KOSTAT-UNFPA Summer Seminar on Population

Workshop 1. Demography in R

Day 3: Worked example of tidy processing

Instructor: Tim Riffe
tim.riffe@ehu.eus

Assistants:
Jinyeon Jo: jyjo43043@gmail.com
Rustam Tursun-Zade: rustam.tursunzade@gmail.com

29 July 2022

Contents

1 Tidy data

1.1 Definition

Recall that tidy data follows a standard structure where each column is a variable, each row is an observation, and each cell is a value. Anything else is messy. It's literally that straightforward. A more complete definition can be found here: https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html Demographic data is often delivered in a tidy format. When it is not, then it can be reshaped into a tidy format.

1.2 Example of not-tidy data

The following layout (screenshot from Excel) is not tidy. This example data was manually extracted from here: https://appsso.eurostat.ec.europa.eu/nui/show.do?dataset=demo_fasec&lang=en, and the given format was concocted with EUROSTAT's online data widget.

	A	B	C	D	E	F	G	H	I
1	Live births by mother's age and newborn's sex [demo_fasec]								
2									
3	Last update	28.06.21							
4	Extracted o	20.07.21							
5	Source of d	Eurostat							
6									
7	SEX	Total							
8	UNIT	Number							
9									
10	AGE	GEO/TIME	2011	2012	2013	2014	2015	2016	
11	Total	Belgium	128,705	128,051	125,606	125,014	122,274	121,896	
12	Total	Czechia	108,673	108,576	106,751	109,860	110,764	112,663	
13	Total	Spain	470,553	453,348	424,440	426,076	418,432	408,734	
14	Total	Croatia	41,197	41,771	39,939	39,566	37,503	37,537	
15	15 years	Belgium	74	72	48	52	47	37	
16	15 years	Czechia	67	53	55	62	66	47	
17	15 years	Spain	414	379	391	375	390	341	
18	15 years	Croatia	35	36	42	27	24	18	
19	16 years	Belgium	171	161	183	180	130	119	
20	16 years	Czechia	224	225	204	230	177	223	
21	16 years	Spain	930	896	855	878	806	813	
22	16 years	Croatia	97	102	92	98	86	75	
23	17 years	Belgium	436	417	338	346	303	292	
24	17 years	Czechia	511	483	422	412	447	417	
25	17 years	Spain	1,784	1,677	1,581	1,489	1,493	1,394	
26	17 years	Croatia	228	191	172	182	182	158	
27	18 years	Belgium	822	753	606	621	559	523	
28	18 years	Czechia	818	952	801	771	752	744	
29	18 years	Spain	2,914	2,699	2,397	2,309	2,217	2,182	
30	18 years	Croatia	428	360	365	356	308	314	
31	19 years	Belgium	1,470	1,346	1,056	1,027	967	917	
32	19 years	Czechia	1,434	1,338	1,334	1,253	1,164	1,192	
33	19 years	Spain	4,160	3,866	3,529	3,365	3,220	3,113	

The main thing that makes it not-tidy are the years spread over columns. These should be stacked into to columns: `TIME` (per the original codes) and `Births`, which are the values in the cells. The fact that `AGE` is coded as an arithmetically unusable character string is something we'll want to recode, but it is orthogonal to the *tidiness* of the data. Finally, we will ensure that age-specific births sum up to the stated total births per year and country.

To follow along, create a folder in your project called `Data`. Then, go to the `Data` folder of the course repository on github: https://github.com/timriffe/KOSTAT_Workshop1/blob/master/Data/demo_fasec.xlsx and click `Download`. Move it to the data folder you just made. You can also do the same for a second file that we'll use later today: https://github.com/timriffe/KOSTAT_Workshop1/blob/master/Data/demo_pjan.xlsx

2 Tidy processing fertiltiy data

Today's entire session will be working with this smallish births dataset.

2.1 Reshape, Rescale, Recode it

We'll use the `read_excel()` function from the `readxl` package to get the data in. First let's look at the help file using `?read_excel`. Visual inspection of the data shows us that we need to skip several rows, plus there's a note at the bottom of the sheet that we want to ignore. We specify an explicit cell range using the argument `range` and giving spreadsheet coordinates `range = "A10:H158"` (determined based on visual inspection of the file).

```
library(tidyverse)
library(readxl)
# ?read_excel
Wide <- read_excel(
           path = "Data/demo_fasec.xlsx",
           range = "A10:H158")
glimpse(Wide)
```

```
## Rows: 148
## Columns: 8
## $ AGE        <chr> "Total", "Total", "Total", "Total", "15 years", "15 years",~
## $ `GEO/TIME` <chr> "Belgium", "Czechia", "Spain", "Croatia", "Belgium", "Czech~
## $ `2011`     <dbl> 128705, 108673, 470553, 41197, 74, 67, 414, 35, 171, 224, 9~
## $ `2012`     <dbl> 128051, 108576, 453348, 41771, 72, 53, 379, 36, 161, 225, 8~
## $ `2013`     <dbl> 125606, 106751, 424440, 39939, 48, 55, 391, 42, 183, 204, 8~
## $ `2014`     <dbl> 125014, 109860, 426076, 39566, 52, 62, 375, 27, 180, 230, 8~
## $ `2015`     <dbl> 122274, 110764, 418432, 37503, 47, 66, 390, 24, 130, 177, 8~
## $ `2016`     <dbl> 121896, 112663, 408734, 37537, 37, 47, 341, 18, 119, 223, 8~
```

2.1.1 `pivot_longer()` (`pivot_wider()`)

These data are not tidy because `TIME` is spread over columns. Instead we should have a column called `TIME`, containing the years, and the cell values currently in the columns `2011` to `2016` should be in a column called births. The function `pivot_longer()` will do this for us. See `?pivot_longer` or the handout from the previous session.

```
Long <- pivot_longer(
          data = Wide,
          cols = `2011`:`2016`,
          names_to = "TIME",
          values_to = "Births")

glimpse(Long)
```

```
## Rows: 888
## Columns: 4
## $ AGE        <chr> "Total", "Total", "Total", "Total", "Total", "Total", "Tota~
## $ `GEO/TIME` <chr> "Belgium", "Belgium", "Belgium", "Belgium", "Belgium", "Bel~
## $ TIME       <chr> "2011", "2012", "2013", "2014", "2015", "2016", "2011", "20~
## $ Births     <dbl> 128705, 128051, 125606, 125014, 122274, 121896, 108673, 108~
```

A nice thing about `pivot_longer` is you can specify column ranges using names (see how we put back-ticks on years? That's so that 2011 doesn't get interpreted at the 2011th column!). Integer ranges will also do, as would listing out the columns by name or position. There are other tricks for intelligently picking out columns, see `?tidyr_tidy_select`. From that help file, we see other things would have worked in our case, for example `where(is.double)`, which is handy.

```
pivot_longer(
          data = Wide,
          cols = where(is.double),
          names_to = "TIME",
          values_to = "Births")
```

2.1.2 select() columns

Note, some of these columns are clean and ready to use (`TIME` and `Births`), but `GEO/TIME` is not a useful column name, and the values in `AGE` might not be useful in practice. Instead, maybe we want `AGE` to be an integer value so that we can use it to sort, or use it in a numeric way in a plot axis. Maybe also we don't want to have a `Total` value for `AGE`, but instead we want to ensure that the age-specific births add up to the total? This will give us a chance to showcase some more tidyverse tools:

