Temperature modeling using nested dataframes

Prepare the data

http://ijlyttle.github.io/isugg_purrr/presentation.html#(1)

Packages to run this presentation

```r
library("readr")
library("tibble")
library("dplyr")
library("tidyr")
library("stringr")
library("ggplot2")
library("purrr")
library("broom")
```

Motivation

As you know, purrr is a recent package from Hadley Wickham, focused on lists and functional programming, like dplyr is focused on data-frames.

I figure a good way to learn a new package is to try to solve a problem, so we have a dataset:

- you can view or download
- you can download the source of this presentation
- these are three temperatures recorded simultaneously in a piece of electronics
- it will be very valuable to be able to characterize the transient temperature for each sensor
- we want to apply the same set of models across all three sensors
- it will be easier to show using pictures

Let's get the data into shape

Using the readr package

```r
temperature_wide <-
  read_csv("temperature.csv") %>%
  print()
```

```
# A tibble: 327 x 4
   instant             temperature_a temperature_b temperature_c
   <dttm>                      <dbl>         <dbl>         <dbl>
 1 2015-11-13 06:10:19          116.          91.7          84.2
 2 2015-11-13 06:10:23          116.          91.7          84.2
 3 2015-11-13 06:10:27          116.          91.6          84.2
 4 2015-11-13 06:10:31          116.          91.7          84.2
 5 2015-11-13 06:10:36          116.          91.7          84.2
```

```
 6 2015-11-13 06:10:41          116.          91.6          84.2
 7 2015-11-13 06:10:46          116.          91.5          84.2
 8 2015-11-13 06:10:51          116.          91.5          84.2
 9 2015-11-13 06:10:56          116.          91.5          84.2
10 2015-11-13 06:11:01          115.          91.5          84.2
# ... with 317 more rows
```

Is `temperature_wide` "tidy"?

```
# A tibble: 327 x 4
   instant             temperature_a temperature_b temperature_c
   <dttm>                      <dbl>         <dbl>         <dbl>
 1 2015-11-13 06:10:19          116.          91.7          84.2
 2 2015-11-13 06:10:23          116.          91.7          84.2
 3 2015-11-13 06:10:27          116.          91.6          84.2
 4 2015-11-13 06:10:31          116.          91.7          84.2
 5 2015-11-13 06:10:36          116.          91.7          84.2
 6 2015-11-13 06:10:41          116.          91.6          84.2
 7 2015-11-13 06:10:46          116.          91.5          84.2
 8 2015-11-13 06:10:51          116.          91.5          84.2
 9 2015-11-13 06:10:56          116.          91.5          84.2
10 2015-11-13 06:11:01          115.          91.5          84.2
# ... with 317 more rows
```

Why or why not?

Tidy data

1. Each column is a variable
2. Each row is an observation
3. Each cell is a value

(http://www.jstatsoft.org/v59/i10/paper)

My personal observation is that "tidy" can depend on the context, on what you want to do with the data.

Let's get this into a tidy form

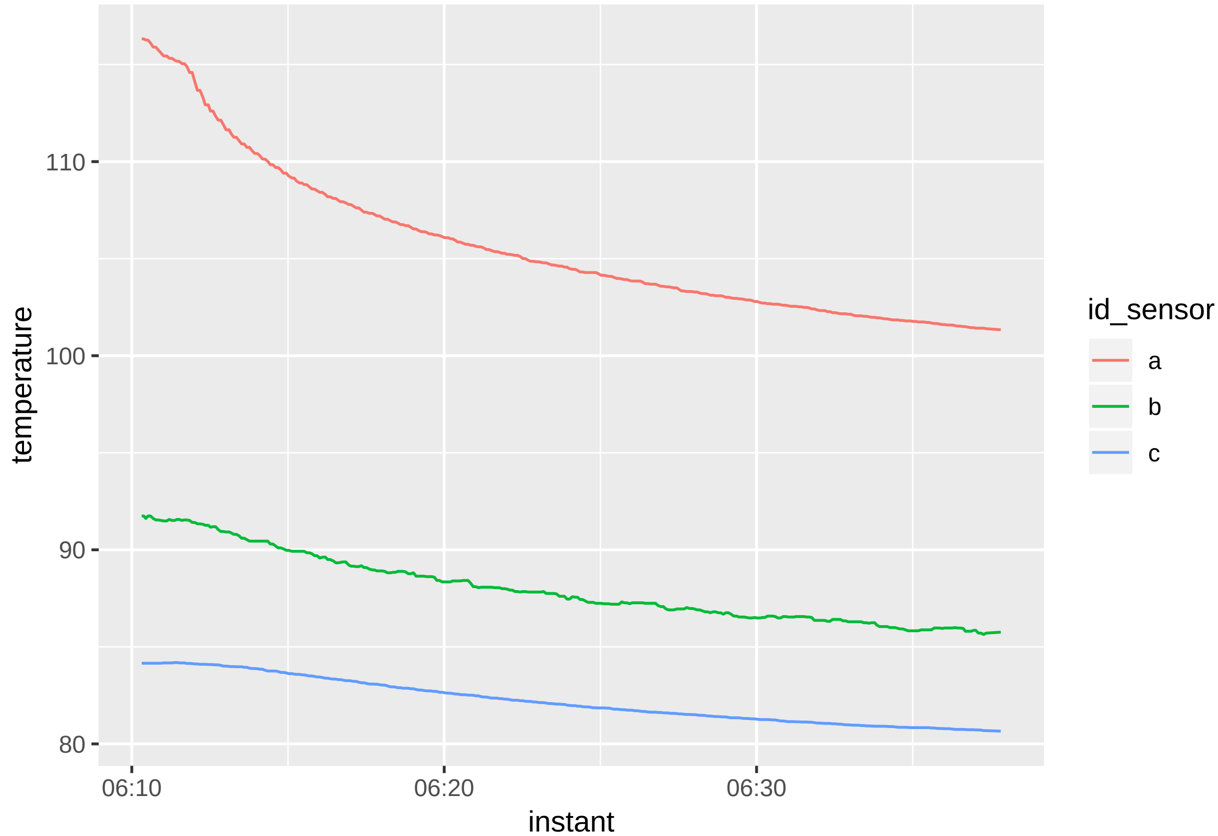
```
temperature_tall <-
  temperature_wide %>%
  gather(key = "id_sensor", value = "temperature", starts_with("temp")) %>%
  mutate(id_sensor = str_replace(id_sensor, "temperature_", "")) %>%
  print()
```

```
# A tibble: 981 x 3
   instant             id_sensor temperature
   <dttm>              <chr>           <dbl>
 1 2015-11-13 06:10:19 a                116.
 2 2015-11-13 06:10:23 a                116.
 3 2015-11-13 06:10:27 a                116.
 4 2015-11-13 06:10:31 a                116.
 5 2015-11-13 06:10:36 a                116.
 6 2015-11-13 06:10:41 a                116.
```

```
 7 2015-11-13 06:10:46 a                116.
 8 2015-11-13 06:10:51 a                116.
 9 2015-11-13 06:10:56 a                116.
10 2015-11-13 06:11:01 a                115.
# ... with 971 more rows
```

