rerollR user manual

I Introduction

The rerollR package is built to provide a way of propagating the intra-individual (or intra-object) uncertainties of measurements to the scale of population comparison tests (t-test, Wilcoxon rank-sum test, randomization tests). Although it has been originally designed to handle geochemical data (i.e. isotopic and elemental data), rerollR is very flexible in its inputs and output and can provide valuable insights in all fields. The principles of each function from this package are relatively simple (see sections III to V) and the package is designed to allow an easy and quick grasp on the tool, even for beginner R users. The functions have been designed to allow a flexible input structure that aim to match the most common data structures users may come with. Functions are also designed with a precise error and warning message system to guide the user if something happen to go wrong. For advanced users that would like to take a closer look to the source code of the package, function source codes are publicly available (see Github: https://github.com/a-hassler) and are annotated in detail to ease comprehension.

The v1 of rerollR is composed of 5 main functions. The main functions are split in two families: the ***re_*** functions and ***roll_*** function. The ***re_*** functions are the first layer by which the data are processed. There is 2 distinct ***re_*** functions (***re_gauss*** and ***re_boot***) which condition the way measurement uncertainties will be propagated to population comparison tests. The output of ***re_*** functions (a data frame) constitute the base input for ***roll_*** functions. Each of the 3 ***roll_*** functions, ***roll_ttest***, ***roll_wtest*** and ***roll_rtest***, takes data frames produced by re_ functions to perform series of either t-tests, Wilcoxon rank-sum test or randomization tests, respectively. The output of roll_ functions is a list containing a data frame of the p-values* of all the performed tests, a data frame containing the main metrics of the distribution these p-values*, an optional graph (generated through the ggplot2 package) showing the distribution of these p-values* (*: randomization tests, like within roll_rtest, do not produce exact p-values unless all possible combination of random assignment are explored, which is usually out of reach because of the computation time it would implies).

II Installation

Installing rerollR from GitHub

Guide copied and adapted from: https://kbroman.org/pkg_primer/pages/github.html

How do you install a package that’s sitting on GitHub?

1. First, you need to install the devtools package. You can do this from CRAN. Invoke R and then type the following in the R console:

   install.packages("devtools")

2. Then load the devtools package by typing the following in the R console:

   library(devtools)

3. Then install rerollR from GitHub by typing:

   install_github("a-hassler/rerollR")

4. You should now be able to load rerollR like any other R package using:

```
library(rerollR)
```

You are ready to go, you can start playing with rerollR functions and example datasets.

Installing rerollR from CRAN

The rerollR package is not yet available on CRAN.

III Example of application (Cat Pop)

The following section is a generic example to showcase rerollR logic, process and aims. You found yourself in the following situation. You've realized a number of measurements of a given metric on a selection of individuals (or object) identified to be part of two distinct populations, an identification that is independent from your measurements. It could be the body length of cats from 2 different local species or, for fellow geochemists, the δ^{13}C values from 2 groups of sedimentary rocks. For the sake of simplicity, we wills stick to the cat example in this section. Cats being liquid, as everyone knows, it is hard to measure their body length in precise and accurate manners. Thus, you decided to replicate the measurement 3 time for all 20 randomly selected cats from the 2 cat populations (Pop1 and Pop2, 10 cats each). You end up with the following dataset (the code to replicate this example is on GitHub as an rmd file: Cat_pop_example.rmd, and is also contained within the rerollR package as data frames used in the examples of the help section):

Table 1: Cat body length data (not aggregated)

Pop1		**Pop2**	
Body length	**ID**	**Body length**	**ID**
69.14806	1	78.43989	1
69.37075	1	80.27638	1
62.8614	1	69.70271	1
68.30448	2	77.12924	2
66.41746	2	60.09871	2
65.19096	2	80.8229	2
67.36588	3	60.18335	3
61.34667	3	65.19147	3
66.56992	3	82.66504	3
67.05065	4	75.29447	4
64.57742	4	69.48898	4
67.19112	4	70.89429	4
69.34672	5	60.93578	5
62.55429	5	84.3385	5
64.62293	5	70.79378	5
69.40015	6	83.93941	6
69.78226	6	82.19387	6
61.17487	6	75.99947	6
64.74997	7	84.27417	7
65.60333	7	75.47096	7
69.04031	7	68.33568	7

61.3871	8	68.66871	8
69.88892	8	69.96214	8
69.46668	8	79.61732	8
60.82438	9	60.97341	9
65.14212	9	78.71988	9
63.90203	9	76.93192	9
69.05738	10	64.28161	10
64.4697	10	66.5272	10
68.36004	10	72.86032	10

Which you can aggregate as follow: Where mean is the mean of the 3 body length measurements for each cat, sd the standard deviation and n the number of replicates (3 each).

Table 2: Cat body length data aggregated as the mean, the standard deviation (sd) and number of replicates per individual (n)

Pop1	**mean**	**sd**	**n**		**Pop2**	**mean**	**sd**	**n**
1	67.12674	3.695572	3		**1**	76.13966	5.649684	3
2	66.63763	1.568392	3		**2**	72.68362	11.054218	3
3	65.09416	3.269733	3		**3**	69.34662	11.802772	3
4	66.27306	1.47015	3		**4**	71.89258	3.028754	3
5	65.50798	3.481634	3		**5**	72.02268	11.74966	3
6	66.78576	4.862926	3		**6**	80.71092	4.172535	3
7	66.46454	2.271128	3		**7**	76.02693	7.983775	3
8	66.91423	4.79129	3		**8**	72.74939	5.98286	3
9	63.28951	2.223087	3		**9**	72.20841	9.770774	3
10	67.29571	2.472109	3		**10**	67.88971	4.448698	3

The data look like this (Figure 1):

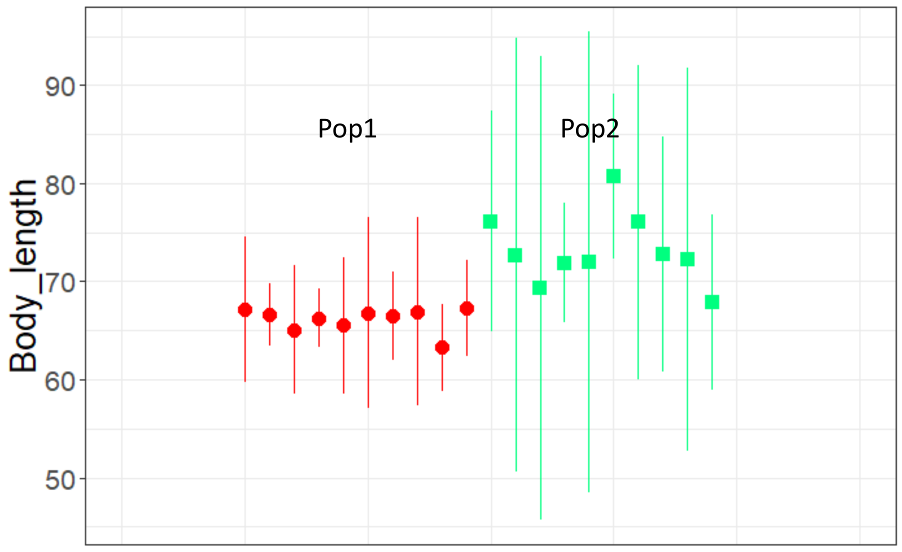


Figure 1: Cat body length. Pop1 in red and Pop2 in green. Error bars represent ±2sd.