Select, rearrange, and rename **columns** using `select()`. Notice that i) the left side of each `=` is the new column name, ii) the order you list the columns is the new order, iii) if you forget to list a column you lose it! iv) when in doubt, but back-ticks on the column name to ensure it won't be misinterpreted. `GEO/TIME` looks like math, so stick it in back-ticks...

```
Long <- select(
  .data = Long,
  Country = `GEO/TIME`,
  Age = AGE,
  Year = TIME,
  Births
)
head(Long)
```

```
## # A tibble: 6 x 4
##   Country Age   Year  Births
##   <chr>   <chr> <chr>  <dbl>
## 1 Belgium Total 2011  128705
## 2 Belgium Total 2012  128051
## 3 Belgium Total 2013  125606
## 4 Belgium Total 2014  125014
## 5 Belgium Total 2015  122274
## 6 Belgium Total 2016  121896
```

2.1.3 pipes, %>%

Let's re-express the above using *pipes*. Pipes allow us to string together data operations into a single sequentially evaluated execution step. Thus far we have read in the data, reshaped it, and re-specified columns. All together, this becomes:

```
Long <-
  read_excel(
    path = "Data/demo_fasec.xlsx",
    range = "A10:H158") %>%
  pivot_longer(
    cols = `2011`:`2016`,
    names_to = "Year",
    values_to = "Births") %>%
  select(
     Country = `GEO/TIME`,
     Year,
     Age = AGE,
     Births)
```

This reads as "first `read_excel()`, then `pivot_longer()`, then `select()` columns, renaming some of them".

```r
Long <-
  # first read in from Excel
  read_excel(
    path = "Data/demo_fasec.xlsx",
    range = "A10:H158") %>%
  # stack years
  pivot_longer(
    cols = `2011`:`2016`,
    names_to = "Year",
    values_to = "Births") %>%
  # pick out the columns
  select(
     Country = `GEO/TIME`,
     Year,
     Age = AGE,
     Births)
```

Note: you can run this code by simply placing the cursor anywhere in the pipeline and pressing `Ctrl + Enter`. There is no need to select the whole statement before running, although this also works (you could in this case also click the green play arrow).

We will augment this pipeline step by step and then recompose it in its entirety at the end.

2.1.4 filter(), mutate(), group_by()

So what age classes to we have? `unique()` picks out just the unique values present in a vector.

```r
# same thing using tidy:
Long %>%
  pull(Age) %>% # pull() extracts column as vector
  unique()
```

```
##  [1] "Total"    "15 years" "16 years" "17 years" "18 years" "19 years"
##  [7] "20 years" "21 years" "22 years" "23 years" "24 years" "25 years"
## [13] "26 years" "27 years" "28 years" "29 years" "30 years" "31 years"
## [19] "32 years" "33 years" "34 years" "35 years" "36 years" "37 years"
## [25] "38 years" "39 years" "40 years" "41 years" "42 years" "43 years"
## [31] "44 years" "45 years" "46 years" "47 years" "48 years" "49 years"
## [37] "Unknown"
```

The `Age` column should be changed to consist in just integers. But this raises another issue: what to do with the `Total` and `Unknown` ages? My preference is usually to redistribute unknowns proportional to the distribution of any *knowns*:

$$\widehat{Y_x} = Y_x + Y_{unknown} * \frac{Y_x}{\sum Y_x}$$

where the denominator excludes unknowns... This is just the same as rescaling the distributions of *known* ages to sum to the stated total

$$\widehat{Y_x} = Y * \frac{Y_x}{\sum Y_x}$$

(where x excludes unknowns)

Once we do one or the other of these operations, we'll end up with just ages `15 years` through `49 years`, and can convert using string operators. We can throw out either `Total` or `Unknown` using `filter()` to select rows. Calculations to redistribute can be done using the function `mutate()`. The basic structure of said operation would be something similar to:

```
# don't run this
Long %>%
  # 1
  mutate(TOT = Births[Age == "Total"]) %>%
  # 2
  filter(! Age %in% c("Total", "Unknown")) %>%
  # 3
  mutate(Births = Births / sum(Births) * TOT)
```

I'll first explain the basic logic, then why it won't *yet* work as expected. In step 1, we use `mutate()` to create a new column called `TOT`, which just repeats the respective value for each row of the data.

Now for the `filter()` statement.

2.1.5 Time out for logicals

Each value of `TOT` is intended to be the value of `Births` where `Age` is equal to `"Total"`. Note `==` is a *logical* equals, meaning you're asking if values are equal. The result will be `TRUE`, `FALSE`, or `NA` if pertinent.

Example:

```
1:5 == 5
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE
```

Other useful logical operators include `!=` (inequality), `<`, `>`, `<=`, `>=`. Further logical functions include: `is.na()`, `any()`, `all()`. Each of these operators and functions is vectorized, meaning they can evaluate long vectors of expressions element-wise.

Here we want to use this logical vector to select values:

```
abcde <- c("a","b","c","d","e")
abcde[1:5 == 5]
```

```
## [1] "e"
```

Namely, we get back the values where the logical vector evaluates to `TRUE`.

Given a columns `TOT`, we can remove age classes equal to `"Total"` or `"Unknown"` with `filter()`. `%in%` is a logical operator for set membership.

```
c("a","d","k") %in% abcde
```

```
## [1]  TRUE  TRUE FALSE
```

Finally, `mutate()` can be used to do the rescale operation using our basic arithmetic.

2.1.6 `group_by()` (`ungroup()`)

An issue that you may foresee at this point is that either of the above formulas is independent within each `Country` and `Year`. We can deal with this by declaring each combination of these two variables as an *independent* group using `group_by()`, and then removing groups when no longer needed using `ungroup()`. That's just good housekeeping, but it keeps the pipeline rigorous: You can assume group declarations will persist until explicitly removed.

```
Long2 <-
  Long %>%
  # add group metadata
  group_by(Country, Year) %>%
  # raise Total count to column for element-wise rescale
  mutate(TOT = Births[Age == "Total"]) %>%
  # throw out Total and Unknown ages
  filter(! Age %in% c("Total", "Unknown")) %>%
  # rescale proportions known to stated total
  mutate(Births = Births / sum(Births) * TOT) %>%
  # groups no longer needed, let's remove them:
  ungroup()
```

Finally, we can clean up the `Age` column! Here I'll take the string substitution strategy, although other options would also work. `gsub()` looks for a pattern in the string `" years"` and replaces it. In this case, I replace with an empty string `""`, so `"15 years"` becomes `"15"`, still a character string. We can then modify it in the same `mutate()` call: comma-separated statements in `mutate()` are evaluated in sequence, and they can be sequentially dependent!

```
Long2 %>%
  mutate(Age = gsub(Age,
                    pattern = " years",
                    replacement = ""),
         Age = as.integer(Age))
```

```
## # A tibble: 840 x 5
##    Country Year    Age Births    TOT
##    <chr>   <chr> <int>  <dbl>  <dbl>
##  1 Belgium 2011     15   74.1 128705
##  2 Belgium 2012     15   72.1 128051
##  3 Belgium 2013     15   48.5 125606
##  4 Belgium 2014     15   52.0 125014
##  5 Belgium 2015     15   47.5 122274
##  6 Belgium 2016     15   37.4 121896
##  7 Czechia 2011     15   67.0 108673
##  8 Czechia 2012     15   53.0 108576
##  9 Czechia 2013     15   55.0 106751
## 10 Czechia 2014     15   62.0 109860
## # ... with 830 more rows
## # i Use `print(n = ...)` to see more rows
```

A helper function from the `readr` package `parse_number()` can do this for us in a single step:

```
Long2 %>%
  mutate(Age = parse_number(Age))
```

Note, you can also use pipes inside function calls, like `mutate()`, so the above could become:

```
Long2 %>%
  mutate(Age = Age %>%
               gsub(
                 pattern = " years",
```

```
                  replacement = "") %>%
                as.integer())
```

```
## # A tibble: 840 x 5
##    Country Year    Age Births    TOT
##    <chr>   <chr> <int>  <dbl>  <dbl>
##  1 Belgium 2011     15   74.1 128705
##  2 Belgium 2012     15   72.1 128051
##  3 Belgium 2013     15   48.5 125606
##  4 Belgium 2014     15   52.0 125014
##  5 Belgium 2015     15   47.5 122274
##  6 Belgium 2016     15   37.4 121896
##  7 Czechia 2011     15   67.0 108673
##  8 Czechia 2012     15   53.0 108576
##  9 Czechia 2013     15   55.0 106751
## 10 Czechia 2014     15   62.0 109860
## # ... with 830 more rows
## # i Use `print(n = ...)` to see more rows
```

Depending on what you're doing, one or the other of these could be more *legible*. Human-legible code is more robust than illegible code, can we agree on this point?