Now, it's easier to visualize

```
temperature_tall %>%
  ggplot(aes(x = instant, y = temperature, color = id_sensor)) +
  geom_line()
```



Calculate delta time (Δt) and delta temperature (ΔT)

`delta_time` Δt

change in time since event started, s

`delta_temperature`: ΔT

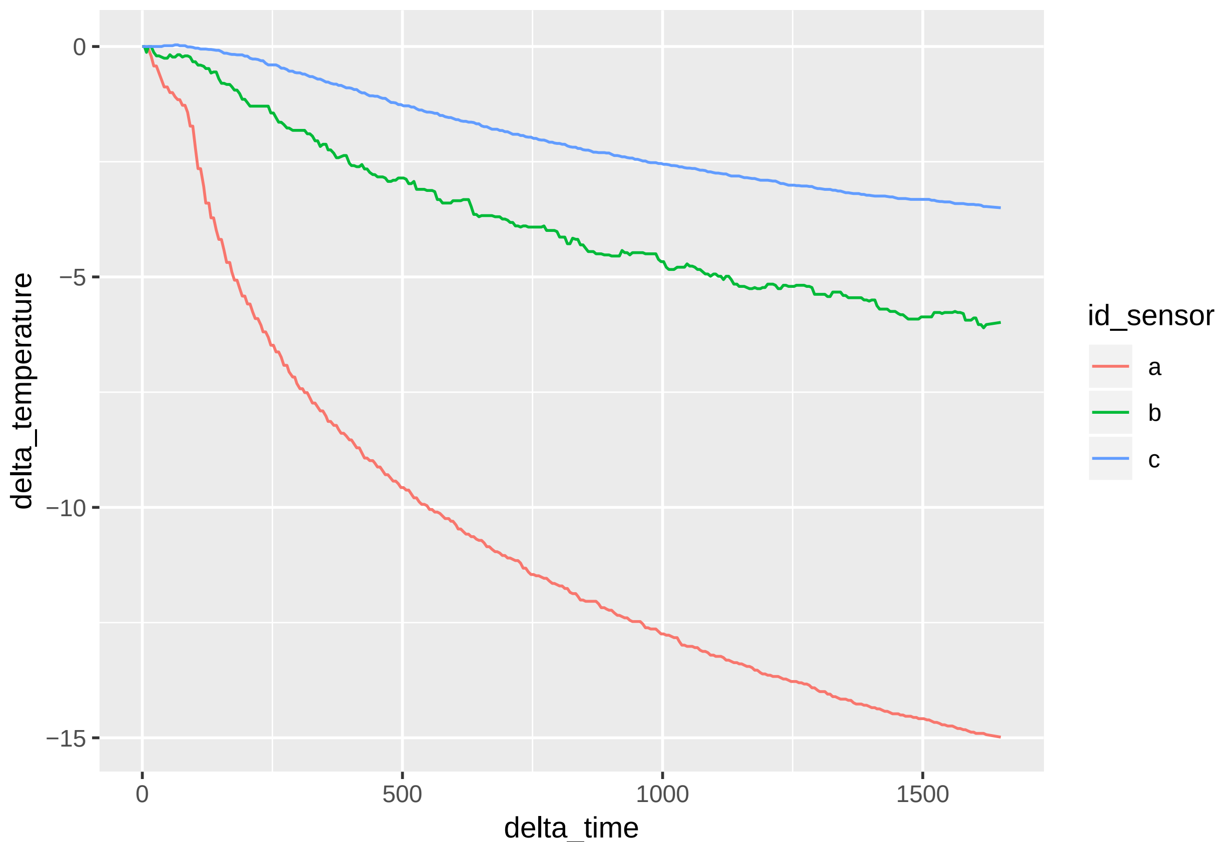
change in temperature since event started, °C

```
delta <-
  temperature_tall %>%
  arrange(id_sensor, instant) %>%
  group_by(id_sensor) %>%
```

```r
  mutate(
    delta_time = as.numeric(instant) - as.numeric(instant[[1]]),
    delta_temperature = temperature - temperature[[1]]
  ) %>%
  select(id_sensor, delta_time, delta_temperature)
```