If you perform a regular two sample t-test based on these data (the number of cats per population is a bit low for a t-test to be really accurate but this is not the point here), in order to have some insight on whereas these two populations are on average distinct in terms of body length, you end up with a p-value of about 0.0001 (0.000128 to be exact). A p-value this low tells that we can very confidently reject the null hypothesis that the average body length of these two species is equal, and thus accept the alternative hypothesis that their mean body length is different. An unambiguous result right? Well, maybe not. You may have been surprised to see how low is this p-value regarding how much overlap there is between the two populations when we consider error bars. The reason is the following, such a test (t-test, but also Wilcoxon-Mann-Whitney tests or randomization tests) do not compute the degree of uncertainty of measurement that we have for each individual (illustrated by the ±2sd error bars in figure 1). It is virtually like if the test was comparing the following data (Figure 2):

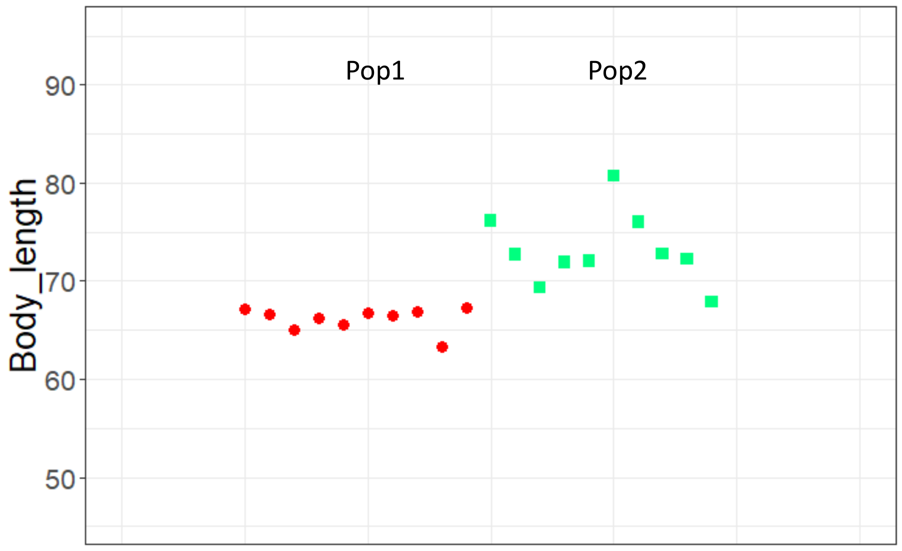


Figure 2: Cat body length. Pop1 in red and Pop2 in green. Error bars not displayed.

In a scenario like this, where intra-individual measurement uncertainties are important in regard to the global distribution of the data, it is no surprise that putting aside these uncertainties lead to surprising and misleading conclusions. It is the very purpose of rerollR to provide a solution for such situations.

The first step of rerollR is to generate new alternative populations based on the distribution of the original data. It the aim of the two re_ functions. For each individual, re_ functions iteratively generate a specified number of new data (n argument in the functions) that aim to represent the degree of measurement uncertainty recognized for each individual. To do this, re_gauss uses the function rnorm from the package stats. In the default configuration of re_gauss, rnorm will generate normally distributed data centered around the mean value (of cat body length in this example) of each individual, and dispersed according to the standard deviation (sd) of the individuals (Figure 3). re_boot does a similar job, but instead of relying on rnorm it performs a specified number of bootstrap within the individual data (boot.n argument) and compute the mean of those (Figure 3).

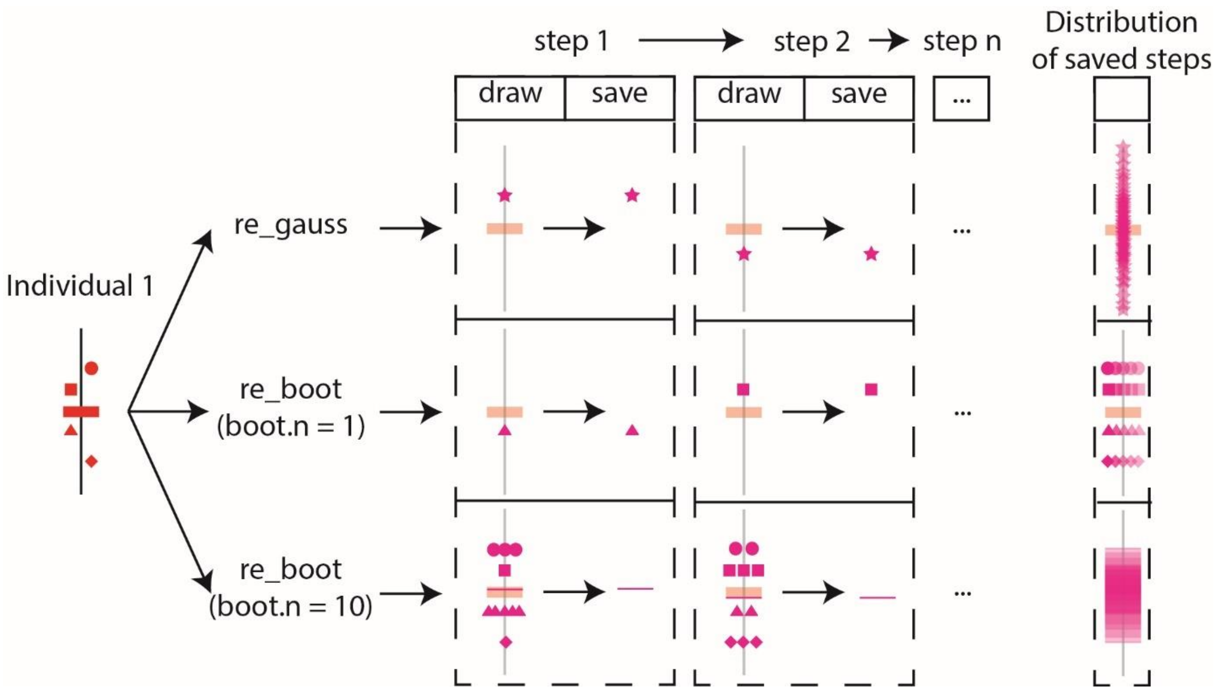


Figure 3: Example of re_ functions operations for each individual of a population in input. Horizontal bars are the mean of the original measurements (thick, red) or computed from bootstrapped values (fine, pink), with error bars representing ± 2 times the standard deviation (sd) of the original measurements. Circles, squares, triangles, and diamonds represent replicated measurements of individual 1. They are either the original measurements (red) or resampled values (pink). Pink stars represent values generated by the rnorm function (stats package) embed in re_gauss, with the mean and sd values of the original measurements as input. On the right, a summary of the distribution of all the saved values that would be displayed in the data frame generated by re_ functions after n steps ≤ 10 000.

