How to Use the xrfr Package

Anna Grytaas

2021-01-12

This R package was created to convert elemental composition x-ray fluorescence (XRF) data of filtrated water samples from kcps to µM. It was created with R version 4.0.3. For source code, vignette, and more information, please visit https://github.com/agryt/xrfr.

Problems? Issues can be reported at https://github.com/agryt/xrfr/issues and questions can be sent to grytaasanna@gmail.com.

The files

First, you have to make sure to have the right files, and that these are all in the same folder on your computer. There are three files you need for your analysis:

- A file with **raw data** from the XRF machine.
- A file with **information about the samples and project**, like which location the samples are from, whether they are blank samples, and the volume of the filtrated water.
- A file containing **information about the machine, filters and elements**, like crystal drift and calibrated constants.

The raw data file

This file is created by the XRF machine when you export your data from it. You should not edit this file unless there are errors in the sample names. This file will be a .txt file using comma (,) as a decimal mark and tabs to separate columns. It will contain a column named "**Sample**", a column named "**Date**", and several columns with results from the machine's analysis. The ones we are interested in in this case are those containing "(Int)" in the column name.

The project information file

This file should contain all relevant information that you wish to include about your data that is not from the XRF machine. There are several columns that must be included for the functions in this package to work:

- **Sample**: The name of your sample. Must be the same as in the raw data file. Any samples that do not have a matching name will not be included in the created file.
- **Filter_type**: The type of filter used when filtering. Can be PC, ANO, or GFF. This column must be filled out even if the same filter type was used on all samples because R needs to know which detection limits and calibrated constants to use.
- **Filter_size**: The pore size of the filter used. This column can be left empty if the size was the same in all filters, but the column must exist.

- **Filter_box_nr**: A way to differentiate between different filter boxes, for example by labelling the boxes "1", "2", "3", etc. Lets R know which box the filter came from so it knows which blank samples to apply when finding the net count in this sample.
- **Filter_blank**: In this column you will need to write "blank" for all your blank samples. The rest of the column can be empty.
- **Volume**: The volume of water that was filtered for each sample.

You may include as many more columns as you want, for example dates, depth, location or treatment, but these columns all need to be present in your project information file.

The base information file

This file must contain all the relevant information related to the XRF machine, filters, and elements. This is a file that will not change between experiments except when a new measurement of the crystal drift is added. The base info file must contain these columns:

- **Element**: All the elements measured by the machine. Should be written as "C", "O", "Mg", etc.
- **MolarW**: The molar weight of each element.
- **PC**, **ANO**, and **GFF**: The calibrated constant for each element for each of the three filter types.
- **DL_PC**, **DL_ANO**, and **DL_GFF**: The detection limit of each element for each of the three filter types.
- **Drift_2008**, **Drift_2010**, etc.: The crystal drift each year it was measured. 2008 was the first year this was measured on the XRF machine at UiB, and this value is used during the calculations, so Drift_2008 must be included.

New to R/RStudio?

Both R and Rstudio can be downloaded to your computer for free from the internet. R is available at https://cran.r-project.org/ and Rstudio at https://rstudio.com/products/rstudio/download/. R is a language and environment used for statistical computing and graphics, while RStudio is an integrated development environment for R. This means that RStudio uses the R language and environment, but also gives you the possibility to keep and come back to your code, easy access to for example data frames and plots, etc.

Getting started

Once you have both R and RStudio installed, open RStudio. At this point you need to tell R where to look for files and save new files to (this is your work directory). This can be done by manually setting the work directory using `setwd()` or *Session* → *Set Working Directory* → *Choose Directory...*, or by creating a project (recommended). To create a project you can click *File* → *New Project...* or the "R within a box"-icon with a plus in a green circle at the top left. You can use an existing folder (where your project files are saved) as a directory by clicking *Existing Directory*. If you choose to click *New Directory*, you will need to move the project files to this new folder. You then choose the project name and where to put it, click *Create Project*, and you are ready to start coding. Next time you want to work on this code, open the .Rproj file in your directory, and it will open exactly as you left it.

Need help?