Let's have a look

```r
# plot delta time vs delta temperature, by sensor
delta %>%
  ggplot(aes(x = delta_time, y = delta_temperature, color = id_sensor)) +
  geom_line()
```



Define the models

We want to see how three different curve-fits might perform on these three data-sets:

Newtonian cooling

$$\Delta T = \Delta T_0 * (1 - e^{-\frac{\delta t}{\tau_0}})$$

Semi-infinite solid

$$\Delta T = \Delta T_0 * erfc(\sqrt{\frac{\tau_0}{\delta t}}))$$

Semi-infinite solid with convection

$$\Delta T = \Delta T_0 * \left[erfc(\sqrt{\frac{\tau_0}{\delta t}}) - e^{Bi_0 + (\frac{Bi_0}{2})^2 \frac{\delta t}{\tau_0}} * erfc(\sqrt{\frac{\tau_0}{\delta t}} + \frac{Bi_0}{2} * \sqrt{\frac{\delta t}{\tau_0}} \right]$$

`erf` and `erfc` functions

```
# reference: http://stackoverflow.com/questions/29067916/r-error-function-erfz
# (see Abramowitz and Stegun 29.2.29)
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
```

Newton cooling equation

```
newton_cooling <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}
```

Temperature models: simple and convection

```
semi_infinite_simple <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * erfc(sqrt(tau_0 / delta_time)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}

semi_infinite_convection <- function(x){
  nls(
    delta_temperature ~
      delta_temperature_0 * (
        erfc(sqrt(tau_0 / delta_time)) -
        exp(Bi_0 + (Bi_0/2)^2 * delta_time / tau_0) *
          erfc(sqrt(tau_0 / delta_time) +
        (Bi_0/2) * sqrt(delta_time / tau_0))
      ),
    start = list(delta_temperature_0 = -5, tau_0 = 50, Bi_0 = 1.e6),
    data = x
  )
}
```

Test modeling on one dataset

Before going into purrr

Before doing anything, we want to show that we can do something with one dataset and one model-function:

```
# only one sensor; it is a test
tmp_data <- delta %>% filter(id_sensor == "a")

tmp_model <- newton_cooling(tmp_data)

summary(tmp_model)
```

```
Formula: delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0))

Parameters:
                     Estimate Std. Error t value Pr(>|t|)
delta_temperature_0 -15.06085    0.05262  -286.2   <2e-16 ***
tau_0               500.01382    4.83673   103.4   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3267 on 325 degrees of freedom

Number of iterations to convergence: 7
Achieved convergence tolerance: 4.136e-06
```

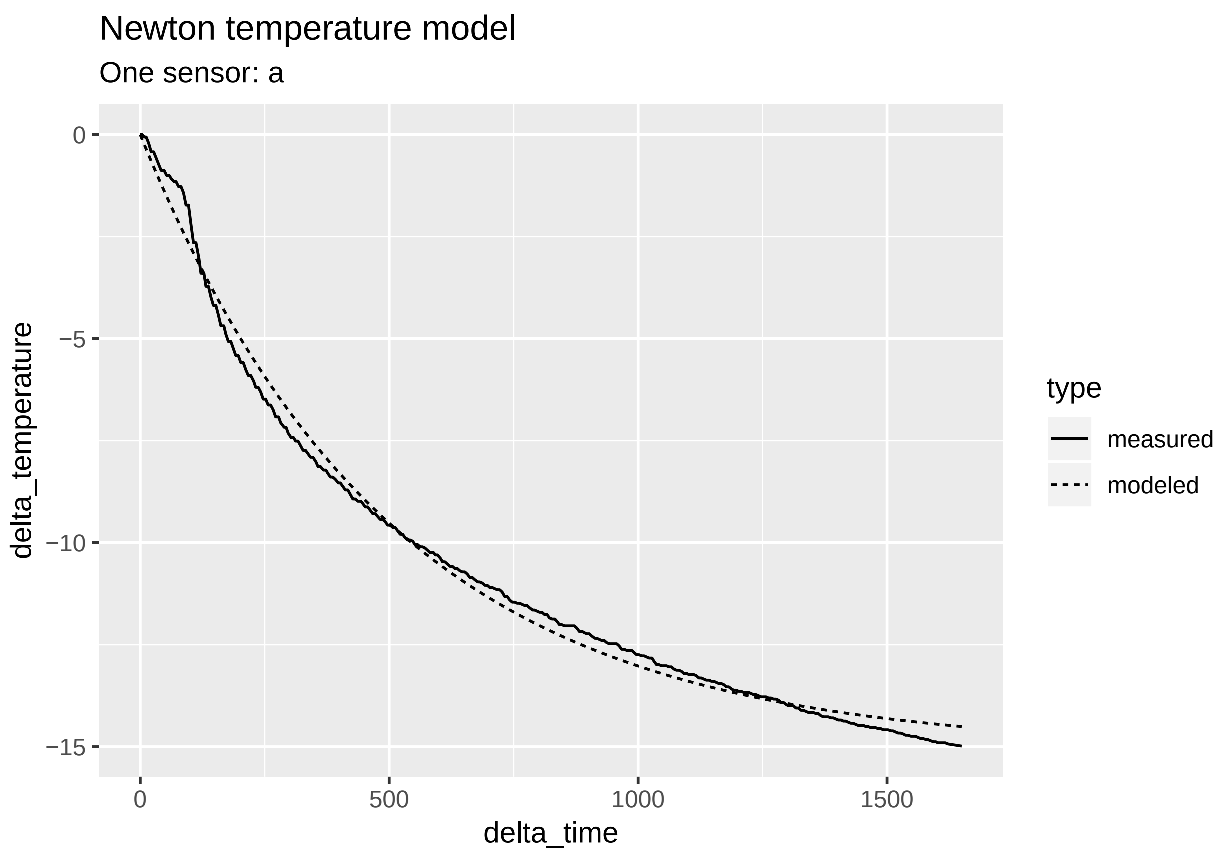
Look at predictions

```
# apply prediction and make it tidy
tmp_pred <-
  tmp_data %>%
  mutate(modeled = predict(tmp_model, data = .)) %>%
  select(id_sensor, delta_time, measured = delta_temperature, modeled) %>%
  gather("type", "delta_temperature", measured:modeled) %>%
  print()
```

```
# A tibble: 654 x 4
# Groups:   id_sensor [1]
   id_sensor delta_time type     delta_temperature
   <chr>          <dbl> <chr>                <dbl>
 1 a                  0 measured             0
 2 a                  4 measured             0
 3 a                  8 measured            -0.06
 4 a                 12 measured            -0.06
 5 a                 17 measured            -0.211
 6 a                 22 measured            -0.423
 7 a                 27 measured            -0.423
 8 a                 32 measured            -0.574
 9 a                 37 measured            -0.726
10 a                 42 measured            -0.878
# ... with 644 more rows
```

