far, different projects may require different analyses, and not all users will have the same analysis steps. The **Analyzing data** section of this document,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

## Pre-processing with 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, max = 0.1)  
  
#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.

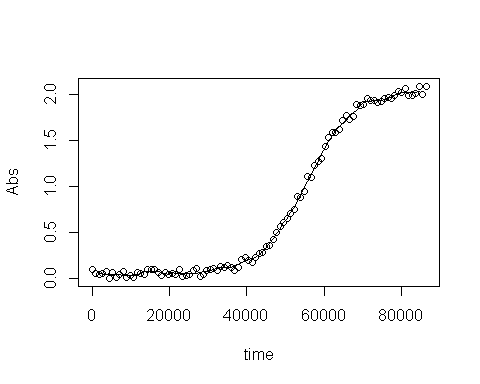
To use smooth\_data, pass your x and y values, your method of choice, and any additional arguments needed for the method. It will return a vector of your smoothed y values.

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 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.

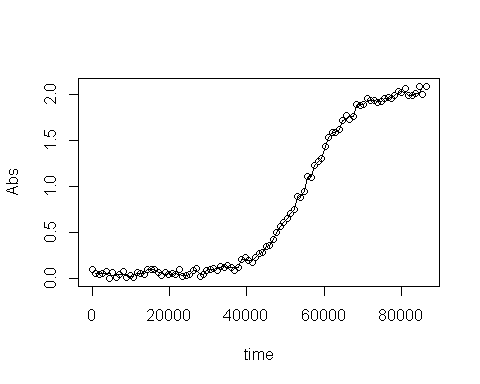
### Smoothing with moving-average

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 = 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"])



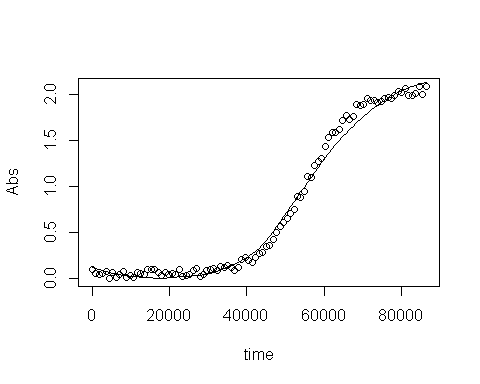
### Smoothing with moving-median

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 = 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"])



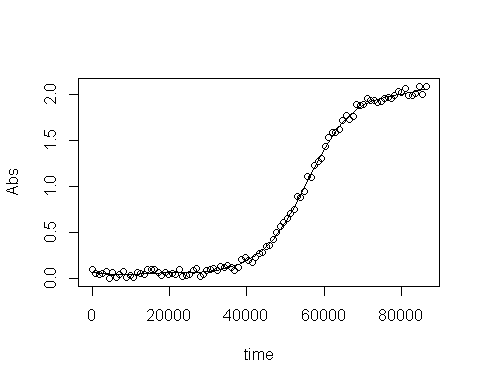
### Smoothing with LOESS

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"])



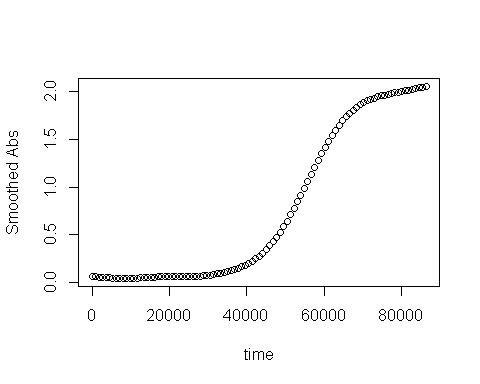
## Calculating derivatives

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 original absorbance data curve we’ll be getting the derivatives of:**

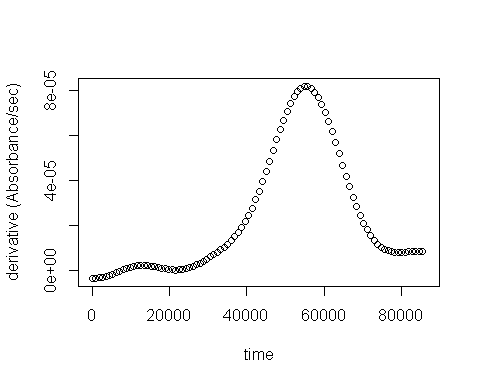
#Now let's plot the absorbance to remind ourselves what it looks like  
plot(example\_data\_and\_designs$Time[  
 example\_data\_and\_designs$Well == "A2"],  
 example\_data\_and\_designs$smoothed[  
 example\_data\_and\_designs$Well == "A2"],  
 xlab = "time", ylab = "Smoothed Abs")