2.1.7 Bringing it all together

There are times when it may make sense to keep steps separate, in separate data objects, but our first example is a case of wanting to keep all steps contained in a single pipeline. That's because the intermediate pieces are redundant and add no value. Combined into a single pipeline, we'd end up with something like this:

```
Births <-
  # first read in from Excel
  read_excel(
    path = "Data/demo_fasec.xlsx",
    range = "A10:H158") %>%
  # stack years
  pivot_longer(
    cols = `2011`:`2016`,
    names_to = "Year",
    values_to = "Births") %>%
  # pick out the columns
  select(
     Country = `GEO/TIME`,
     Year,
     Age = AGE,
     Births) %>%
  # add group metadata
  group_by(Country, Year) %>%
  # raise Total count to column for element-wise rescale
  mutate(TOT = Births[Age == "Total"]) %>%
  # throw out Total and Unknown ages
  filter(! Age %in% c("Total", "Unknown")) %>%
  # rescale proportions known to stated total
  mutate(Births = Births / sum(Births) * TOT) %>%
```

```r
  # groups no longer needed, let's remove them:
  ungroup() %>%
  # clean up Age
  mutate(Age = Age %>%
               gsub(
                 pattern = " years",
                 replacement = "") %>%
               as.integer(),
         Year = as.integer(Year)) %>%
  # sort rows
  arrange(Country, Year, Age) %>%
  # remove TOT column, no longer needed
  select(-TOT)

# have a look
glimpse(Births)
```

```
## Rows: 840
## Columns: 4
## $ Country <chr> "Belgium", "Belgium", "Belgium", "Belgium", "Belgium", "Belgiu~
## $ Year    <int> 2011, 2011, 2011, 2011, 2011, 2011, 2011, 2011, 2011, 2011, 20~
## $ Age     <int> 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30~
## $ Births  <dbl> 74.1071, 171.2475, 436.6310, 823.1896, 1472.1275, 2032.9379, 2~
```

This is a tidy pipeline. And tidy code, no matter who writes it, usually ends up looking something like this. To finish off the pipeline, I've sorted the rows. `arrange(Country, Year, Age)` sorts `Age` within `Year` within `Country`), and we delete the `TOT` column using subtraction inside `select()`.

You see all those annotations between many of the pipe steps? That's not *just* for you, the reader. It's good practice to do that. Possibly because someone else might like to interpret your code, so why not make it easier, but also you should comment your code out of respect for *future you*, because *future you* won't remember what you were thinking when you wrote it.

2.2 Group to 5-year ages with `summarize()`

Aggregation typically implies a reduction in the number of rows in a data set. Let's see examples of grouping countries, grouping ages, and calculating marginal sums. Grouping ages or years often follows a similar logic. We will exploit to the *modulo* operator, `%%`, which tells us the remainder after Euclidean division. Example:

```r
a <- 1:10
a %% 2
```

```
##  [1] 1 0 1 0 1 0 1 0 1 0
```

```r
a %% 5
```

```
##  [1] 1 2 3 4 0 1 2 3 4 0
```

That is the divisor (2 or 5) is subtracted away an integer number of times until what remains is smaller than the divisor. This is useful for redefining `Age`, see:

```r
Age <- 0:20
Age - Age %% 5
```

```
## [1]  0  0  0  0  0  5  5  5  5  5 10 10 10 10 10 15 15 15 15 15 20
```

That is, subtracting `Age` modulo 5 from a vector of single ages tells you the lower bound of the five year age group that each single age lays within. We can then use this new age vector to group data, and finally we aggregate `Births` using `summarize()`.

```
Births %>%
  mutate(Age = Age - Age %% 5) %>%
  group_by(Country, Year, Age) %>%
  summarize(Births = sum(Births),
            .groups = "drop")
```

```
## # A tibble: 168 x 4
##    Country  Year   Age Births
##    <chr>   <int> <dbl>  <dbl>
##  1 Belgium  2011    15  2977.
##  2 Belgium  2011    20 18239.
##  3 Belgium  2011    25 44498.
##  4 Belgium  2011    30 41905.
##  5 Belgium  2011    35 17368.
##  6 Belgium  2011    40  3542.
##  7 Belgium  2011    45   175.
##  8 Belgium  2012    15  2751.
##  9 Belgium  2012    20 17631.
## 10 Belgium  2012    25 43746.
## # ... with 158 more rows
## # i Use `print(n = ...)` to see more rows
```

`Births = sum(Births)` might look strange. The left side is a single outgoing row, whereas the right side is a vector with five values. Our dataset of 840 rows is in this way reduced to $840/5 = 168$ rows. This works out cleanly in our case because the age groups were cleanly divisible. Note the argument `.groups = "drop"` at the end of `summarize()`, this is just the same as adding `%>% ungroup()` at the end of the pipeline.

2.2.1 Marginal sums

The result of a summary statement could be just a single row, in this case a probably not-useful result.

```
Births %>%
  summarize(Births = sum(Births))
```

```
## # A tibble: 1 x 1
##    Births
##     <dbl>
## 1 4247929
```

To get totals by `Country` and `Year`, once again we apply groups:

```
Births %>%
  group_by(Country, Year) %>%
  summarize(Births = sum(Births),
            .groups = "drop")
```

```
## # A tibble: 24 x 3
##    Country  Year Births
```

```
##    <chr>   <int>  <dbl>
##  1 Belgium  2011 128705
##  2 Belgium  2012 128051
##  3 Belgium  2013 125606
##  4 Belgium  2014 125014
##  5 Belgium  2015 122274
##  6 Belgium  2016 121896
##  7 Croatia  2011  41197
##  8 Croatia  2012  41771
##  9 Croatia  2013  39939
## 10 Croatia  2014  39566
## # ... with 14 more rows
## # i Use `print(n = ...)` to see more rows
```

Likewise, we could group countries using `case_when()`. First we use `case_when()` then I'll explain how it works.

```
Births %>%
  mutate(Country_Group = case_when(Country == "Czechia" ~ "A",
                                   Country == "Spain" ~ "A",
                                   Country == "Belgium" ~ "B",
                                   Country == "Croatia" ~ "B")) %>%
  group_by(Country_Group, Year) %>%
  summarize(Births = sum(Births),
            .groups = "drop")
```

```
## # A tibble: 12 x 3
##    Country_Group  Year Births
##    <chr>         <int>  <dbl>
##  1 A              2011 579226
##  2 A              2012 561924
##  3 A              2013 531191
##  4 A              2014 535936
##  5 A              2015 529196
##  6 A              2016 521397
##  7 B              2011 169902
##  8 B              2012 169822
##  9 B              2013 165545
## 10 B              2014 164580
## 11 B              2015 159777
## 12 B              2016 159433
```

2.2.2 `case_when()`

This helper function is a generalization of `ifelse()` or `if_else()`, as may be familiar from other programs such as Excel. `case_when()` is premised on you being able to delimit all cases given in your data exhaustively. Each case is comma separated and defined in formula notation, where `~` separates a left and a right side. On the left of `~` you define the case with a **logical** statement and on the right side you specify what to assign for that case. By the end of the `case_when()` statement all cases must be handled. Further, cases are handled in the order specified, so where pertinent it makes sense to list cases ordered from specific to general. If there is a most general case meaning something like *everything else*, then you can end `case_when()` with `TRUE ~ 1` (or whatever value you want).