Plot Newton model

```r
tmp_pred %>%
  ggplot(aes(x = delta_time, y = delta_temperature, linetype = type)) +
  geom_line() +
  labs(title = "Newton temperature model", subtitle = "One sensor: a")
```



"Regular" data-frame (deltas)

```r
print(delta)
```

```
# A tibble: 981 x 3
# Groups:   id_sensor [3]
   id_sensor delta_time delta_temperature
   <chr>          <dbl>             <dbl>
 1 a                  0             0
 2 a                  4             0
 3 a                  8            -0.06
 4 a                 12            -0.06
 5 a                 17            -0.211
 6 a                 22            -0.423
 7 a                 27            -0.423
 8 a                 32            -0.574
 9 a                 37            -0.726
```

```
10 a                    42              -0.878
# ... with 971 more rows
```

Each column of the dataframe is a vector - in this case, a character vector and two doubles

Making a nested dataframe

How to make a weird data-frame

Here's where the fun starts - a column of a data-frame can be a list.

- use `tidyr::nest()` to makes a column `data`, which is a list of data-frames
- this seems like a stronger expression of the `dplyr::group_by()` idea

```
# nest delta_time and delta_temperature variables
delta_nested <-
  delta %>%
  nest(-id_sensor) %>%
  print()
```

```
# A tibble: 3 x 2
  id_sensor data
  <chr>     <list>
1 a         <tibble [327 x 2]>
2 b         <tibble [327 x 2]>
3 c         <tibble [327 x 2]>
```

Map dataframes to a modeling function (Newton)

- `map()` is like `lapply()`
- `map()` returns a list-column (it keeps the weirdness)

```
model_nested <-
  delta_nested %>%
  mutate(model = map(data, newton_cooling)) %>%
  print()
```

```
# A tibble: 3 x 3
  id_sensor data               model
  <chr>     <list>             <list>
1 a         <tibble [327 x 2]> <S3: nls>
2 b         <tibble [327 x 2]> <S3: nls>
3 c         <tibble [327 x 2]> <S3: nls>
```

We get an additional list-column `model`.

We can use `map2()` to make the predictions

- `map2()` is like `mapply()`
- designed to map two columns (`model`, `data`) to a function `predict()`

```r
predict_nested <-
  model_nested %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
```

```
# A tibble: 3 x 4
  id_sensor data               model     pred
  <chr>     <list>             <list>    <list>
1 a         <tibble [327 x 2]> <S3: nls> <dbl [327]>
2 b         <tibble [327 x 2]> <S3: nls> <dbl [327]>
3 c         <tibble [327 x 2]> <S3: nls> <dbl [327]>
```

Another list-column `pred` for the prediction results.

We need to get out of the weirdness

- use `unnest()` to get back to a regular data-frame

```r
predict_unnested <-
  predict_nested %>%
  unnest(data, pred) %>%
  print()
```

```
# A tibble: 981 x 4
   id_sensor   pred delta_time delta_temperature
   <chr>      <dbl>      <dbl>             <dbl>
 1 a          0              0             0
 2 a         -0.120          4             0
 3 a         -0.239          8            -0.06
 4 a         -0.357         12            -0.06
 5 a         -0.503         17            -0.211
 6 a         -0.648         22            -0.423
 7 a         -0.792         27            -0.423
 8 a         -0.934         32            -0.574
 9 a         -1.07          37            -0.726
10 a         -1.21          42            -0.878
# ... with 971 more rows
```