After re_ functions have realised a specified number of steps for every individual of one population (e.g. n argument = 10 000), they return a data frame containing all the generated “alternative” measurements for all individuals. Each round of generation create a value for every individual, so if n = 10 000 in the re_ function call, and if the population contain 10 individuals, the output will be a 10 x 10 000 data frame. To illustrate that for our cat study, the table 3 is an example of output generated by re_gauss on the Pop1 population (*re_gauss(popm=Pop1_agr$mean, popu =Pop1_agr$sd, u = "SD", n=10000)*). Ideally, if intra-individual uncertainties are well constrained and the number of n step high enough, re_ function output should properly explore the space of these uncertainties. In the data frame, each column then represents an “alternative” pool of measurements from the population, a likely theoretical outcome if we had redone our measurements with slightly different odds. Like rerolling the dices of probabilities to see how different the outcome could have been in a parallel universe.

Table 3: Example of re_ output (re_gauss here) for the population Pop1 of cats. The complete data frame dimension is 10 x 10000.

individual\ step	**1**	**2**	...	**10 000**
1	70.48402	67.2774		60.14032
2	68.05159	65.17749		65.50516
3	62.44734	68.63527		70.9685
4	65.9052	67.41749		65.02204
5	70.90395	65.22877		67.1571
6	64.442	69.35187		66.67377
7	65.36295	65.12484		64.08655
8	67.86267	63.75148		63.30043
9	61.31853	64.12763		64.18003
10	66.40602	72.38383		62.7829

Going a bit further with our cat study we now have two re_ outputs (like Table 3), one data frame for each cat population, respectively called re_Pop1 and re_Pop2 for this example. Note it is important that the same re_ function and same parameters are used for the two populations to compare (same n notably). The next step is to call a roll_ function with these two "re_ made" data frame in input. The three options available are roll_ttest, roll_wtest and roll_rtest, to perform t-tests, Wilcoxon-Mann-Whitney tests (i.e. Wilcoxon rank-sum test) and randomization tests, respectively. For all these three functions, the designated test will be successively performed by taking columns in each re_ data frame (i.e. re_Pop1 and re_Pop2) and use their data as input for the test (Figure 4). For example, with roll_ttest, the function will first perform a t-test by comparing the first column of re_Pop1 with the first column of re_Pop2, save the p-value obtained from the test in a data frame, then proceed to compare the second columns of both re_Pop1 and re_Pop2 with a t-test, and so on until it reaches the last column of re_Pop1 and re_Pop2 (Figure 4). In the end, the function performed as many t-test as re_Pop1 and re_Pop2 have columns and saved all the resulting p-values in a one column data frame (Figure 4).

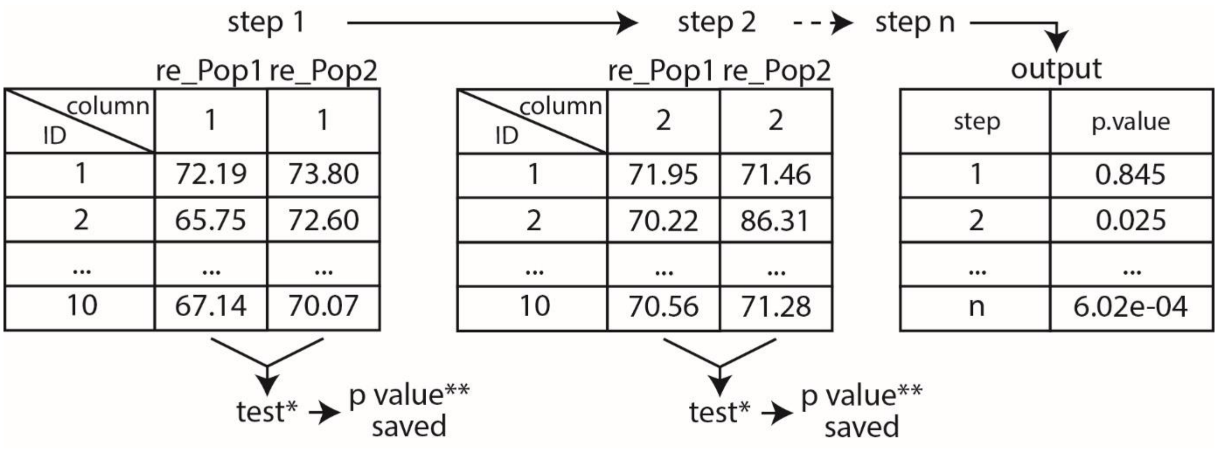
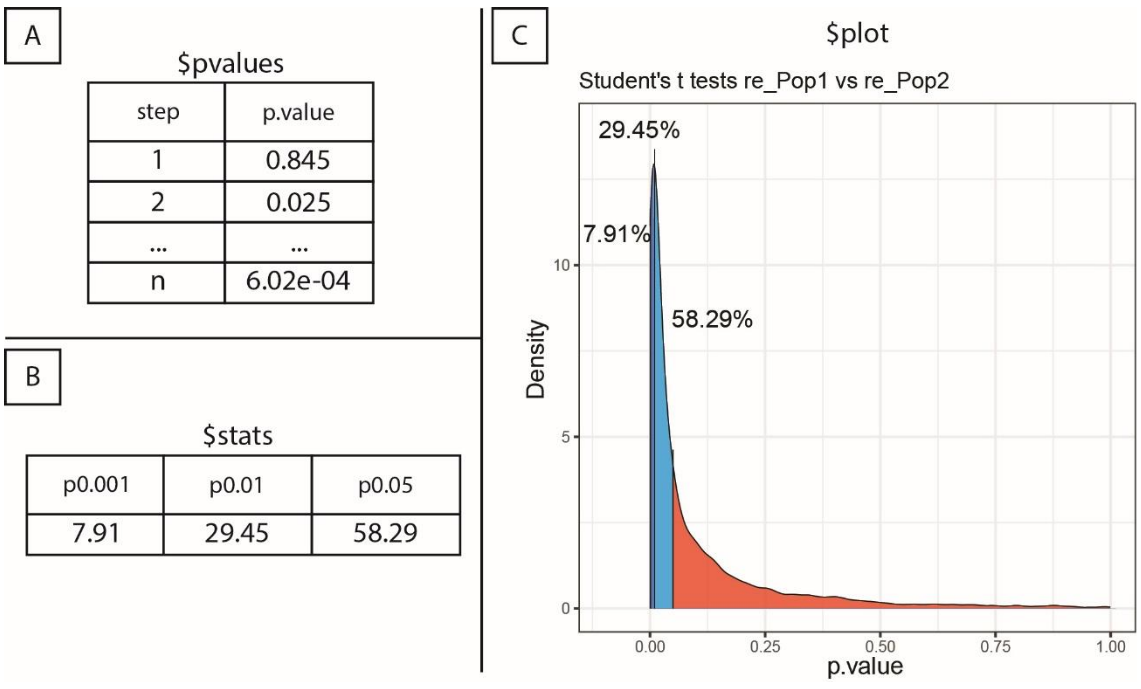