For example, just to demonstrate the concepts, say I have an algorithm where you start with an integer. If the integer is: 1. divisible by 6 then divide by 2 and add 1 2. divisible by 3 then multiply by 2 3. odd add 1 4. even add 2

This is a silly algorithm, I admit. Note only the first condition produces an *odd* result. Note, all integers are handled by conditions 1-4. Note that conditions 3 and 4 handle more cases than conditions 1 and 2. Note also that condition 1 is more specific than 2, because all numbers divisible by 6 are also divisible by 3, but not vice versa. Using `case_when()` and exercising our new modulo skills, an example would be:

```
a <- 1:17
case_when(a %% 6 == 0 ~ a / 2 + 1,
          a %% 3 == 0 ~ a * 2,
          a %% 2 == 1 ~ a + 1,
          a %% 2 == 0 ~ a + 2)
```

```
##  [1]  2  4  6  6  6  4  8 10 18 12 12  7 14 16 30 18 18
```

If we write the same cases but changing the order of the first two conditions, we see that condition (1) from the initial algorithm is never activated, because divisibility by 3 handles the case earlier.

```
case_when(a %% 3 == 0 ~ a * 2,
          a %% 6 == 0 ~ a / 2 + 1,
          a %% 2 == 1 ~ a + 1,
          a %% 2 == 0 ~ a + 2)
```

```
##  [1]  2  4  6  6  6 12  8 10 18 12 12 24 14 16 30 18 18
```

2.2.3 Weighted means

Our main use of `summarize()` today will be for evaluating weighted means. More specifically, we'll calculate the mean age at childbearing.

In general a weighted mean is defined as

$$\bar{x} = \frac{\sum x_i \cdot w_i}{\sum w_i}$$

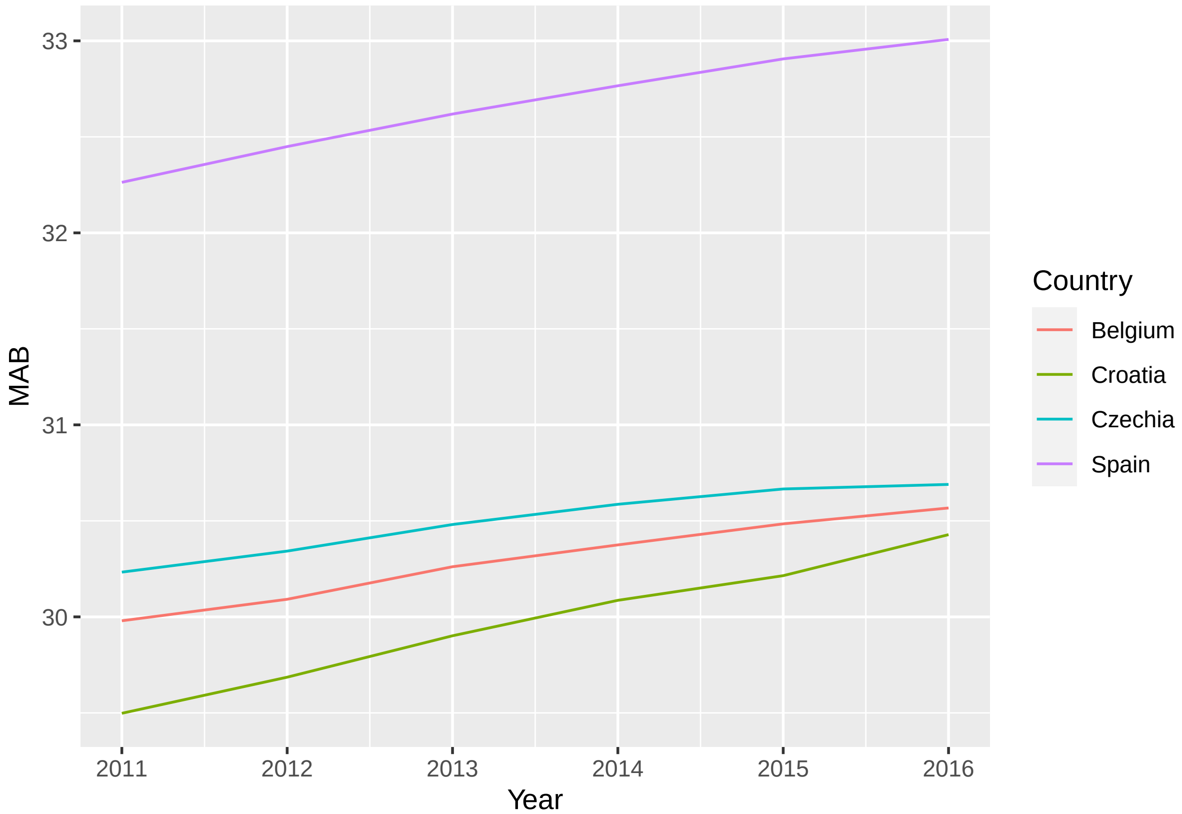
For the mean age at childbearing, x is age (exact age at mid-interval we prefer), and w should be either birth counts or age-specific fertility. Since we don't have exposures (yet) to calculate fertility rates, we'll just use raw births by age as the weights.

```
MAB <-
  Births %>%
  mutate(Age = Age + .5) %>%
  group_by(Country, Year) %>%
  summarize(MAB = sum(Births * Age) / sum(Births),
            .groups = "drop")
glimpse(MAB)
```

```
## Rows: 24
## Columns: 3
## $ Country <chr> "Belgium", "Belgium", "Belgium", "Belgium", "Belgium", "Belgiu~
## $ Year    <int> 2011, 2012, 2013, 2014, 2015, 2016, 2011, 2012, 2013, 2014, 20~
## $ MAB     <dbl> 29.97964, 30.09161, 30.26122, 30.37466, 30.48438, 30.56714, 29~
```

While we're here, how about a plot teaser, even though we don't get serious about `ggplot2` until Thursday:

```
MAB %>%
  ggplot(aes(x = Year,
             y = MAB,
             group = Country,
             color = Country)) +
  geom_line()
```



Allow me to pose a question: All of these lines are increasing. These mean ages are based on observed births in each mother age group, which are a product of fertility rates and population size in each age group. How much of this trend do you suppose is due to changes in age-specific fertility rates versus changes in underlying population structure? To answer this question, we will need to obtain, harmonize, and merge population data to the birth counts data we've been working with. Let's get to it!