We can wrangle the predictions

- get into a form that makes it easier to plot

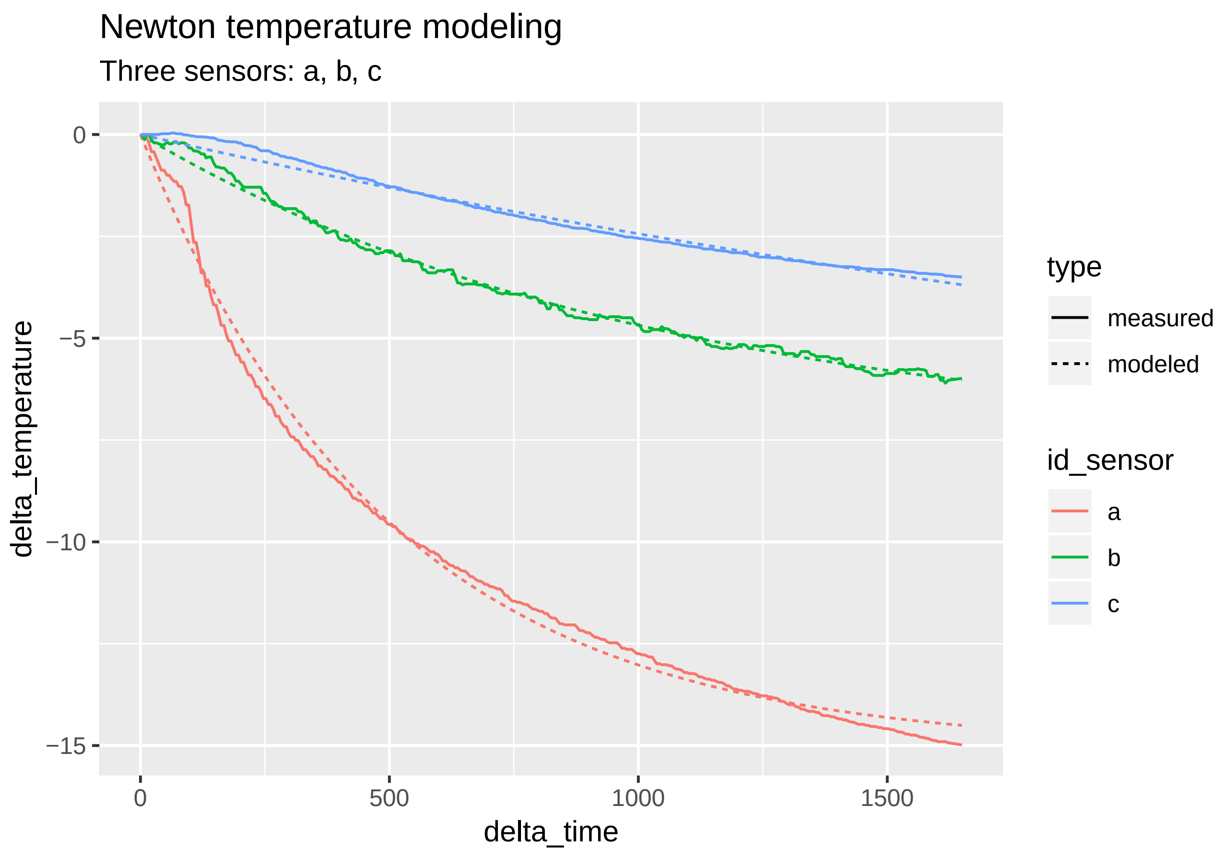
```r
predict_tall <-
  predict_unnested %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
```

```
# A tibble: 1,962 x 4
   id_sensor delta_time type     delta_temperature
   <chr>          <dbl> <chr>                <dbl>
 1 a                  0 modeled              0
 2 a                  4 modeled             -0.120
 3 a                  8 modeled             -0.239
```

```
 4 a                 12 modeled            -0.357
 5 a                 17 modeled            -0.503
 6 a                 22 modeled            -0.648
 7 a                 27 modeled            -0.792
 8 a                 32 modeled            -0.934
 9 a                 37 modeled            -1.07
10 a                 42 modeled            -1.21
# ... with 1,952 more rows
```

We can visualize the predictions

```r
predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  labs(title = "Newton temperature modeling",
       subtitle = "Three sensors: a, b, c")
```



Apply multiple models on a nested structure

Step 1: Selection of models

Make a list of functions to model:

```
list_model <-
  list(
    newton_cooling = newton_cooling,
    semi_infinite_simple = semi_infinite_simple,
    semi_infinite_convection = semi_infinite_convection
  )
```

Step 2: write a function to define the "inner" loop

```
# add additional variable with the model name

fn_model <- function(.model, df) {
  # one parameter for the model in the list, the second for the data
  # safer to avoid non-standard evaluation
  # df %>% mutate(model = map(data, .model))

  df$model <- map(df$data, possibly(.model, NULL))
  df
}
```

- for a given model-function and a given (weird) data-frame, return a modified version of that data-frame with a column `model`, which is the model-function applied to each element of the data-frame's `data` column (which is itself a list of data-frames)
- the purrr functions `safely()` and `possibly()` are **very** interesting. I think they could be useful outside of purrr as a friendlier way to do error-handling.

Step 3: `map_df()` to define the "outer" loop

```
# this dataframe will be the second input of fn_model
delta_nested %>%
  print()
```

```
# A tibble: 3 x 2
  id_sensor data
  <chr>     <list>
1 a         <tibble [327 x 2]>
2 b         <tibble [327 x 2]>
3 c         <tibble [327 x 2]>
```

```
# fn_model is receiving two inputs: one from list_model and from delta_nested
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  print()
```

```
# A tibble: 9 x 4
  id_model                 id_sensor data               model
  <chr>                    <chr>     <list>             <list>
1 newton_cooling           a         <tibble [327 x 2]> <S3: nls>
2 newton_cooling           b         <tibble [327 x 2]> <S3: nls>
3 newton_cooling           c         <tibble [327 x 2]> <S3: nls>
4 semi_infinite_simple     a         <tibble [327 x 2]> <S3: nls>
```

```
5 semi_infinite_simple     b         <tibble [327 x 2]> <S3: nls>
6 semi_infinite_simple     c         <tibble [327 x 2]> <S3: nls>
7 semi_infinite_convection a         <tibble [327 x 2]> <NULL>
8 semi_infinite_convection b         <tibble [327 x 2]> <NULL>
9 semi_infinite_convection c         <tibble [327 x 2]> <NULL>
```