Figure 4: Example of roll_ functions operations (re_Pop1 and re_Pop2 are two data frames resulting from re_gauss runs on Pop1 and Pop2 cat populations, respectively). At each step, re_ functions select a column in each data frame, run a test between the two and save the resulting p-value** in a data frame. The total number of steps is equal to the number of columns of the re_ data frames in input. *: test can be a two sample t-test (roll_ttest), a Wilcoxon-Mann-Whitney test (roll_wtest) or a randomization test (roll_rtest). **: For the case of randomization tests, obtaining an exact p-value is often unpractical, so p-values are often only estimations for these tests.*

The data frame compiling all the p-values generated by roll_ functions is not the only output of these functions. The list of output that the functions return also contain a small additional data frame, and an optional plot generated within roll_ functions which use the package ggplot2. Both illustrate the distribution of the p-values generated by roll_ functions. The second data frame contain the percentage of generated p-values < 0.001, <0.01 and <0.05. The plot display the distribution of these p-values as a density, histogram or hybrid plot (both density and histogram), highlight the fractions of p-values < 0.001, between 0.001 and 0.01, between 0.01 and 0.05, and > 0.05. It also displays the results contained in the second data frame. An example of these possible outputs is displayed in Figure 5.


A

$pvalues

step	p.value
1	0.845
2	0.025
...	...
n	6.02e-04

B

$stats

p0.001	p0.01	p0.05
7.91	29.45	58.29

Figure 5: Example of output of a roll_ function (roll_ttest results for Pop1 and Pop2 cat body length data). A) Is a data frame of all the calculated p-values. B) Is a data frame with the percentage of generated p-values below 0.001, 0.01 and 0.05. C) Is a density plot of the generated p-values (minorly edited to accommodate the very wide range of p-values of the example). The three percentages correspond to those displayed in the data frame in B. With default colors like here, these categories of p-values take a darker blue color the smaller they are, while p-values above 0.05 are shown in red. (Note: the plot can also be a histogram alone or combined with the density plot).

Looking at roll_ttest results in Figure 5, it appears that the initial p-value obtained by doing a t-test on the individual mean of the original measurements only (p-value = 0.00128) is not representative at all of the p-values obtained when individual means are modified to explore the space of intra-individual uncertainties of measurement. Only 7.91% of these p values are smaller than 0.001, while 58.29% are smaller than 0.05. In other words, about 41,71% of the 10 000 configurations we tested in roll_ttest gave a p-value higher than 0.05. This suggest that rejecting the null hypothesis that the two populations have the same mean is not that strongly supported by the data when intra-individual uncertainties are considered. In this case, improving our method of cat body length measurement to reduce the intra-individual uncertainties of measurement, and maybe also improving our sampling efforts on these cat populations (i.e. increasing the number of individuals measured in each population), seems to be necessary to bring more conclusive evidence about the differences of mean body lengths that may exist or not between the two species. Note, that setting the t-test of roll_ttest to verify is Pop2 mean is greater than Pop1 (i.e. roll_ttest(p1=re_Pop1, p2= re_Pop2, density=T, histogram = F, alternative="less”)) gives more support to the hypothesis that Pop2 cat population may have a greater mean body length than Pop2 (p-values <= 0.05 = 71.07%), but still leaves an important fraction of configurations (28.93%) where the risk to reject the null hypothesis and being wrong is higher than 5%.

IV Basics and quick start examples

Running re_ functions

Which re_ function is better for me?

After having installed rerollR, the first question to ask yourself is which re_ function should you use. In short, re_gauss performs well when intra-individual measurement values are normally distributed, can better handle limited intra-individual replicates number than re_boot (by nature and through the use of the min.u argument), and is more flexible with the data it can takes in input. re_boot on the other hand can stay appropriate even when intra-individual measurement values are not normally distributed, but can underperform when the number of duplicated measurements per individual is small. Indeed, re_boot, only properly explore the space of intra-individual uncertainties if the measurements on individuals have been replicated enough to explore this space (note: it is true for re_gauss too but to a lesser degree). Notably, by nature bootstrap generated values never go beyond the maximum or below the minimum of the original measurements (see Figure 3), which can lead to an underestimation of the level of uncertainty.

Possible inputs

We can distinguish two forms of possible input for re_ functions. The “not-aggregated form” where all duplicated measurements are fully detailed (e.g. Table 1), and the “aggregated form” where duplicated measurements are describe through mean values and uncertainty (or dispersion) data like standard deviation, standard error, u uncertainties and derivatives (e.g. Table 2). A “not aggregated” data frame can be used both for re_gauss and re_boot. With such input it very easy to switch between re_gauss and re_boot (re_boot syntax is very close from re_gauss syntax in partitioned=T mode). However, an “aggregated“ data frame can serve as input for re_gauss function only, as bootstrapping is impossible from pre-aggregated data. Nevertheless, some advanced configurations of re_gauss are restrict its inputs to “aggregated“ data only. As it is generally easy to convert not-aggregated data to aggregated ones but not the other way around, users should favor to keep aggregated data when possible and be aware that only re_gauss will be able to process pre-aggregated ones.

re_boot

For a basic understanding of how re_boot process the data check section III and Figure 3. This section will give you the basic syntax knowledge to use this function in a example of simple configuration. For more detailed explanation check section V.

re_boot = function(data=NULL, popm, ID, n, boot.n=1)

Starting from a not-aggregated data frame from one population of individual/object you want to perform roll_ tests on is probably the most straightforward approach (e.g. like the Population 1 part of Table 1). In this data frame (called df for example), you need the values from the measurements you want to look at (called body_length for example) associated with an identifier (e.g. a series of names or ID numbers) that allow the function to identify what measurements belong to which individual/object from the population (called lab_name for example). You can fill the arguments in the two following ways:

- re_boot(data = df, popm = “body_length”, ID = “lab_name”, …)
- re_boot(popm = df$body_length, ID = df$lab_name, …)

data = NULL by default and has to be NULL if you want to give vector inputs (like df$body_length) for popm and ID arguments. If you provide vector inputs for popm and ID without calling them from the same data frame, be sure that both vectors match each other in length and order.

n and boot.n inputs need to be positive integers. n specifies the number of columns generated in re_boot data frame output (i.e. the number of generated “alternative population”). A n = 100 or 1000 can be useful for quick preliminary testing, a n ≥ 10 000 is generally advised to provide reliable results later on with roll_ functions (lengthier computation time). There is no general rule to what input you should choose for boot.n. To picture the effect of boot.n on the distribution of the results check Figure 3 and chose depending of the kind of intra-individual value distribution you wish to explore with re_boot. A higher boot.n increase the computation time of re_boot but do not affect the computation time of subsequent roll_ functions.