3 Process population data

Often we get data from different sources that needs to be merged (or joined) into a single merged dataset in order to carry out an analysis. In this case, I've pulled January 1st female population counts data from EUROSTAT, and to make things interesting it's formatted differently and has its own challenges. Our objective is to prepare this data to join it to the births data from above. This will allow us to repeat some concepts.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Population on 1 January by age and sex [demo_pjan]											
2												
3	Last update	05.07.21										
4	Extracted o	21.07.21										
5	Source of d	Eurostat										
6												
7	SEX	Total										
8	UNIT	Number										
9												
10	TIME	GEO/AGE	Less than	1 year	2 years	3 years	4 years	5 years	6 years	7 years	8 years	9 year
11	2012	Belgium	128,701	131,406	130,525	131,088	128,720	128,016	125,150	123,401	120,347	11
12	2012	Bulgaria	66,214	69,093	72,786	70,163	67,761	66,918	65,285	64,906	62,602	6
13	2012	Czechia	108,753	119,389	121,285	122,912	118,351	108,766	102,953	98,187	94,357	9
14	2012	Denmark	59,306	64,168	63,842	66,358	65,529	66,506	65,597	65,693	65,667	6
15	2012	Germany ir	658,332	679,236	669,579	689,691	689,748	677,338	687,268	705,953	706,644	72
16	2012	Estonia	14,728	15,945	15,765	15,940	15,588	14,679	14,097	13,625	12,716	1
17	2012	Ireland	73,041	71,754	72,015	71,481	69,861	66,438	64,950	65,078	63,891	6
18	2012	Greece	105,955	110,528	115,191	112,981	108,392	110,138	106,916	105,638	105,963	10
19	2012	Spain	475,603	483,549	494,158	521,121	502,163	498,183	487,772	482,425	473,952	45
20	2012	France	789,537	808,857	803,643	809,817	808,910	824,503	809,663	805,752	803,913	81
21	2012	France (me	763,531	781,550	775,094	780,356	778,613	794,066	778,941	775,211	773,146	77
22	2012	Croatia	40,948	43,350	44,149	43,130	41,278	41,225	42,435	40,384	39,909	4
23	2012	Italy	531,372	544,814	554,608	561,250	560,099	560,414	556,531	561,023	556,258	55
24	2012	Cyprus	9,737	10,112	10,058	9,757	9,217	9,415	8,960	9,060	8,868	
25	2012	Latvia	18,573	19,333	21,175	22,991	22,253	21,073	20,283	19,123	19,503	1
26	2012	Lithuania	30,194	30,534	31,190	29,502	27,564	27,133	26,924	26,954	27,249	2
27	2012	Luxembour	5,842	6,049	5,894	5,904	5,872	5,917	5,881	5,865	5,851	
28	2012	Hungary	87,680	89,906	96,679	99,423	97,391	100,953	97,693	94,168	93,822	9
29	2012	Malta	4,161	3,948	4,068	4,101	3,842	3,830	3,860	3,834	3,979	
30	2012	Netherland	179,653	184,741	185,690	185,962	182,690	185,808	187,772	193,575	200,289	20
31	2012	Austria	77,562	80,080	77,993	79,552	78,512	80,388	80,685	81,887	80,584	8
32	2012	Poland	380,668	412,836	429,002	427,212	399,979	380,049	366,058	354,559	348,459	35
33	2012	Portugal	95,703	99,519	96,085	100,281	98,361	101,075	105,083	103,654	106,372	10
34	2012	Romania	185,286	207,241	215,521	216,015	212,002	213,127	215,766	209,761	210,066	20
35	2012	Slovenia	22,003	22,576	22,057	22,297	20,477	19,543	18,605	18,416	17,734	1
36	2012	Slovakia	60,598	57,712	59,722	57,077	54,439	54,008	54,527	54,020	51,694	5
37	2012	Finland	60,074	61,504	61,109	60,486	59,804	60,185	58,901	59,077	57,982	5
38	2012	Sweden	112,114	117,242	114,285	112,444	111,284	110,630	106,635	106,647	104,977	10
39	2012	Iceland	4,486	4,859	4,880	4,720	4,543	4,423	4,339	4,340	4,187	
40	2012	Liechtenste	393	333	411	353	366	365	387	387	349	
41	2012	Norway	60,466	62,521	63,713	62,810	61,257	61,671	60,120	60,433	60,010	5
42	2012	Switzerland	78,426	81,663	80,196	79,522	78,288	77,429	77,450	77,458	76,270	7
43	2012	United Kind	808,517	797,560	787,737	788,447	779,341	758,633	735,842	714,696	697,812	68

3.1 Read in population data

Source: https://appsso.eurostat.ec.europa.eu/nui/show.do?dataset=demo_pjan&lang=en

```
Pop <- read_excel("Data/demo_pjan.xlsx",
          range = "A10:CZ510",
          na = ":")
dim(Pop)
```

```
## [1] 500 104
```

When we read this in, some rows are entirely `NA` values for population. It will be easier to `filter()` these out a after the population values are stacked in a single column.

3.2 Reformat for joining

To be able to join, we must be able to exactly match on each of our structural criteria: `Country` names, `Year` and `Age` categories.

3.2.1 Reshape to tidy

```
# check first and last age classes
# colnames(Pop)
```

```
Pop <-
  Pop %>%
  pivot_longer(`Less than 1 year`:`Unknown`,
               names_to = "AGE",
               values_to = "Population") %>%
  rename(Country = `GEO/AGE`) %>%
  # is.na() gives TRUE if a value is NA, FALSE otherwise,
  # ! negates this, so we keep all rows that are not NA
  filter(!is.na(Population))

glimpse(Pop)
```

```
## Rows: 39,423
## Columns: 4
## $ TIME       <chr> "2012", "2012", "2012", "2012", "2012", "2012", "2012", "20~
## $ Country    <chr> "Belgium", "Belgium", "Belgium", "Belgium", "Belgium", "Bel~
## $ AGE        <chr> "Less than 1 year", "1 year", "2 years", "3 years", "4 year~
## $ Population <dbl> 62808, 64238, 63685, 63823, 62920, 62733, 61007, 60190, 588~
```

3.2.2 Recode Age classes

What age classes to we have?

```
Pop %>%
  pull(AGE) %>%
  unique()
```

```
##   [1] "Less than 1 year"     "1 year"               "2 years"
##   [4] "3 years"              "4 years"              "5 years"
##   [7] "6 years"              "7 years"              "8 years"
##  [10] "9 years"              "10 years"             "11 years"
##  [13] "12 years"             "13 years"             "14 years"
##  [16] "15 years"             "16 years"             "17 years"
##  [19] "18 years"             "19 years"             "20 years"
##  [22] "21 years"             "22 years"             "23 years"
##  [25] "24 years"             "25 years"             "26 years"
##  [28] "27 years"             "28 years"             "29 years"
##  [31] "30 years"             "31 years"             "32 years"
##  [34] "33 years"             "34 years"             "35 years"
##  [37] "36 years"             "37 years"             "38 years"
##  [40] "39 years"             "40 years"             "41 years"
##  [43] "42 years"             "43 years"             "44 years"
##  [46] "45 years"             "46 years"             "47 years"
##  [49] "48 years"             "49 years"             "50 years"
##  [52] "51 years"             "52 years"             "53 years"
##  [55] "54 years"             "55 years"             "56 years"
##  [58] "57 years"             "58 years"             "59 years"
##  [61] "60 years"             "61 years"             "62 years"
##  [64] "63 years"             "64 years"             "65 years"
##  [67] "66 years"             "67 years"             "68 years"
##  [70] "69 years"             "70 years"             "71 years"
##  [73] "72 years"             "73 years"             "74 years"
##  [76] "75 years"             "76 years"             "77 years"
```

```
##  [79] "78 years"               "79 years"               "80 years"
##  [82] "81 years"               "82 years"               "83 years"
##  [85] "84 years"               "85 years"               "86 years"
##  [88] "87 years"               "88 years"               "89 years"
##  [91] "90 years"               "91 years"               "92 years"
##  [94] "93 years"               "94 years"               "95 years"
##  [97] "96 years"               "97 years"               "98 years"
## [100] "99 years"               "Open-ended age class" "Unknown"
```

Now what would be the easiest way to code this to integer? I'd say: we have special cases for ages 0, 1, the open age group (100) and unknown ages. Every other age follows an exact pattern. Therefore, I propose we treat this with `case_when()` handling all special cases first, then doing some sort of string operation to handle all other cases that follow the pattern `"n years"`. This latter operation could either be a string operation that extracts digits, or a string substitution that deletes `" years"`.

