How to Use the xrfr Package

Anna Grytaas

2021-05-10

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 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.

Brief instructions on using the xrfr package

1. Make sure you have the right files and that they are structured as described under "The files".
2. Install the package if you do not already have it: `install.packages("devtools")` (if you do not have this installed already), then `devtools::install_github("agryt/xrfr", build_vignettes = TRUE)`.
3. Import your data files to R using the appropriate functions (see "Preparing your data" if you are unsure of how to do this). If you are using .ssd files instead of the .txt file created by the XRF machine, see "Importing .ssd files".
4. Load the package: `library(xrfr)`.
5. Join your data frame containing raw data and the data frame with your project information to a new data frame:

```
DATA0.df <- readxrf(raw_data = YOURRAWDATA.df, project_info = YOURPROJECTINFO.df)
```

6. Convert the data from kcps to µM using the data frame with base information:

```
DATA.df <- convertxrf(imported_data = DATA0.df, base_info = YOURBASEINFO.df, year = "2019",
                      first_element = "C", last_element = "As")
```

year = year in which the crystal drift was measured closest to when your samples were measured, first_element = name of first column showing data in kcps in "DATA0.df", last_element = name of last column showing data in kcps in "DATA0.df".

7. OPTIONAL: Use one or more of the "after" functions:

- Widen your data frame:

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

- Widen and calculate means based on one or two factors:

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

Note that "first_factor" must be defined while "second_factor" is optional.

- Widen and exclude values below detection limits:

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

- Widen, calculate means based on one or two factors, and exclude mean values below detection limits:

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

Note that "first_factor" must be defined while "second_factor" is optional.

- View the mean values of blanks, calculated based on filter type, filter size, and filter box number:

```
blanks.df <- show_blanks(imported_data = DATA0.df, first_element = "C", last_element =
                           "As")
```

8. Save any of the data frames you wish to keep, for example as a CSV file: `write_csv(data.df, file = "my_data.csv")`.

More detailed information can be found under "Detailed instructions on using the xrfr package" and in the vignette (`vignette("xrfr")`).

The files

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 when the samples were collected, whether they are blank samples, and the volume of water filtrated.
- A file containing **information about the machine, filters and elements**, like crystal drift and calibrated constants.

These files must all be in the same folder on your computer.

The raw data file

This file is created by the computer connected to the XRF machine when you export your raw data. The computer creates 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 here are those containing "(Int)" in the column name. *You should not edit this file unless there are errors in the sample names.*

The project information file

This file should contain all relevant information about your data that is not from the XRF machine. These are the columns you *must* include:

- **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, as this information is used to apply the correct detection limits and calibrated constants.
- **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 it must exist.
- **Filter_box_nr**: A way to differentiate between different filter boxes, for example by labelling the boxes "1", "2", "3", etc. This information is necessary to apply the correct blank samples.
- **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 (in ml) that was filtered for each sample. Must be a number, do not include "ml" in the cells.

You may include as many additional columns as you want. Additional columns could for example show dates, depths, locations, or treatments.

The base information file

This file must contain all the relevant information related to the XRF machine, filters, and elements. This file is not specific for each experiment, and only needs editing when adding a new measurement of the crystal drift. 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. Drift_2008 must be included, as this value is used during calculations (it was the first measurement on the XRF machine at UiB).

Details and examples

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

If you are new to R/RStudio, see "New to R/RStudio?" first.

Installing the package

If it is your first time using this package, you need to install it before use. To do this, use the function `install_github()` from the **remotes** package. This is part of the **devtools** group of packages. If you do not already have either **devtools** or the **remotes** package, you must install this before installing the **xrfr** package:

```r
install.packages("remotes")
# or:
install.packages("devtools")

# then:
remotes::install_github("agryt/xrfr", build_vignettes = TRUE)
# or:
devtools::install_github("agryt/xrfr", build_vignettes = TRUE)
```

Preparing your data

Before using the **xrfr** functions, your files must be imported as data frames to RStudio. How this is done depends on which format your files are saved in. Unless you have changed the format of the raw data file created by the computer by the XRF machine, you can import that using the code `read_delim("YOURRAWDATAFILE.txt", delim = "\t", locale = locale(decimal_mark = ","))`. This function is part of the **readr** package from the **tidyverse**. 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, use your search engine to find out how to import them.

Here is an example of how to import a .txt file and two .xlsx files:

```r
library(tidyverse)
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 new data frames appear in your environment. At this point it is a good idea to open them (by clicking on them in the environment or running the code `view(NAMEOFDATAFRAME)`) and check that they were imported correctly.

Using the main functions

First you must load the package:

```r
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.

To run `readxrf()` you need to define the arguments "raw_data" (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). For example:

```
imported.df <- readxrf(raw_data = rawdata.df, project_info = projectinfo.df)
```

To run `convertxrf()` you need to define "imported_data" (the name of the data frame created with `readxrf()`), "base_info" (the name of the data frame containing the base information data), and "year" (the year in which the crystal drift was measured closest to when your samples were taken). 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 will likely be "C" for "first_element" and "As" for "last_element", but does sometimes change.