- for each element of a list of model-functions, run the inner-loop function, and row-bind the results into a data-frame
- we want to discard the rows where the model failed
- we also want to investigate why they failed, but that's a different talk

Step 4: `map()` to identify the null models

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map(model, is.null)) %>%
  print()
```

```
# A tibble: 9 x 5
  id_model                 id_sensor data               model     is_null
  <chr>                    <chr>     <list>             <list>    <list>
1 newton_cooling           a         <tibble [327 x 2]> <S3: nls> <lgl [1]>
2 newton_cooling           b         <tibble [327 x 2]> <S3: nls> <lgl [1]>
3 newton_cooling           c         <tibble [327 x 2]> <S3: nls> <lgl [1]>
4 semi_infinite_simple     a         <tibble [327 x 2]> <S3: nls> <lgl [1]>
5 semi_infinite_simple     b         <tibble [327 x 2]> <S3: nls> <lgl [1]>
6 semi_infinite_simple     c         <tibble [327 x 2]> <S3: nls> <lgl [1]>
7 semi_infinite_convection a         <tibble [327 x 2]> <NULL>    <lgl [1]>
8 semi_infinite_convection b         <tibble [327 x 2]> <NULL>    <lgl [1]>
9 semi_infinite_convection c         <tibble [327 x 2]> <NULL>    <lgl [1]>
```

- using `map(model, is.null)` returns a list column
- to use `filter()`, we have to escape the weirdness

Step 5: `map_lgl()` to identify nulls and get out of the weirdness

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  print()
```

```
# A tibble: 9 x 5
  id_model                 id_sensor data               model     is_null
  <chr>                    <chr>     <list>             <list>    <lgl>
1 newton_cooling           a         <tibble [327 x 2]> <S3: nls> FALSE
2 newton_cooling           b         <tibble [327 x 2]> <S3: nls> FALSE
3 newton_cooling           c         <tibble [327 x 2]> <S3: nls> FALSE
4 semi_infinite_simple     a         <tibble [327 x 2]> <S3: nls> FALSE
5 semi_infinite_simple     b         <tibble [327 x 2]> <S3: nls> FALSE
6 semi_infinite_simple     c         <tibble [327 x 2]> <S3: nls> FALSE
```

```
7 semi_infinite_convection a         <tibble [327 x 2]> <NULL>    TRUE
8 semi_infinite_convection b         <tibble [327 x 2]> <NULL>    TRUE
9 semi_infinite_convection c         <tibble [327 x 2]> <NULL>    TRUE
```

- using `map_lgl(model, is.null)` returns a vector column

Step 6: `filter()` nulls and `select()` variables to clean up

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  filter(!is_null) %>%
  select(-is_null) %>%
  print()
```

```
# A tibble: 6 x 4
  id_model             id_sensor data               model
  <chr>                <chr>     <list>             <list>
1 newton_cooling       a         <tibble [327 x 2]> <S3: nls>
2 newton_cooling       b         <tibble [327 x 2]> <S3: nls>
3 newton_cooling       c         <tibble [327 x 2]> <S3: nls>
4 semi_infinite_simple a         <tibble [327 x 2]> <S3: nls>
5 semi_infinite_simple b         <tibble [327 x 2]> <S3: nls>
6 semi_infinite_simple c         <tibble [327 x 2]> <S3: nls>
```

Step 7: Calculate predictions on nested dataframe

```
predict_nested <-
  model_nested_new %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
```