Check:

```
a <- c("10 years","11 years")
# standard
a %>% gsub(pattern = " years", replacement = "") %>% as.integer()
```

```
## [1] 10 11
```

```
# terse regular expression
a %>% gsub(pattern = "([0-9]+).*$", replacement = "\\1") %>% as.integer()
```

```
## [1] 10 11
```

```
# or a handy helper function from the readr package:
parse_number(a)
```

```
## [1] 10 11
```

Any of these checks would work to handle "everything else" at the end of our `case_when()`

```
Pop <- # overwrite the same object here
  Pop %>%
  mutate(Age = case_when(
    AGE == "Less than 1 year" ~ 0,
    AGE == "Open-ended age class" ~ 100,
    AGE == "Unknown" ~ NA_real_,
    TRUE ~ parse_number(AGE)
  ),
  Year = as.integer(TIME)) %>%
  select(-TIME, -AGE)
```

3.2.3 Redistribute unknown ages

Here, rather than rescaling to the stated total as we did for `Births`, we take the other formula that applies the same principle, but framed in terms of redistributing counts with unknown age:

$$\hat{P_x} = P_x + \frac{P_x}{\sum P_x} * P_{Unkown}$$

where the denominator excludes P_{Unkown}. Once again, this operation is done inside `mutate()`. Note, we're using `is.na()` three different times as a logical selector! Here, `if_else()` is used rather than `case_when()` because there is only one condition and it is faster to type out.

```
Pop <-
  Pop %>%
  # declare groups
  group_by(Country, Year) %>%
  # 1. move Unknown age up to column
  # 2. replace NAs w 0s in the new UNK column
  mutate(UNK = Population[is.na(Age)],
         UNK = if_else(is.na(UNK), 0, UNK)) %>%
  # remove rows with Unknown age
  filter(!is.na(Age)) %>%
  # do the redistribution
  mutate(Population = Population + Population / sum(Population) * UNK) %>%
  # remove groups
  ungroup() %>%
  # remove column no longer needed
  select(-UNK)
```

3.3 Calculate exposures

Probably we'd rather join exposures to `Births` than January 1st population counts. One final calculation will allow us to introduce a join operation. The approximation we'd like to do is:

$$Exposure_x = \frac{P_x^{Jan1} + P_x^{Dec31}}{2}$$

In other words, just take the simple average of the population at the start and end of the year. We can approximate the end-of-year population using the following year's January 1st population. Our goal is to do this arithmetic like so `mutate(Exposure = (P1 + P2) / 2)`, so the trick is to create a second `Population` column, consisting in the same `Population` column we already have, but back-dated one year.

We will do this in two ways, first using joins. To do this we create a copy of `Pop`, then deduct `Year` by one in that copy, then merge it back to the original `Pop` that we started with. In the process we'll also rename both versions of `Population` to `P1` and `P2` so that we don't get confused. The year-range for `P2` will lose the most recent year, and it will also have one extra year on the lower end, due to the shift. When we **join** the objects together we want to do so only where we have overlapping combinations of `Year` (and `Age` and `Country` need to match too, but these will match exactly in our case).

3.3.1 joining datasets

There are different kinds of joining. Joins have a *left* and *right* side data object. Here are the basic ones, with some example data to make concepts clear:

```
x <- tibble(A = c("a", "b", "c"),
            B = c("t", "u", "v"),
            C = 1:3)
y <- tibble(A = c("a","b","d"),
            B = c("t","u","w"),
            D = 3:1)
x
```

```
## # A tibble: 3 x 3
```

```
##   A     B         C
##   <chr> <chr> <int>
## 1 a     t         1
## 2 b     u         2
## 3 c     v         3
```

```
y
```

```
## # A tibble: 3 x 3
##   A     B         D
##   <chr> <chr> <int>
## 1 a     t         3
## 2 b     u         2
## 3 d     w         1
```

1. `left_join()` the left object is primary and the right object is secondary. (left side row count unchanged, but right side could grow or shrink)

```
left_join(x,y)
```

```
## Joining, by = c("A", "B")
```

```
## # A tibble: 3 x 4
##   A     B         C     D
##   <chr> <chr> <int> <int>
## 1 a     t         1     3
## 2 b     u         2     2
## 3 c     v         3    NA
```

2. `right_join()` the right object is primary and the left object is secondary. (right side row count unchanged, but left side could grow or shrink)

```
right_join(x,y)
```

```
## Joining, by = c("A", "B")
```

```
## # A tibble: 3 x 4
##   A     B         C     D
##   <chr> <chr> <int> <int>
## 1 a     t         1     3
## 2 b     u         2     2
## 3 d     w        NA     1
```

3. `inner_join()` only keep combinations present in both the left and right. (row count can stay same or shrink)

```
inner_join(x,y)
```

```
## Joining, by = c("A", "B")
```

```
## # A tibble: 2 x 4
##   A     B         C     D
##   <chr> <chr> <int> <int>
## 1 a     t         1     3
## 2 b     u         2     2
```

4. `full_join()` keep all combinations (row count can stay same or grow)

```
full_join(x,y)
```

```
## Joining, by = c("A", "B")

## # A tibble: 4 x 4
##   A     B         C     D
##   <chr> <chr> <int> <int>
## 1 a     t         1     3
## 2 b     u         2     2
## 3 c     v         3    NA
## 4 d     w        NA     1
```

You see in each of these examples that we're politely told in the console which variables were used to determine structural combinations? In these examples, it made good default choices, but in general, we should specify which columns to consider, using the `by` argument:

```
left_join(x, y, by = c("A", "B"))
right_join(x, y, by = c("A", "B"))
inner_join(x, y, by = c("A", "B"))
full_join(x, y, by = c("A", "B"))
```

In our case, we want `inner_join(by = c("Country", "Year", "Age"))`, make sense?

There are other kinds of joining too! Check out this Rstudio cheat sheet for data reshaping possibilities with `dplyr`: https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf Here we're interested in the section called `Combine Tables` on page 2. These cheat sheets are pure gold when you're trying to think through something like this. Now, back to our beloved pipeline!

```
Exp <- Pop %>%
  # jan 1 of this year = dec 31 of last year
  mutate(Year = Year - 1) %>%
  # back-dated, this becomes P2 of the previous year
  rename(P2 = Population) %>%
  # Join together where Year overlaps
  inner_join(Pop, by = c("Country", "Year", "Age")) %>%
  rename(P1 = Population) %>%
  # Exposure calc averaging within-age
  mutate(Exposure = (P1 + P2) / 2) %>%
  # sort
  arrange(Country, Year, Age)
```

3.3.2 Alternative way to compute exposure

We could also reframe the calculation of exposure as a moving average. Here the functions `lead()` and `lag()` are often helpful:

```
a <- 1:10
lead(a)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 NA
```

```
lag(a)
```

```
##  [1] NA  1  2  3  4  5  6  7  8  9
```

```r
# mimics our objective calculation:
(a + lead(a)) / 2
```

```
##  [1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5  NA
```