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

The output is a long data frame containing new columns showing the concentration and detection limit (adjusted based on volume) of each element for each sample. You should save this data frame as you should use this when doing statistical analysis, creating graphics, etc. Saving a data frame as a CSV file can be done like this:

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

`write_csv()` is a function from the package **readr**. In the example above, the data frame named "data.df" will be saved in the working directory as a CSV file named "my_data.csv". If you wish to save in a different format, that is entirely possible, just use your search engine!

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 calculate means or only see the concentrations that are above the detection limits, in addition to widening the data. 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 location or treatment. 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 use the function `widen_means_above()` to calculate means and exclude those mean concentrations that are under 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")
```

Lastly, you can use `show_blanks()` to create a dataframe showing only the mean values for the blank samples, in case you wish to see which values are used when calculating the net count in the `convertxrf()` function.

```
blanks.df <- show_blanks(imported_data = imported.df, first_element = "C", last_element =
                           "As")
```

Further work with your files

If you wish to further handle or manipulate your data, I suggest looking into the packages of the **tidyverse**, particularly **dplyr** and **tidyr**. For making graphics, **ggplot2** (also a part of **tidyverse**) is extremely useful. If you want to present your data as a table, there are many packages available to make nice tables. An overview of some of these are available at https://rfortherestofus.com/2019/11/how-to-make-beautiful-tables-in-r/. If you wish to perform statistical analyses there is also a lot of information and many packages available on the internet regarding statistical analyses and R. A cheat sheet for the most common statistical analysis functions in base R can be found at https://www.dummies.com/programming/r/statistical-analysis-with-r-for-dummies-cheat-sheet/.

Good luck with your data analyses!

Importing .ssd files

To import .ssd files into R, you will first need to download HxD (https://mh-nexus.de/en/hxd/), a free hex editor. You can then **open your .ssd file in HxD**, and you will be able to see both the hex values and the text it translates to. **Select all of the hex values** (ctrl + a/cmd + a, or right click → Select all), **copy**, and **paste into a text editor**. You can use the computer's notebook, Word, etc. **Save this document as a .txt file** in your working directory. This .txt file should then only contain the hex values, like this: "53 53 44 33 12 00 00 00 73 72 2D 63 61 6C 31 32 30 00 00 (. . .)"

Once you have prepared your text file, you can read it into R using `readLines()` and use the `transformssd()` function:

```
sample1 <- readLines("ssd_sample1.txt")
sample1.df <- transformssd(hex_data = sample1)
```

The output will be a dataframe showing the results from your sample in the same format as the .txt file created by the XRF computer, but only showing one sample.

Once you have done this to all your samples, you can combine all the dataframes into a larger dataframe containing the results from all samples. This can be done using the `rbind()` function:

```
all.samples.df <- rbind(sample1.df, sample2.df, sample3.df)
# include all your samples inside rbind
```

This new dataframe is then possible to use the `readxrf()` and `convertxrf()` functions on.

New to R/RStudio?

Both R and Rstudio can be downloaded 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 while also being easier to use, for example by giving the user easy access to data frames, plots, code, etc.

Getting started

Once you have both R and RStudio installed, open RStudio. At this point you need to give R a working directory: a place to look for files and save new files to. 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 click *File → New Project…* or the "R within a box"-icon with a plus sign 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 click *New Directory*, you will need to move the project files to this new folder. After choosing the type of directory you assign a project name, decide where to put it, and click *Create Project*. You are now ready to start writing code. Next time you want to work on this code, open the .Rproj file in your directory.

Need help?

There are many ways to access help when you have problems in R/RStudio:

- In the bottom right window in RStudio you can go to the tab "**Help**" and search for specific functions to learn how they work. The same information is 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")`.
- 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 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/.

Any information that can be accessed through functions in R will also be available on the internet and can be found by searching for the function or package in your search engine.

R packages

R packages are extensions to R that contain functions not in base R. 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 use 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 from CRAN using the code `install.packages("name_of_package")`, for example `install.packages("tidyverse")`. Once you have installed a package, you need to load it using the function `library()`, for example `library(dplyr)`. Loading the package needs to be done each time you restart R.

Some tips

- 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 tidier and easier to read. 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. 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. I recommend writing a lot of comments so you remember what the code does when coming back to it later.

Raw code

If your **xrfr** functions are not working, 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 is indicated with uppercase letters.