It is very useful to know how to get help if you have problems in R/RStudio or there is something you don't understand. In the bottom right window in RStudio, where you can see the files in your directory, you can go to the tab "**Help**" and search for specific functions to learn how they work. This can also be accessed by running the code `help("name_of_function")`, for example `help("summarise")`. Some functions have further

information available in vignettes, which can be accessed via the code `vignette("name_of_function")`, for example `vignette("nest")`. Information about packages can be accessed by using the code `help(package = "name_of_package")`, for example `help(package = "ggplot2")`. This information and more is always available on the internet as well, and you can find it by searching for the function or package in your search engine. In addition, a lot of the popular packages have cheat sheets available at https://rstudio.com/resources/cheatsheets/. On https://stackoverflow.com/ you can browse through other people's questions or ask your own. There are also many guides available for free on how to use R/Rstudio, for example this one for people who are used to working in Excel that you may find useful: https://rstudio-conf-2020.github.io/r-for-excel/.

R packages

R packages are extensions to R that can be installed. There are many packages available, and some of the most popular ones are very useful to have. The group of packages called **tidyverse** is particularly useful and one you will likely need if you plan on handling data or creating graphics with R. It includes packages such as **dplyr** (to manipulate data), **tidyr** (to help create tidy data), and **ggplot2** (to create graphics). The package **readxl** is useful if you want to import Excel files (.xlsx or .xls). Many packages can be installed using the code `install.packages("name_of_package")`, for example `install.packages("tidyverse")`. Once you have installed a package, you need to let R know that you wish to use functions from it. This is done with the function `library()`, for example `library(dplyr)`. Loading the package needs to be done each time you have restarted R, including when you reopen a project.

Some tips to make your experience with R/Rstudio better

- To run a line of code from your R Script you press ctrl + enter (on Windows or Linux) or cmd + enter (on Mac). You can also press run at the top right in the toolbar of the R Script window.
- Using pipe operators makes your code more readable and tidier. They are written as `%>%`. The pipe operator is a part of the **magrittr** package, but are also used in the **dplyr** package that you will likely be using (by running `library(dplyr)` or `library(tidyverse)`). On Windows or Linux you can use ctrl + shift + M as a shortcut for the pipe, and on Mac this is cmd + shift + M. `y %>% x = x(y)`.
- Use # at the beginning of a line to create a comment. Comments are ignored by R when running the code. This way you can write comments explaining what a function does etc.

Using the xrfr package

Please see the xrfr vignette for more information about each function and its arguments! (`vignette("xrfr")`)

Installing the package

If it is your first time using it, you need to install the package before you can use it. To do this, you must use the function `install_github()` from the **remotes** package. This package is a part of the **devtools** group of packages. If you do not already have this installed, you need to install either **devtools** or the **remotes** package first, by using the `install.packages()` function (`install.packages("remotes")` or `install.packages("devtools")`). To install the xrfr package you then load the package with the `library()` function and run the code `install_github("agryt/xrfr")`.

Preparing your data

Before starting the analysis, make sure that your files are properly formatted with the right column names and matching sample names. Your code will not work and you will get warning or error messages if they are not.

Before you can use the **xrfr** functions, you must import your data to RStudio. How this is done depends on which format your files are saved in. Your raw data file is created as a .txt file by the XRF computer, so this is likely how yours will be saved. If so, using the code `read_delim("your_raw_data.txt", delim = "\t", locale = locale(decimal_mark = ","))` should work, as long as the **readr** package (or the whole **tidyverse** package) is loaded. Excel files can be imported using the `read_excel()` function from the **readxl** package, and CSV files can be imported using the `read_csv()` or `read_csv2()` functions from the **readr** package depending on which type of CSV file it is. If you are using other file formats, you can use your search engine to find out how to import them, as it is likely possible to do so.

Here is an example of how to import the files:

```
library(tidyverse) # tidyverse includes readr, which is needed here
library(readxl)

rawdata.df <- read_delim("xrf_rawdata.txt", delim = "\t", locale = locale(decimal_mark =
                                                                            ","))
projectinfo.df <- read_excel("xrf_projectinfo.xlsx")
baseinfo.df <- read_excel("xrf_setup.xlsx")
```

This will make three data frames appear in your environment, containing each of the three files. At this point it is a good idea to open them (by clicking on them in the environment or running the code `view(name_of_dataframe)`) and checking that they were imported correctly. If not, you will need to fix the code used to "read" your file.