The trick is to sort the data as needed (time series within `Country` and `Age`), then declare discrete groups on `Age` and `Country`. The result is identical, and this approach is more parsimonius than the join approach demonstrated above.

```r
Pop %>%
  arrange(Country, Age, Year) %>%
  group_by(Country, Age) %>%
  mutate(Exposure = (Population + lead(Population)) / 2) %>%
  ungroup() %>%
  filter(!is.na(Exposure)) %>%
  arrange(Country, Year, Age)
```

```
## # A tibble: 34,214 x 5
##    Country Population   Age  Year Exposure
##    <chr>        <dbl> <dbl> <int>    <dbl>
##  1 Albania      16436     0  2012   16674.
##  2 Albania      16479     1  2012   16285
##  3 Albania      16074     2  2012   16146.
##  4 Albania      16709     3  2012   16302
##  5 Albania      16100     4  2012   16324
##  6 Albania      16513     5  2012   16240
##  7 Albania      18119     6  2012   17253
##  8 Albania      19056     7  2012   18529
##  9 Albania      19102     8  2012   19007
## 10 Albania      21091     9  2012   20036
## # ... with 34,204 more rows
## # i Use `print(n = ...)` to see more rows
```

3.4 Bring it all together

The above steps accentuate how designing a processing pipeline happens in stages, and sometimes needs to be mapped out in advance. It's inherently iterative, and often involves trial and error, especially when we are learning. When doing this sort of work, we always check the results as we go to ensure things are processing as expected. When complete, we can clean up everything into a parsimonious pipeline. This allows you (and others) to think through all the steps in a glance: because tidyverse verbs string together into sentences! We therefore now paste all the above `Pop` processing code into a minimal pipeline, duly annotated:

```r
Pop <- read_excel("Data/demo_pjan.xlsx",
           # cell range from visual inspection
           range = "A10:CZ510",
           # NA character also from visual inspection
           na = ":") %>%
  # stack ages
  pivot_longer(`Less than 1 year`:`Unknown`,
               names_to = "Age",
               values_to = "Population") %>%
  filter(!is.na(Population)) %>%
  rename(Country = `GEO/AGE`) %>%
```

```
  # recode age
  mutate(Age = case_when(
    Age == "Less than 1 year" ~ 0,
    Age == "Open-ended age class" ~ 100,
    Age == "Unknown" ~ NA_real_,
    TRUE ~ parse_number(Age)
  ),
  Year = as.integer(TIME)) %>%
  select(-TIME) %>%
  # Begin redistribution of pop with unknown age
  group_by(Country, Year) %>%
  mutate(UNK = Population[is.na(Age)],
         # Not each Country / Year has an Unknown age category
         UNK = ifelse(is.na(UNK), 0, UNK)) %>%
  filter(!is.na(Age)) %>%
  # The redistribution (only affects some subsets)
  mutate(Population = Population + Population / sum(Population, na.rm = TRUE) * UNK) %>%
  select(-UNK) %>%
  ungroup() %>%
  # set up moving avg version of exposure calcs
  arrange(Country, Age, Year) %>%
  # time-series within age and country
  group_by(Country, Age) %>%
  mutate(Exposure = (Population + lead(Population)) / 2) %>%
  ungroup() %>%
  filter(!is.na(Exposure)) %>%
  arrange(Country, Year, Age)
```

```
## Warning: 783 parsing failures.
## row col expected               actual
## 101  -- a number Open-ended age class
## 102  -- a number Unknown
## 203  -- a number Open-ended age class
## 204  -- a number Unknown
## 305  -- a number Open-ended age class
## ... ... ........ ....................
## See problems(...) for more details.
```

See how this pipeline is in a single piece? This is because we avoided the self-join of the exposure calculation by using the `lead()` trick.

4 Work with merged data

4.1 Join `Pop` and `Births`

Note `Pop` has more `Year` (2012-2019), `Age` (0-100), and `Country` (41) values than does `Births`. However, `Births` has one year that `Pop` does not (2011). If we did `left_join(Pop, Births)` that would be clearly too much. If we did `left_join(Births, Pop)` then we'd be closer, but still have an extra year (2011) with no exposure available. Either of these (and by extension a `full_join()`) would still work, but would require extra `filter()` operations in order to get the data down to just the valid combinations of `Country`, `Year`, and `Age`. Hence, we use `inner_join()` again to create our new object, `Dat`.

```
Dat <-
  Births %>%
  inner_join(Pop, by = c("Country", "Year","Age"))
```

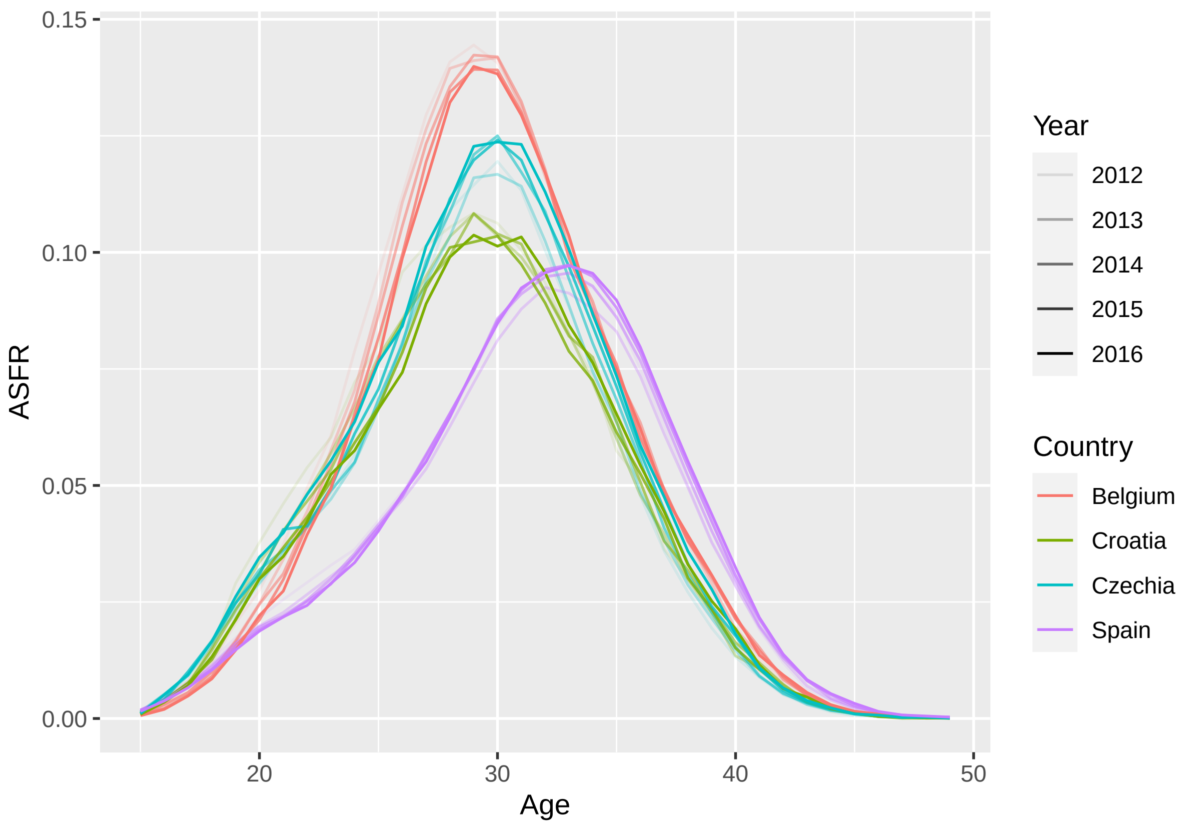
4.2 Calculate rates

Rate calculation is a straightforward `mutate()` statement. There is no need to apply groups, as age-specific fertility is done row-wise.

```
Dat <-
  Dat %>%
  mutate(ASFR = Births / Exposure)
```