```
# A tibble: 6 x 5
  id_model             id_sensor data               model     pred
  <chr>                <chr>     <list>             <list>    <list>
1 newton_cooling       a         <tibble [327 x 2]> <S3: nls> <dbl [327]>
2 newton_cooling       b         <tibble [327 x 2]> <S3: nls> <dbl [327]>
3 newton_cooling       c         <tibble [327 x 2]> <S3: nls> <dbl [327]>
4 semi_infinite_simple a         <tibble [327 x 2]> <S3: nls> <dbl [327]>
5 semi_infinite_simple b         <tibble [327 x 2]> <S3: nls> <dbl [327]>
6 semi_infinite_simple c         <tibble [327 x 2]> <S3: nls> <dbl [327]>
```

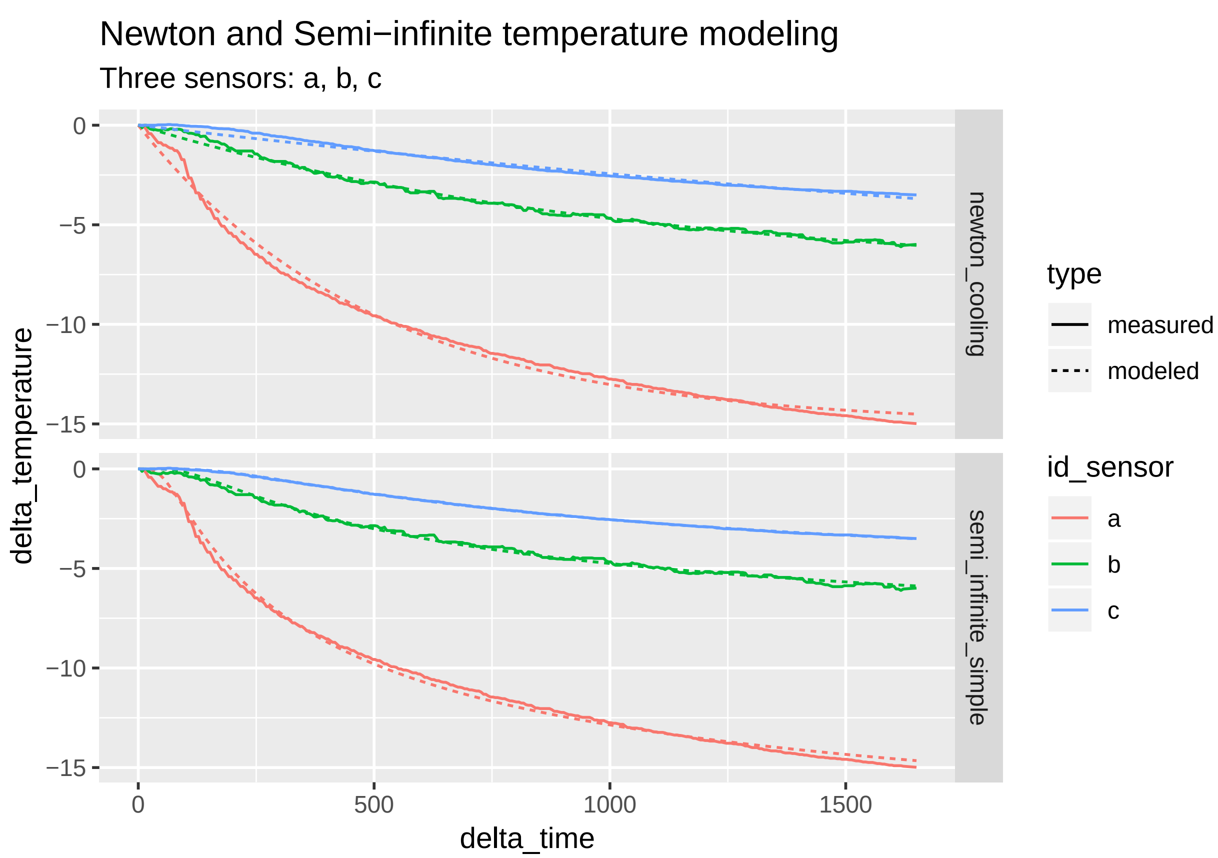
`unnest()`, make it tall and tidy

```
predict_tall <-
  predict_nested %>%
  unnest(data, pred) %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
```

```
# A tibble: 3,924 x 5
   id_model       id_sensor delta_time type    delta_temperature
   <chr>          <chr>          <dbl> <chr>               <dbl>
 1 newton_cooling a                  0 modeled             0
 2 newton_cooling a                  4 modeled            -0.120
 3 newton_cooling a                  8 modeled            -0.239
 4 newton_cooling a                 12 modeled            -0.357
 5 newton_cooling a                 17 modeled            -0.503
 6 newton_cooling a                 22 modeled            -0.648
 7 newton_cooling a                 27 modeled            -0.792
 8 newton_cooling a                 32 modeled            -0.934
 9 newton_cooling a                 37 modeled            -1.07
10 newton_cooling a                 42 modeled            -1.21
# ... with 3,914 more rows
```

Visualize the predictions

```r
predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  facet_grid(id_model ~ .) +
  labs(title = "Newton and Semi-infinite temperature modeling",
       subtitle = "Three sensors: a, b, c")
```



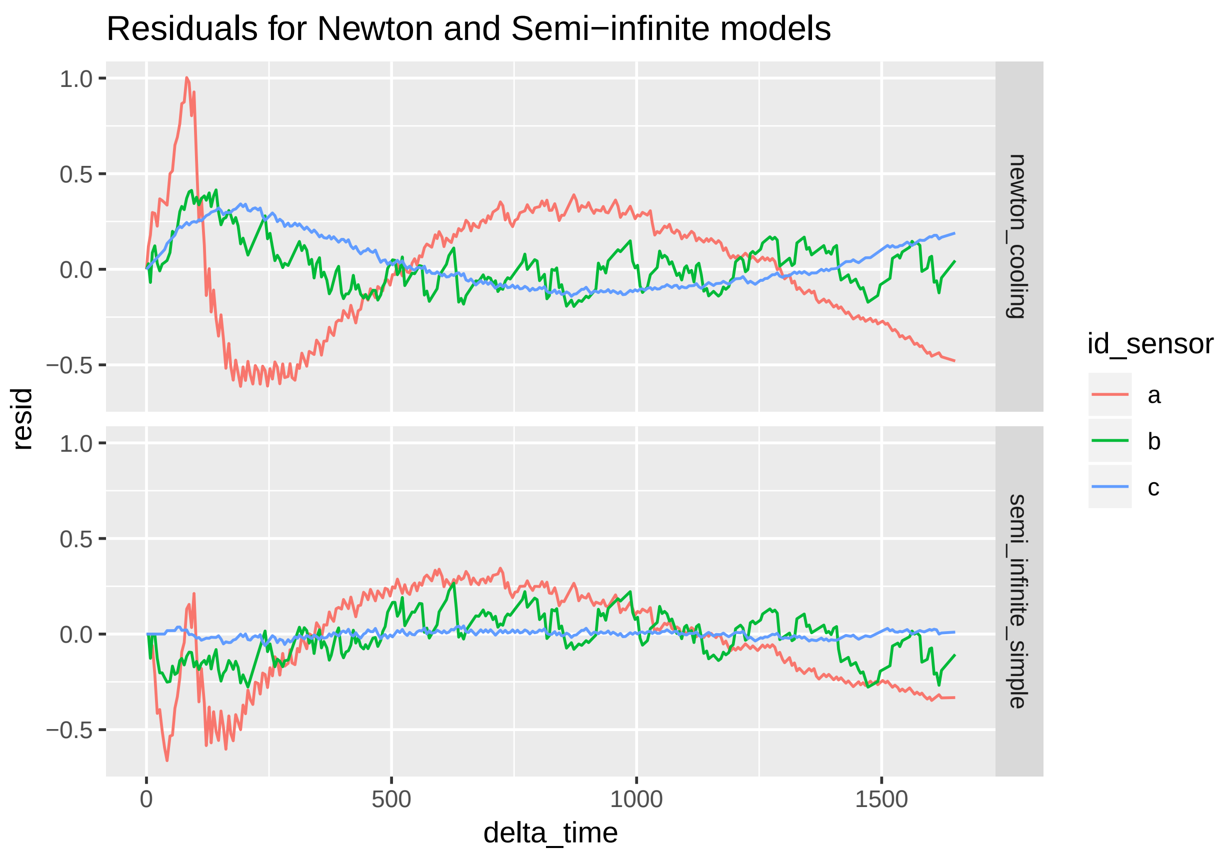
Let's get the residuals