Here are generic examples of what simple re_boot calls should look like:

- re_boot(data = df, popm = “body_length”, ID = “lab_name”, n=1000, boot.n=1)
- re_boot(popm = df$body_length, ID = df$lab_name, n=1000, boot.n=1)

re_gauss

For a basic understanding of how re_gauss process the data check section III and Figure 3. This section will give you the basic syntax knowledge to use this function in a example of simple configuration.

re_gauss can handle a wider variety of inputs than re_boot. This section will focus on two basic configurations (i.e. simple not-aggraded and simple aggregated settings) to quickly get started with this function. For more detailed explanation about what options re_gauss can offer check section V.

re_gauss = function(data=NULL, popm, ID=NULL, popu=NULL, n, partitioned=F, u=NULL, u.conv=F, nm=NULL, na.popu=NULL, min.u=NULL)

Like for re_boot, data, popm, ID, but also popu argument inputs can either take the form of data = df while popm, ID or popu are column headers (e.g. popm = “body_length”), or data can be let to NULL while popm, ID or popu inputs are vectors (e.g. popm = df$body_length). data = NULL by default and has to be NULL if you want to give vector inputs (like df$body_length) for popm, ID or popu arguments. If you provide vector inputs for popm, ID or popu without calling them from the same data frame, be sure that vectors match each other in length and order.

If the data in input are not aggregated, set partitioned = T, then provide inputs for popm, ID and n, the rest can be let to default values for a basic run. popm input must contain the values from the measurements you want to look at (called body_length for this example), while ID input (e.g. a series of names or ID numbers) must attribute the measurements to the individuals/objects of the population (called lab_name for this example). n input needs to be a positive integer and specifies the number of columns generated in re_gauss data frame output (i.e. the number of generated “alternative population”). A n = 100 or 1000 can be useful for quick preliminary testing, a n ≥ 10 000 is generally advised to provide reliable results later on with roll_ functions (lengthier computation time).

Examples:

- re_boot(data = df, popm = “body_length”, ID = “lab_name”, n = 1000, partitioned=T)
- re_boot(popm = df$body_length, ID = df$lab_name, n = 1000, partitioned=T)

If the data in input are aggregated (i.e. partitioned = F, default), then provide inputs for popm, popu, n and u, the rest can be let to default values for a basic run. popm input must contain the mean values from the measurements you want to look at (called body_length for this example), while popu must contain the corresponding uncertainty or dispersion values of these means. For this simple example, the use of standard deviation values (called sd in this example) is recommended as this is the most straightforward approach to use re_gauss. The argument u specifies what is the nature of popu, it must be set to “SD” (for standard deviation) in this example to indicate that popu contains standard deviation values and not data of other nature. Alternatively, popu (and the corresponding u argument) can also be 2SD, SE (standard error), 2SE, as well as u and U uncertainty values. More about this in section V.

The last argument requiring input is n. A n = 100 or 1000 can be useful for quick preliminary testing, a n ≥ 10 000 is generally advised to provide reliable results later on with roll_ functions (lengthier computation time).

Examples:

- re_gauss(data=df, popm="body_length", popu="sd", n=1000, partitioned=F, u="SD")
- re_gauss(popm=df$body_length, popu= df$sd, n=1000, partitioned=F, u="SD")

Running roll_ functions

Generalities

For a basic understanding of what roll function do, check section III, notably Figures 4 and 5. The syntax of all roll_ functions is identical for the most part with only a few specificities, notably for roll_rtest:

- **roll_ttest = function(p1, p2, do.plot=T, density=T, histogram=T, fill.color=c("dodgerblue4","dodgerblue2","deepskyblue2","red"), ...)**
- **roll_wtest = function(p1, p2, do.plot=T, density=T, histogram=T, fill.color=c("dodgerblue4","dodgerblue2","deepskyblue2","red"), ...)**
- **roll_rtest = function(p1, p2, nr, metric="mean", do.plot=T, density=T, histogram=T, fill.color= c("dodgerblue4","dodgerblue2","deepskyblue2","red"))**

p1 and p2 inputs must be data frames generated by re_ functions based on the two populations you wish to compare using roll_ tests. For the result of roll_ tests to make sense, it is important that p1 and p2 have been generated by the same re_ function (either re_boot or re_gauss) using the same settings (notably for n and boot.n arguments).

Most of the other arguments can be let to their default input, but can be used to customize roll_ functions outputs. When do.plot = T (default), the function generates a plot using ggplot2 package. You can turn it off (do.plot = F) if you don't want such plot to be generated or don't want to use ggplot2. density and histogram arguments specify if the plot should be a density plot (density = T), a histogram (histogram = T) or a combination of the two (density = T, histogram = T). fill.color allow to specify the colors the p value areas between 0 to 0.001, 0.001 to 0.01, 0.01 to 0.05 and above 0.05 (in this order), it can be let to default colors.

roll_ttest and roll_wtest specifics

roll_ttest and roll_wtest respectively use the functions t.test and wilcox.test from the stats package. You can pass specific instructions to these embed functions by introducing matching arguments in roll_ calls. For example, you can pass the following instructions: paired = F (default) or T, alternative = "two.sided" (default) or "greater" or "less", etc. For "greater" or "less" configurations, note that x=p1 columns and y = p2 columns. In other words, if alternative="greater" the function will test if p1 is greater than p2 (iteratively, pairs of columns after pairs of columns, see Figure 4).

roll_rtest specifics

roll_rtest need inputs for two more arguments than the other roll_ functions. roll_rtest consist into making a randomization test for each matching pairs of columns from p1 and p2 (see Figure 4). The metric argument specifies the metric that will be compared by the randomization tests. Possible inputs are "mean" (default), "median" and "t". Respectively, randomization tests will either compare the absolute difference between the means of the population (metric = "mean", by default), the absolute difference between the medians of the populations (metric = "median") or the absolute difference between the t statistic of the populations (metric = "t").