Using the main functions

First, of course, you must load the package:

```
library(xrfr)
```

There are two main functions in the xrfr package: `readxrf()` and `convertxrf()`. You will need to use both to perform the calculations from kcps to µM. `readxrf()` takes your data frames, changes them to the format needed, and combines them into one data frame. This data frame will then be used further in the `convertxrf()` function, where the actual calculations are performed, and the necessary data from the base info file is included. `convertxrf()` also makes your dataset longer, as this is a more readable format for R for further analysis.

Here is an example of how to use these functions:

```
imported.df <- readxrf(raw_data = rawdata.df, project_info = projectinfo.df)
data.df <- convertxrf(imported_data = imported.df, base_info = baseinfo.df, year = "2019",
                      first_element = "C", last_element = "As")
```

You will need to run these functions one at a time, as there is information in the data frame created with `readxrf()` that is needed to run `convertxrf()` properly. The result of each function will be a new data frame.

To run `readxrf()` you only need to define the arguments "raw_data", which is the name of the data frame containing your raw data from the XRF-machine, and "project_info", the name of the data frame containing information about all your samples.

To run `convertxrf()` you will need to let R know the name of the data frame created with `readxrf()` ("imported_data"), the name of the data frame containing the base information data ("base_info"), and the year in which the crystal drift was measured closest to when your samples were taken ("year"). You will

also need to define the arguments "first_element" and "last_element". These are the names of the first and last columns in the data frame created with `readxrf()` that contain data in kcps. This can change, but will likely be "C" for "first_element" and "As" for "last_element".

The output of these two functions is a long data frame containing new columns showing the concentration and detection limit of each element for each sample. You will likely want to save this data frame as a file, as this is the file you will use when for example doing statistical analysis or creating graphics. Saving a data frame can be done like this:

```
write_csv(data.df, file = "my_data.csv")
```

Note that `write_csv()` is a function from the package **readr**, so you must have that or the entire **tidyverse** installed and loaded to use it. The data frame will then be saved as a CSV file named "my_data.csv" in your directory. If you wish to save in a different format, that is entirely possible (use your search engine to figure out how!).

Using the "after" functions

After you have your data file with your newly calculated concentrations, you may wish to not only continue working on your data, but also look at your data yourself. This is hard to do when the data is in the long format, so you will likely want to widen the data so each element has its own column.

You can widen your data using the `widen()` function, which also excludes blank samples and unnecessary columns:

```
wide.project.df <- widen(project_data = data.df)
```

You may also wish to not only widen your data, but also calculate means or only see the concentrations that are above the detection limits. This can be done through the `widen_means()` and `widen_above()` functions, respectively.

To calculate means you will need to specify the factor(s) you wish to calculate the means based on, for example you may use location or treatment as factors. You can have either one or two factors. The `widen_means()` function will also remove the blank samples and unnecessary columns. The code shows examples both with and without a second factor:

```
means.project1.df <- widen_means(project_data = data.df, first_factor = "Location")

means.project2.df <- widen_means(project_data = data.df, first_factor = "Location",
                                 second_factor = "Depth")
```

When excluding the concentrations below the detection limits, the blank samples and unnecessary columns are also removed:

```
above.project.df <- widen_above(project_data = data.df)
```

You can also use the function `widen_means_above()` to both calculate means and exclude those concentrations that are under the detection limits. Note that this function calculates means first, *then* excludes mean concentrations below the detection limits.

```
means.above.project1.df <- widen_means_above(project_data = data.df, first_factor =
                                             "Location")
means.above.project2.df <- widen_means_above(project_data = data.df, first_factor =
                                             "Location", second_factor = "Depth")
```

Further work with your files

These "after" functions were made to avoid writing several lines of code for functions that you will likely use regularly, but there are many more ways to transform your data, or even further analyse it or create graphics. If you wish to further handle or manipulate your data, I suggest looking into the packages of the **tidyverse**, particularly **dplyr** and **tidyr** and their functions. For making graphics, **ggplot2** (also a part of **tidyverse**) is extremely useful. If you wish to perform statistical analyses and are wondering how to move forward, UiB's key to univariate statistics (https://folk.uib.no/nzlkj/statkey/) has some examples of R code for simple statistical analyses, though this is quite old. However, the internet is very helpful, and you should be able to figure out which analysis you should perform and find relevant packages and/or examples easily by searching for it.

Good luck with your data analyses!