```
resid <-
  model_nested_new %>%
  mutate(resid = map(model, resid)) %>%
  unnest(data, resid) %>%
  print()
```

```
# A tibble: 1,962 x 5
   id_model       id_sensor resid delta_time delta_temperature
   <chr>          <chr>     <dbl>      <dbl>             <dbl>
 1 newton_cooling a         0              0              0
 2 newton_cooling a         0.120          4              0
 3 newton_cooling a         0.179          8             -0.06
 4 newton_cooling a         0.297         12             -0.06
 5 newton_cooling a         0.292         17             -0.211
 6 newton_cooling a         0.225         22             -0.423
 7 newton_cooling a         0.369         27             -0.423
 8 newton_cooling a         0.360         32             -0.574
 9 newton_cooling a         0.348         37             -0.726
10 newton_cooling a         0.335         42             -0.878
# ... with 1,952 more rows
```

And visualize them

```
resid %>%
  ggplot(aes(x = delta_time, y = resid)) +
  geom_line(aes(color = id_sensor)) +
  facet_grid(id_model ~ .) +
  labs(title = "Residuals for Newton and Semi-infinite models")
```



Using broom package to look at model-statistics

We will use a previous defined dataframe with the model and data:

```
model_nested_new %>%
  print()
```

```
# A tibble: 6 x 4
  id_model             id_sensor data               model
  <chr>                <chr>     <list>             <list>
1 newton_cooling       a         <tibble [327 x 2]> <S3: nls>
2 newton_cooling       b         <tibble [327 x 2]> <S3: nls>
3 newton_cooling       c         <tibble [327 x 2]> <S3: nls>
4 semi_infinite_simple a         <tibble [327 x 2]> <S3: nls>
5 semi_infinite_simple b         <tibble [327 x 2]> <S3: nls>
6 semi_infinite_simple c         <tibble [327 x 2]> <S3: nls>
```

The `tidy()` function extracts statistics from a model.

```
# apply over model_nested_new but only three variables
model_parameters <-
  model_nested_new %>%
  select(id_model, id_sensor, model) %>%
  mutate(tidy = map(model, tidy)) %>%
  select(-model) %>%
  unnest() %>%
  print()
```

```
# A tibble: 12 x 7
   id_model      id_sensor term      estimate std.error statistic   p.value
   <chr>         <chr>     <chr>        <dbl>     <dbl>     <dbl>     <dbl>
 1 newton_cool~ a         delta_te~   -15.1      0.0526    -286.  0.
 2 newton_cool~ a         tau_0       500.       4.84       103.  1.07e-250
 3 newton_cool~ b         delta_te~    -7.59     0.0676    -112.  6.38e-262
 4 newton_cool~ b         tau_0      1041.      16.2         64.2 9.05e-187
 5 newton_cool~ c         delta_te~    -9.87     0.704      -14.0 3.16e- 35
 6 newton_cool~ c         tau_0      3525.     299.          11.8 5.61e- 27
 7 semi_infini~ a         delta_te~   -21.5      0.0649    -332.  0.
 8 semi_infini~ a         tau_0       139.       1.15       121.  2.14e-272
 9 semi_infini~ b         delta_te~   -10.6      0.0515    -206.  0.
10 semi_infini~ b         tau_0       287.       2.58       111.  1.46e-260
11 semi_infini~ c         delta_te~    -8.04     0.0129    -626.  0.
12 semi_infini~ c         tau_0       500.       1.07       468.  0.
```

Get a sense of the coefficients

```r
model_summary <-
  model_parameters %>%
  select(id_model, id_sensor, term, estimate) %>%
  spread(key = "term", value = "estimate") %>%
  print()
```

```
# A tibble: 6 x 4
  id_model              id_sensor delta_temperature_0 tau_0
  <chr>                 <chr>                   <dbl> <dbl>
1 newton_cooling        a                      -15.1   500.
2 newton_cooling        b                       -7.59 1041.
3 newton_cooling        c                       -9.87 3525.
4 semi_infinite_simple a                      -21.5   139.
5 semi_infinite_simple b                      -10.6   287.
6 semi_infinite_simple c                       -8.04  500.
```

Summary

- this is just a smalll part of purrr
- there seem to be parallels between `tidyr::nest()/purrr::map()` and `dplyr::group_by()/dplyr::do()`
  - to my mind, the purrr framework is more understandable
  - update tweet from Hadley

References from Hadley:

- purrr 0.1.0 announcement
- purrr 0.2.0 announcement
- chapter from Garrett Grolemund and Hadley's forthcoming book