The nr argument specifies the number of times the data will be randomly assigned to p1 and p2 and compared to the data of p1 and p2 matching columns with their original assignments (Figure 4). To perform reliably well, randomization tests commonly need to be based on a least several thousand of random assignments. This can be demanding on its own in terms of computation time, but here the aim of the rerollR package is to reproduce these tests several thousand times (i.e. on each p1 and p2 column pairs) in order to account for measurement uncertainties. This multiplies computation time by as much, up to unpractical limits in some configurations. For example, if one randomization (i.e. random assignment) of 20 data can take about 0.002 second on a regular laptop, it takes about 55 hours of calculation time (extrapolated) for nr = 10 000 with p1 and p2 being both 10 row x 10 000 col data frames. In consequence, such large-scale randomization tests should be consider a last resort option if t-test and Wilcoxon-Mann-Whitney test base conditions cannot be met. Smaller nr (e.g. 100 or 1000) allow for quicker experiments but dimmish the reliability of randomization tests. In the future, implementing parallel computation in this function could allow using more cpu power and decrease the overall computation time (R is single-threaded y default).

Additional note: When nr is small (e.g. <= 100) density plot can take unusually scattered gaussian shapes when 100% of randomization test results are identical (which is easy to obtain with small number of random assignments). Solution: increase nr or plot with histogram = T and density = F.

Examples

Here are generic examples of what simple roll_ function calls could look like with most settings let to default. In these examples, re_p1 and re_p2 are data frames that would result from re_ functions:

- roll_ttest = function(p1=re_Pop1, p2=re_p2)
- roll_wtest = function(p1=re_Pop1, p2=re_p2)
- roll_rtest = function(p1=re_Pop1, p2=re_p2, nr = 200)

General tips

- In re_ functions, bigger n means longer computation time for re_ functions and subsequent roll_ functions. On a basic laptop a re_ function call for a population of 10 individual, a n = 1000 takes seconds, a n=10 000 takes a few minutes, a n=100 000 takes few hours.
- If the plot produced by roll_ functions is not to your taste and that you would like to change more than what the argument of the function allows (e.g. histogram vs density, colors), there is actually a fairly easy trick to make global changes to the plot like changes of axis limits and plot labels for example. Save roll_ function output as an object (e.g. roll_results <- roll_ttest(…)) or saves the plot directly (e.g. roll_plot <- roll_ttest(…)$plot), then call the object (or the plot) plus what you want to modify with the ggplot2 syntax (e.g. to change labels and x axis limits: roll_results$plot + labs(title= "Student's t tests Pop1 - Pop2")+ scale_x_continuous(limits=c(-0.1,1)).
- If the plot you would like to produce is too far away from the plot that roll_ functions produce, or if you are looking for specific information regarding p-value distributions that roll_ functions do not provide directly, the data frame containing all the p-values from the function is easy to extract and use for your own plot or search functions. Simply save the output of roll_ functions as an object (e.g. roll_results <- roll_ttest(…)) and call the data frame named “pvalues” from within the resulting list (e.g. roll_results$pvalues). You can then easily plot or dig in these data like you would do with any similar data frame (reminder, this will be a one column data frame, not a vector).
- If you want to avoid saving re_ output data frames as object before using roll_ functions, note that you can insert re_ functions directly in roll_ functions as p1 and p2 inputs, but note that the code is less straightforward to read this way and that running such roll_ function takes more time as is include re_ functions too. Example:

roll_ttest(p1 = re_gauss(data=df1, popm=”body_length”, popu=”sd”, n=1000, u=”SD”), p2 = re_gauss(data=df2, popm=”body_length”, popu=”sd”, n=1000, u=”SD”))

Result randomness and consistency

re_gauss, re_boot and roll_rtest all involve randomness. Use set.seed before the function call to obtain constant results (e.g. for preliminary testing) but remove it for actual analyses.

V In depth function description

re_boot

The function takes a population of objects/individuals known through replicated measurements of a given metric. The function makes random sampling with replacement (bootstrapping) within each object/individual data in order to produce alternative populations.

(output: a data frame containing all created alternative populations, can serve as a input for roll_ functions)

re_boot(data=NULL, popm, ID, n, boot.n=1)

- data (possible input: NULL or a data frame).

  The input must be a data frame containing the population metric measurements (identified by the popm argument) and an identifier (a vector or column header) allowing to identify the objects/individuals composing the population (identified by the ID argument). Data argument input must be NULL (default) if popm and ID inputs are vectors (e.g. data = NULL, popm = df$result, ID = df$column_name) and should be a data frame if popm and ID inputs are column headers of this data frame (e.g. data = df, popm = 'result', ID = 'column_name').

- popm (possible input: a column header from the data argument or a vector).

  The vector or column must contain all replicated measurements of each object/individual of the population. These data are the one that will be iteratively bootstrapped. NA values within popm input are omitted for the bootstrapping. The column or vector length of popm must match the one of ID and be ordered in the exact same way.

- ID (possible input: NULL, a column header from the data argument or a vector).

  The vector or column must contain PERFECTLY MATCHING identifiers (e.g. character strings, ID numbers) regarding popm input, to allow the function to identify objects/individuals within the population. The column or vector length must match the one of popm and be ordered in the exact same way.

- n (possible input: a positive integer).

  Specifies the number of alternative populations generated by the function and stored in the data frame in output. A n >= 10 000 is generally advised for reliable subsequent roll_ testing, while n=100 or 1000 can be useful for quick preliminary testing.

- boot.n (possible input: a positive integer).

  Specifies the number of random sampling with replacement performed within popm data for each of the object/individual and each generation of alternative population. With boot.n = 1 (default) the function takes only one measurement per object/individual for each generation of alternative population. At boot.n >= 2, each object/individual of each generated population become the mean of boot.n numbers of bootstrap sampling from the original object/individual measurement replicates.

re_gauss

The function takes a population of objects/individuals known through replicated measurements of a given metric. Based on the values and distribution of the measurements of each object/individual of this population (uncertainties either provided in input or computed by the function), the function generates new data for each object/individual of the population, using a Gaussian random number generation centered on the mean of each object/individual and conditioned by the dispersion of the replicated measurements around their means (using the function rnorm from the package stats). Each round of data generation represents an alternative population (same number of objects/individuals with different values than the original population), and rounds are successively added as columns (one per iteration) in a data frame.

(output: a data frame containing all created alternative populations, can serve as a input for roll_ functions)

re_gauss(data=NULL, popm, ID=NULL, popu=NULL, n, partitioned=F, u=NULL, u.conv=F, nm=NULL, na.popu=NULL, min.u=NULL)

- data (possible input: NULL or a data frame).