```r
library(tidyverse) # needed for all functions
library(readxl) # needed if you are importing excel files
library(sjmisc) # needed for transformssd()

# anything written in uppercase letters should be changed based on your own files

# 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
```

```r
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.adj.detectionlim.df <- project.detectionlim.df %>%
    mutate(Adjusted_detection_limit = (1000 / Volume) * Detection_limit) %>%
    select(-Detection_limit)
project.df <- project.adj.detectionlim.df %>%
    distinct() %>%
    select(Sample : Element, Concentration, Adjusted_detection_limit)

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

# widen_above
projectabove.df <- project.df %>%
    filter(Concentration > Adjusted_detection_limit) %>%
    filter(!Filter_blank %in% "blank") %>%
    select(-Adjusted_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(-Adjusted_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 > Adjusted_detection_limit) %>%
      select(-Adjusted_detection_limit) %>%
      select(FIRSTFACTOR, SECONDFACTOR, Volume, Element, Concentration) %>%
      pivot_wider(names_from = Element,
                  values_from = Concentration)

# show_blanks
pivotproject.df <- projectfile.df %>%
    pivot_longer(first_element : last_element,
                 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))
wide.mean.blanks.df <- mean.blanks.df %>%
    pivot_wider(names_from = Element, values_from = mean_blank)

# transformssd
string <- readLines("YOURDATA.txt")
string <- gsub(" ", "", string)
splitstring <- as.list(str_split(string, "2F416C676572474646"))
hex <- "2F416C676572474646"
splitstring.added <- mapply(paste0, splitstring, hex)
subsplit <- gsub(".*?(00)(.*?)(2F416C676572474646).*", "\\2", splitstring.added)
subsubsplit <- substr(subsplit, 1, 4)
substring <- paste(unlist(subsubsplit), collapse=’’)
raw.substring <- sapply(seq(1, nchar(substring), by=2), function(x) substr(substring, x,
text.substring <- rawToChar(as.raw(strtoi(raw.substring, 16L)))
list.extracted <- str_extract_all(text.substring, "[A-Z][a-z]|[A-Z//s]")
unlist.extracted <- unlist(list.extracted)
extracted.df <- as.data.frame(unlist.extracted)
colnames(extracted.df) <- "Element"
subsplit1 <- gsub(".*?(204B41)(.*?)(2A0020).*", "\\2", splitstring.added)
subsubsplit1 <- str_sub(subsplit1, start = -8)
subsubsplit1 <- subsubsplit1[-1]
values.df <- data.frame()
for (i in 1:length(subsubsplit1)) {
  a <- readBin(as.raw(strtoi(apply(matrix(strsplit(subsubsplit1,"")[[i]],2),2,paste,
                                   collapse=""), 16)), "double", size=4)
  values.df <- rbind(values.df, data.frame(t(a), row.names = NULL))
  }
colnames(values.df) <- "kcps"
elements.values.df <- cbind(extracted.df, values.df)
samplename <- substr(string, 17, 100) # assumes that the sample name has <41 characters
raw.samplename <- sapply(seq(1, nchar(samplename), by=2), function(x) substr(samplename,
                                                                             x, x+1))
text.samplename <- rawToChar(as.raw(strtoi(raw.samplename, 16L)))
element.values.name.df <- cbind(elements.values.df, Sample = text.samplename)
date <- substr(string, 337, 376)
raw.date <- sapply(seq(1, nchar(date), by=2), function(x) substr(date, x, x+1))
text.date <- rawToChar(as.raw(strtoi(raw.date, 16L)))
text.date <- gsub("-", ".", text.date)
if(str_contains(text.date, "Jan") == TRUE) {
  text.date <- gsub("Jan", "01", text.date)
} else if(str_contains(text.date, "Feb") == TRUE) {
  text.date <- gsub("Feb", "02", text.date)
} else if(str_contains(text.date, "Mar") == TRUE) {
  text.date <- gsub("Mar", "03", text.date)
} else if(str_contains(text.date, "Apr") == TRUE) {
  text.date <- gsub("Apr", "04", text.date)
} else if(str_contains(text.date, "Mai") == TRUE) {
  text.date <- gsub("Mai", "05", text.date)
} else if(str_contains(text.date, "Jun") == TRUE) {
  text.date <- gsub("Jun", "06", text.date)
} else if(str_contains(text.date, "Jul") == TRUE) {
  text.date <- gsub("Jul", "07", text.date)
} else if(str_contains(text.date, "Aug") == TRUE) {
  text.date <- gsub("Aug", "08", text.date)
} else if(str_contains(text.date, "Sep") == TRUE) {
  text.date <- gsub("Sep", "09", text.date)
} else if(str_contains(text.date, "Okt") == TRUE) {
  text.date <- gsub("Okt", "10", text.date)
} else if(str_contains(text.date, "Nov") == TRUE) {
  text.date <- gsub("Nov", "11", text.date)
} else {
  text.date <- gsub("Des", "12", text.date)
}
element.values.name.date.df <- cbind(element.values.name.df, Date = text.date)
element.values.name.date.df$Element <- paste0(element.values.name.date.df$Element, " (Int)")
element.values.name.date.df <- element.values.name.date.df %>%
  pivot_wider(names_from = Element, values_from = kcps)
```