Now a brief detour to examine the fertility curves and do a quick sanity check that TFR is as expected. An explanation of the `ggplot` code: everything inside `aes()` is a *mapping* of our data to coordinate or aesthetic properties. Since `ggplots` are composed of additive layers, we can keep adding layers using `+`. `geom_line()` is the geometric form that that mapping is translated to. Other geometric mappings are also possible. We'll breeze through several other low-key `ggplot` examples before more explicitly explaining things later in the workshop.

```
Dat %>%
  ggplot(aes(x = Age,
             y = ASFR,
             group = interaction(Country, Year),
             color = Country,
             alpha = Year)) +
  geom_line()
```



I can't visually integrate those curves, can you? So let's just do a quick check of TFR:

```r
Dat %>%
  group_by(Country, Year) %>%
  summarize(TFR = sum(ASFR),
            .groups = "drop") %>%
  pivot_wider(names_from = Country, values_from = TFR)
```

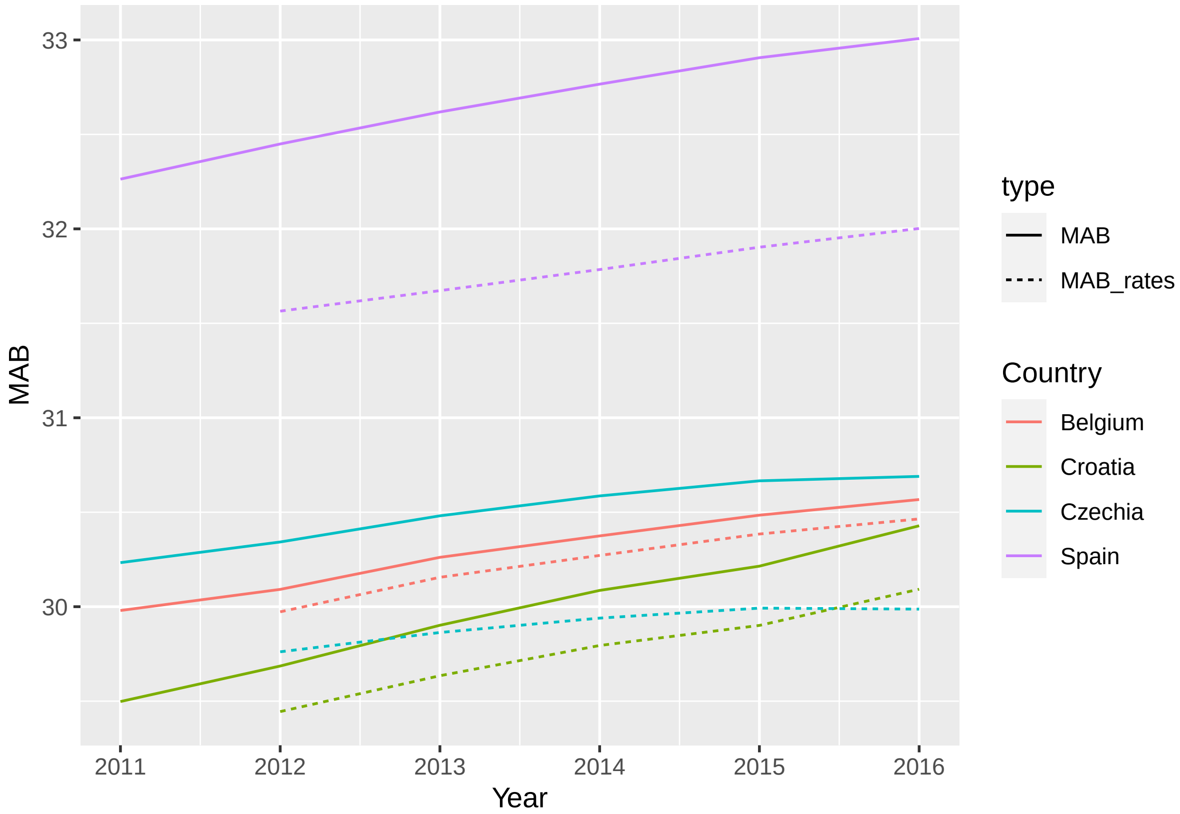
```
## # A tibble: 5 x 5
##    Year Belgium Croatia Czechia Spain
##   <int>   <dbl>   <dbl>   <dbl> <dbl>
## 1  2012    1.80    1.51    1.45  1.32
## 2  2013    1.76    1.46    1.46  1.27
## 3  2014    1.74    1.46    1.53  1.32
## 4  2015    1.70    1.40    1.57  1.33
## 5  2016    1.68    1.42    1.63  1.34
```

Full disclosure: When setting up this exercise I at first downloaded Total population by `Country`, `Year`, and `Age`, and I literally didn't realize it until checking the TFRs. They were too small, so I re-downloaded denominators to be sure and that was the problem! Lesson: always do these side checks! If your script is cluttered with checks of this kind, then put them aside in a supplementary script.

4.3 Recalculate mean age at birth using rates

Now we can calculate the MAB using fertility rates rather than birth counts, which ought to reduce the effects of population structure.

```r
Dat %>%
  # age midpoint
  mutate(Age = Age + .5) %>%
  # independent groups
  group_by(Country, Year) %>%
  # weighted mean for MAB
  summarize(MAB_rates = sum(ASFR * Age) / sum(ASFR),
            .groups = "drop") %>%
  # join to previous estimate. We do full
  # because year range different, but we can plot everything
  full_join(MAB, by = c("Country","Year")) %>%
  # stack so we can plot together easier, mapping MAB types to `linetype`
  pivot_longer(c(MAB_rates,MAB),
               names_to = "type",
               values_to = "MAB") %>%
  # remove NAs from asfr-weighted MAB (no 2011 info)
  filter(!is.na(MAB)) %>%
  ggplot(aes(x = Year,
             y = MAB,
             linetype = type,
             color = Country,
             group = interaction(Country,type))) +
  geom_line()
```



From this we see that trends are mostly the same, but not levels, and sometimes slopes are different. One could easily imagine a situation in which `ASFR`-weighted MAB gives a different trend than `Birth`-weighted MAB. One senses Czechia is close this case. Certainly levels can be quite different, and any discrepancy is due to departures from non-uniformity in population structure, which is an odd but precise way of putting it. In this case, if birth-weighted MAB is higher than ASFR-weighted MAB, it means that the population structure has more weight in higher ages.

5 Excercises

I will offer exercise solutions in a separate file, and walk through these in the following day's lecture in case there is no time in the current session.

1. plot time trends of TFR by country.

2. Use the time-series trick `lead()` and/or `lag()` to calculate a variable `rt` as:

$$r(t) = \frac{MAB(t+1) - MAB(t-1)}{2}$$

Here $r(t)$ (in R call the new variable `rt`) approximates the average rate of change in the mean age at childbearing. We can use it to calculate an adjusted form of TFR, `TFR_adj` like so:

$$TFR_{adj}(t) = \frac{TFR(t)}{1 - r(t)}$$

The basic idea is that if fertility is undergoing postponement, then the TFR of today is an underestimate (Bongaarts and Feeney (1998)). Without over-thinking the theory, can you calculate a variable called `TFR_adj`, and compare it's (short) time trends with those of TFR in a plot?

3. Choose either rate-weighted MAB or birth-weighted MAB, but redo the calculations in terms of 5-year age groups. Assume that the average age within the interval is simply the midpoint (`Age + 2.5`). Does this change MAB very much?

References

Bongaarts, John, and Griffith Feeney. 1998. "On the Quantum and Tempo of Fertility." *Population and Development Review*, 271–91.