  The input must be a data frame containing the population metric measurements (identified by the popm argument) and: if partitioned = F (default), a metric of the distribution of the object/individual replicated measurements (e.g. SD, SE or U values at the object/individual level, identified by the popu argument); or if partitioned = T, an identifier (a vector or column header) allowing to identify the objects/individuals composing the population (identified by the ID argument). If partitioned = F, u.conv = T and u = "SE" or "2SE", the input of the nm argument can also be a column header of data argument. Data argument input must be NULL (default) if popm, ID or popu inputs are not column headers (e.g. if ID = df$column_name).

- popm (possible input: a column header from the data argument or a vector).

  The vector or column must contain the mean values of measurements for each object/individual of the population (if partitioned = F), or all replicated measurements of the objects/individuals of the population (if partitioned = T). The column or vector length must match the one of popu (if partitioned = F) or ID (if partitioned = T). popm, popu and/or ID must always have the same length and be ordered in the exact same way.

- ID (possible input: NULL, a column header from the data argument or a vector).

  Input only required if partitioned = T. The vector or column must contain PERFECTLY MATCHING identifiers (e.g. character strings, ID numbers) regarding popm input, to allow the function to identify objects/individuals within the population. The column or vector length must match the one of popm and be ordered in the exact same way.

- popu (possible input: NULL, a column header from the data argument or a vector).

  Only required if partitioned = F. The vector or column must contain values from a metric of distribution or uncertainty of popm object/individual measurement values (e.g. SD, SE or U uncertainty values).

- n (possible input: a positive integer).

  Specifies the number of alternative populations generated by the function and stored in the data frame in output. A n >= 10 000 is generally advised for reliable subsequent roll_ testings, while n=100 or 1000 can be useful for quick preliminary testing.

- partitioned (possible input: TRUE or FALSE).

  In partitioned = F mode (default), the intra-object/individual mean and dispersion or uncertainty values (e.g. SD, SE, U uncertainty values) are taken directly from popm and popu inputs, respectively. In such configuration, providing an input for ID is not necessary. Within the embed rnorm function (package stats), popu data are either used as such in rnorm (u.conv = F) or are converted to SD values before being processed (u.conv = T). In partitioned = T mode, the mean and standard deviation (SD) of each pool of object/individual popm values are calculated within re_gauss by grouping data with matching ID identifiers (NA values in popm are omitted; other NA values are generated when there is no replicate for a given object/individual measurement, see na.popu argument for how to deal with these NA values). These calculated mean and SD values are inputs for the rnorm function (package stats) embed in the re_gauss function.

- u (possible input: NULL, "SD", "2SD", "SE", "2SE", "u", "U(k=2)", "U(k=3)").

  Specifies the nature of dispersion or uncertainty values contained in popu when partitioned = F (can be left as NULL when partitioned = T). "SD" (for standard deviation) is the standard mode, as the rnorm function (package stats) embed in re_gauss is designed to receive SD values as input. "2SD", "SE" and "2SE" are converted to SD values within re_gauss if u.conv = T and that appropriate input for the nm argument is provided (if u = "SE" or "2SE"). re_gauss cannot convert "u", "U(k=2)" or "U(k=3)" to SD values (even with u.conv = T). For custom uses, all of these possible inputs can directly serve as input for the embed rnorm function in replacement of SD values (with u.conv = F), but the user should acknowledge rnorm will still treat it as SD values which affect the meaning of the output.

- u.conv (possible input: TRUE or FALSE). Argument only active when partitioned = F.

  Indicate if popu data should be converted in SD values from another form of dispersion or uncertainty metric (u.conv = T) or not (u.conv = F). For conversion from SE or 2SE values, specific input in the nm argument are required.

- nm (possible input: NULL, a positive integer, a column header from the data argument or a vector).

  Input (other than NULL) only necessary when partitioned = F, u.conv = T and u = "SE" or "2SE". Specifies the number of replicates for each object/individual of the population. Necessary for converting popu into SD values from SE or 2SE values. Can be provided as a vector matching popm and popu in length and order, as a column header from data (if data = a data frame), or as a positive integer. This last option set the same nm (equal to the integer) for every object/individual of the population.

- na.popu (possible input: NULL, a positive numeric value, "na.omit", "mean", "med", "min" or "max").

  Specifies the method to deal with NA values in dispersion/uncertainty data (within popu input or generated with partitioned=T), commonly generated when object/individual measurements have not been replicated (number of measurement = 1). If na.popu input is a positive numeric value, NA values are replaced by this numeric value. "na.omit" remove objects/individuals with NA values. If set to "mean", "med", "min" or "max", replace NA values by the calculated mean, median, minimum or maximum value (NA excluded) of the population, respectively. Values created by na.popu in replacement of NA values are processed like any other dispersion/uncertainty value regarding conversion to SD values (u.conv = T) and replacement through min.u related operations. Thus, consistency between the nature of popu and na.popu inputs is important.

- min.u (possible input: NULL or a positive numeric value).

  Inactive if NULL (default). When set to a numeric value, all values in popu below the specified value of min.u are replaced by the value of min.u (e.g. for when popu is thought to be underestimated). This also applies for popu values generated by na.popu and SD values generated through partitioned = T. Values modified by min.u are processed like any other dispersion/uncertainty value regarding conversion to SD values (u.conv = T). Thus, consistency between the nature of popu and min.u inputs is important and must match u argument input (e.g. "SD", "2SD", etc.). When partitioned = T, min.u must be a standard deviation (SD) value.

roll_ttest

Perform t tests on data frames generated by re_ functions.

(output: a list of object containing: 1)"pvalues", a one column data frame compiling all the p.values resulting from the tests; 2)"stats", a 1 x 3 data frame containing the percentages of tests resulting in a p.value lower than 0.001, 0.01 and 0.05; 3)"plot", an optional ggplot2 graph showing the distribution of p.values as a plot of density, histogram or both)

roll_ttest(p1, p2, do.plot=T, density=T, histogram=T, fill.color= c("dodgerblue4","dodgerblue2","deepskyblue2","red"), ...)

- p1, p2 (input: data frames from re_ functions).

  p1 and p2 inputs must be two data frames generated from re_ functions representing the two populations that the user want to compare. p1 and p2 need to have the same number of columns and it is recommended to generate them using the same re_ function (either re_boot or re_gauss) and same settings (notably n and boot.n).

- do.plot (input: TRUE or FALSE).

  Specifies if a plot of the result has to be generated (TRUE) or not (FALSE). The package ggplot2 is needed if do.plot = TRUE.

- density (input: TRUE or FALSE).

  Specifies if the plot is a density plot (TRUE) or not (FALSE). Note: density and histogram can be combined in one plot.

- histogram (input: TRUE or FALSE).

  Specifies if the plot is a histogram plot (TRUE) or not (FALSE). Note: density and histogram can be combined in one plot.

- fill.color (input: a vector of length 4 containing only characters referring to R colors).