### A simple derivative

To calculate a simple derivative, 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)

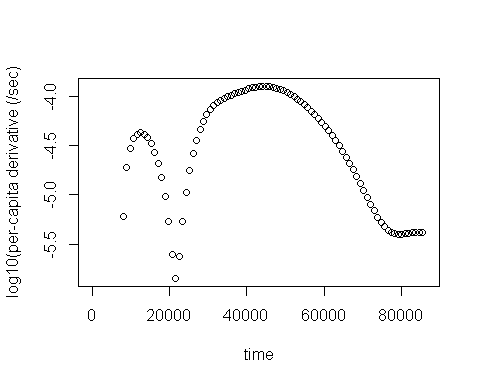
example\_data\_and\_designs$deriv <-  
 calc\_deriv(x = example\_data\_and\_designs$Time,  
 y = example\_data\_and\_designs$smoothed,  
 subset\_by = example\_data\_and\_designs$Well)  
  
#Now let's plot the derivative  
plot(example\_data\_and\_designs$Time[  
 example\_data\_and\_designs$Well == "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, simply by setting percapita = TRUE

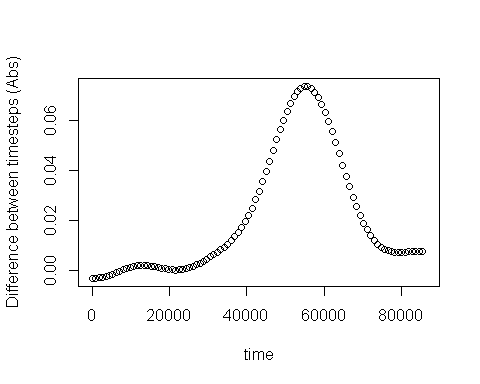
example\_data\_and\_designs$deriv\_percap <-  
 calc\_deriv(x = example\_data\_and\_designs$Time,  
 y = example\_data\_and\_designs$smoothed,  
 subset\_by = example\_data\_and\_designs$Well,  
 percapita = TRUE)  
  
#Now let's plot the per-capita derivative  
plot(example\_data\_and\_designs$Time[  
 example\_data\_and\_designs$Well == "A2"],  
 log10(example\_data\_and\_designs$deriv\_percap[  
 example\_data\_and\_designs$Well == "A2"]),  
 xlab = "time", ylab = "log10(per-capita derivative (/sec)")  
#> Warning in xy.coords(x, y, xlabel, ylabel, log): NaNs produced



### Finite differences

If, instead of derivatives, you simply want the difference between each time-step, 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)

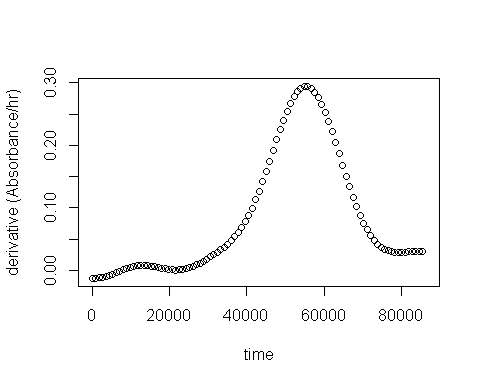
example\_data\_and\_designs$difference <-  
 calc\_deriv(y = example\_data\_and\_designs$smoothed,  
 subset\_by = example\_data\_and\_designs$Well,  
 x\_scale = NA)  
  
#Now let's plot the finite differences  
plot(example\_data\_and\_designs$Time[  
 example\_data\_and\_designs$Well == "A2"],  
 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)")



## Finding local extrema

### Introducing the multi-purpose function: find\_local\_extrema

### A common special-case: the first peak

## Threshold identification

## Area under the curve

# Handling multiple plates simultaneously

[further documentation to-be-written]