Raw code

If your **xrfr** functions do not work, you can try using the raw code shown here, inserting your own file names, data frame names, and column names where needed. Where you need to enter your own names are indicated with names in uppercase letters.

```
library(tidyverse)
library(readxl)

# these functions will vary based on which format your files are saved in, but an example:
YOURDATAFILE.df <- read_delim("RAW_DATA_FILE.txt", delim = "\t",
                              locale = locale(decimal_mark = ","))
YOURPROJECTINFOFILE.df <- read_excel("PROJECT_INFO_FILE.xlsx")
YOURBASEINFOFILE.df <- read_excel("BASE_INFO_FILE.xlsx")

# readxrf
datafile.df <- YOURDATAFILE.df %>%
    select(c(Sample, Date, contains("Int"))) %>%
    rename_all(str_remove, pattern = " .*")
projectfile.df <- inner_join(datafile.df, YOURINFOFILE.df, by = "Sample")
notinprojectfile.df <- anti_join(datafile.df, YOURINFOFILE.df, by = "Sample")
# notinprojectfile.df should have 0 rows if all samples match between your files

# convertxrf
filter_area <- 9.078935
pivotproject.df <- projectfile.df %>%
    pivot_longer(FIRSTELEMENT : LASTELEMENT,
                 names_to = "Element",
                 values_to = "Count")
mean.blanks.df <- pivotproject.df %>%
    filter(Filter_blank == "blank") %>%
    group_by(Filter_type, Filter_size, Filter_box_nr, Element) %>%
    summarise(mean_blank = mean(Count))
adjusted.for.blanks.df <- left_join(pivotproject.df, mean.blanks.df, by = c("Filter_type",
                                    "Filter_size", "Filter_box_nr", "Element")) %>%
    mutate(Net_count = Count - mean_blank)
basefile.df <- YOURBASEINFOFILE.df %>%
    pivot_longer(c(PC, ANO, GFF),
                 names_to = "Filter_type",
                 values_to = "Cal_const") %>%
```

```
    relocate(Filter_type)
joined.df <- left_join(adjusted.for.blanks.df, basefile.df, by = c("Filter_type",
                       "Element"))
calculations.df <- joined.df %>%
    mutate(Concentration = ((Net_count * Cal_const) * filter_area * (1000 / Volume) *
                              1000 * (Drift_2008 / Drift_YOURYEAR)) / MolarW)
detectionlimits.df <- basefile.df %>%
    select(DL_PC, DL_ANO, DL_GFF, Element) %>%
    pivot_longer(c(DL_PC, DL_ANO, DL_GFF),
                 names_to = "Filter_type",
                 values_to = "Detection_limit") %>%
    mutate(Filter_type = str_remove(Filter_type, "DL_"))
project.detectionlim.df <- left_join(calculations.df, detectionlimits.df, by =
                                       c("Filter_type", "Element"))
project.df <- project.detectionlim.df %>%
    distinct() %>%
    select(Sample : Element, Concentration, Detection_limit)

# widen
projectwide.df <- project.df %>%
    filter(!Filter_blank %in% "blank") %>%
    select(-Detection_limit) %>%
    pivot_wider(names_from = Element, values_from = Concentration)

# widen_above
projectabove.df <- project.df %>%
    filter(Concentration > Detection_limit) %>%
    filter(!Filter_blank %in% "blank") %>%
    select(-Detection_limit) %>%
    pivot_wider(names_from = Element, values_from = Concentration, id_cols = Sample)

# widen_means
# if you only have one factor, simply remove "SECONDFACTOR" wherever applicable
projectaverageabove.df <- project.df %>%
      filter(!Filter_blank %in% "blank") %>%
      group_by(FIRSTFACTOR, SECONDFACTOR, Element) %>%
      summarise_if(is.numeric, mean) %>%
      select(-Detection_limit) %>%
      select(FIRSTFACTOR, SECONDFACTOR, Volume, Element, Concentration) %>%
      pivot_wider(names_from = Element,
                  values_from = Concentration)

# widen_means_above
# if you only have one factor, simply remove "SECONDFACTOR" wherever applicable
projectaverageabove.df <- project.df %>%
      filter(!Filter_blank %in% "blank") %>%
      group_by(FIRSTFACTOR, SECONDFACTOR, Element) %>%
      summarise_if(is.numeric, mean) %>%
      filter(Concentration > Detection_limit) %>%
      select(-Detection_limit) %>%
      select(FIRSTFACTOR, SECONDFACTOR, Volume, Element, Concentration) %>%
      pivot_wider(names_from = Element,
                  values_from = Concentration)
```