  Specifies the colors used to fill the [0 to 0.001], [0.001 to 0.01], [0.01 to 0.05], [0.05 to +Inf] sections (in this order) of the density plot (density = T) or histogram plot (histogram = T, density = F).

- ...

  Allow to pass specific instructions to the test function (e.g. alternative = "greater", paired = T, etc.).

roll_wtest

Perform Wilcoxon-Mann-Whitney tests (Wilcoxon rank sum tests) on data frames generated by re_ functions.

(output: a list of object containing: 1)"pvalues", a one column data frame compiling all the p.values resulting from the tests; 2)"stats", a 1 x 3 data frame containing the percentages of tests resulting in a p.value lower than 0.001, 0.01 and 0.05; 3)"plot", an optional ggplot2 graph showing the distribution of p.values as a plot of density, histogram or both)

roll_wtest(p1, p2, do.plot=T, density=T, histogram=T, fill.color=c("dodgerblue4","dodgerblue2","deepskyblue2","red"), ...)

- p1, p2 (input: data frames from re_ functions).

  p1 and p2 inputs must be two data frames generated from re_ functions representing the two populations that the user wants to compare. p1 and p2 need to have the same number of columns and it is recommended to generate them using the same re_ function (either re_boot or re_gauss) and same settings (notably n and boot.n).

- do.plot (input: TRUE or FALSE).

  Specifies if a plot of the result has to be generated (TRUE) or not (FALSE). The package ggplot2 is needed if do.plot = TRUE.

- density (input: TRUE or FALSE).

  Specifies if the plot is a density plot (TRUE) or not (FALSE). Note: density and histogram can be combined in one plot.

- histogram (input: TRUE or FALSE).

  Specifies if the plot is a histogram plot (TRUE) or not (FALSE). Note: density and histogram can be combined in one plot.

- fill.color (input: a vector of length 4 containing only characters referring to R colors).

  Specifies the colors used to fill the [0 to 0.001], [0.001 to 0.01], [0.01 to 0.05], [0.05 to +Inf] sections (in this order) of the density plot (density = T) or histogram plot (histogram = T, density = F).

- ...

  Allow to pass specific instructions to the test function (e.g. alternative = "greater", paired = T, etc.).

roll_rtest

Perform randomization tests on data generated by re_ functions.

(output: a list of object containing: 1)"pvalues", a one column data frame compiling all the p.values resulting from the tests; 2)"stats", a 1 x 3 data frame containing the percentages of tests resulting in a p.value lower than 0.001, 0.01 and 0.05; 3)"plot", an optional ggplot2 graph showing the distribution of p.values as a plot of density, histogram or both)

roll_rtest(p1, p2, nr, metric="mean", do.plot=T, density=T, histogram=T, fill.color=c("dodgerblue4","dodgerblue2","deepskyblue2","red"))

- p1, p2 (input: data frames from re_ functions).

  p1 and p2 inputs must be two data frames generated from re_ functions representing the two populations that the user wants to compare. p1 and p2 need to have the same number of columns and it is recommended to generate them using the same re_ function (either re_boot or re_gauss) and same settings (notably n and boot.n).

- nr (input: a positive integer).

  Specifies the number of randomizations to be made for each pairs of column of p1 and p2 data frame. Warning: regarding the typical size of re_ functions output (>1000 columns data frame), a nr <= 10 000, common for randomization tests, takes a considerable amount of computation time because the total number of randomization operations = nr x ncol(p1) (or nr x ncol(p2)).

- metric (input: "mean", "median", "t").

  Specifies the metric that will be monitored within the randomization tests. "mean" monitors the difference between the means of the populations. "median" monitors the difference between the medians of the populations. "t" monitors the difference between the t statistic of the populations.

- do.plot (input: TRUE or FALSE).

  Specifies if a plot of the result has to be generated (TRUE) or not (FALSE). The package ggplot2 is needed if do.plot = TRUE.

- density (input: TRUE or FALSE).

  Specifies if the plot is a density plot (TRUE) or not (FALSE). Note: density and histogram can be combined in one plot.

- histogram (input: TRUE or FALSE).

  Specifies if the plot is a histogram plot (TRUE) or not (FALSE). Note: density and histogram can be combined in one plot.

- fill.color (input: a vector of length 4 containing only characters referring to R colors).

  Specifies the colors used to fill the [0 to 0.001], [0.001 to 0.01], [0.01 to 0.05], [0.05 to +Inf] sections (in this order) of the density plot (density = T) or histogram plot (histogram = T, density = F).

VI Future developments

Planned future developments for rerollR notably include:

- Setting up multi-threaded computation, especially for roll_rtest
- Creating alternative functions to simplify re_ data frame generation on many populations.
- Creating alternative functions to simplify 2 by 2 comparisons of a set of re_ data frames by roll functions.

VII Words from the author and acknowledgments

I am Auguste Hassler, creator of this R package. I am a researcher in geochemistry, working on developing original (paleo-)biology applications in the fast-developing field of metal stable isotopes. A lot of questions addressed within my field imply to look at the isotopic compositions of a variety of species, individuals, or organic tissues, to try to identify differences of isotopic compositions (or an absence of it) that could be related to certain life history traits, physiological mechanisms, environmental influences, or pathologies. However, since my beginnings in this field, I always been frustrated by how available and widely used tools to make population scaled comparisons (e.g. t-test, Wilcoxon rank-sum test, randomization tests) were neglecting the uncertainties of the measurements constitutive of the compared populations. The uncertainties of measurements are never computed in the most used versions of these tests and my search for better tools remained unsuccessful for a while. Yet, a very inspiring talk from Jérémie Bardin about re-sampling and randomization in paleontology during the 2022 annual congress of the APF (Association de Paléontologie Française), finally made me realize that the solution was just under my nose for years. At this time I was already using re-sampling methods for some estimations of uncertainty, but I had never thought it could be the key for my population test problem. It then struck me that solving my population test issue could not only be beneficial to my own research, but also to other geochemists and scientists from other fields. This is what pushed me to develop this R package, with the aim of making this tool accessible and as easy to use as possible for all. I hope this can help those who were suffering from the same test frustrations as me, and that it can overall contribute to improve the depth of discussions of populations tests.

Deserved acknowledgments are never superfluous, so I would like to thank Jérémie Bardin for his inspiring talk, as well as the organizers of the APF 2022 conference. Thanks to Clément Bataille, Kate Britton and their respective teams for their feedbacks to the project and the time they gave me to work on it. Thanks to the UKRI for funding my time working on this package as part of the early stages of the PleistoDem project (MSCA Postdoctoral Fellowships 2021 selection, UKRI Fellowship Postdoc Guarantee). Thanks to Jeremy Martin, Vincent Balter and the rest of my former PhD team and lab for what I learnt during my first years of research. Thanks to all the testers who contributed to improve this package and to all the persons who provided valuable feedback for it.