exams.forge

Sigbert Klinke, Kleio Chrysopoulou Tseva

2024-07-09

Contents

```
library(exams.forge)
#> Lade nötiges Paket: tools
#> Lade nötiges Paket: polynom
#> exams.forge 1.0.10: see the package vignette with `vignette("exams.forge")`
```

Introduction

When devising exercises for students, two primary challenges emerge. Firstly, not all datasets prove suitable; students' tendencies to round intermediate results can lead to differing yet accurate solutions. Secondly, the lack of access to these intermediate values may hinder solution explanation. This can be addressed by expanding existing routines.

The former problem arises mainly when students are prohibited from using computers for exercises. Genuine comprehension of statistical coefficients or graphical representations is best achieved through manual calculation —an approach similar to memorizing multiplication tables. Without this hands-on engagement, students risk becoming mere button-pushers without deeper understanding.

To tackle these challenges effectively, a decision was made to create carefully curated datasets, allowing for precise control over their nuances. The general approach is outlined below:

```
library("exams")
library("exams.forge")
repeat {
  ... # some data generation
  if (condition_holds) break
}
```

For instance, in calculating the median from five observations x_i, we determine that the solution lies with the third sorted observation, $x_{(3)}$. Yet, it's crucial to verify that this third sorted observation doesn't coincide with the third observation itself. Otherwise, a student might overlook a crucial step in median computation. This concern is resolved as follows:

```
library("exams")
library("exams.forge")
repeat {
  x  <- sample(1:10, size=5)
  sx <- sort(x)
  if (x[3]!=sx[3]) break
}
x
```

The `exams.forge` package was developed with the primary objective of "forging" exam tasks in combination

with the `exams` package, along with auxiliary functions aimed at streamlining the process of generating Moodle exercises. The package consists of various functions divided into 7 categories based on their attributes. The nomenclature of the categories is as follows: Data Conversion and Modification, Statistical Analysis, Mathematical Computations, Exercise Generation, String Manipulation, LaTeX and HTML Functions and General Purpose Functions.

The `exams.forge` package is intended for educators responsible for crafting examination materials within the domain of statistics, for example fundamental courses like Statistics I and II, utilizing the R programming language. The objective is to optimize the process of generating a substantial array of assessment items, thus allowing instructors to channel their efforts toward enhancing the substantive quality of the tasks.

To what extent has the package been employed thus far?

As a result of the onset of the COVID-19 pandemic in the spring of 2020, the Chair of Statistics at the Humboldt University of Berlin implemented non-compulsory, remote, digital examinations accommodating up to 500 participants. These examinations were administered within the domain of the foundational courses, Statistics I and II, deploying the Moodle platform for exams.

In the context of Statistics I and II, each examination comprised a set of twenty questions, and for every question, an extensive array of one hundred variants was crafted. These variants encompassed a spectrum of distinctions, including variations in numerical values, shifts in content, or the weaving of diverse narratives. Moodle, our chosen platform, employed a random selection process, where one of the hundred available variants was assigned to each student. This meticulous approach guaranteed that each student received a unique examination, as opposed to the two-variant approach frequently taken in traditional face-to-face examinations.

In summary, `exams.forge` is an R package designed for educators that simplifies the creation of statistical exercises. Beyond enhanced statistical functions, it offers specialized formatting tools, data generation functions, and XML file adaptations, created by the exams package, to facilitate the integration of exercises into Moodle.

Now, let's explore the specifics of the first category, where we delve into a set of essential functions designed to enhance utility and streamline various data processing tasks.

Data Conversion and Modification

Classical Univariate Time Series

`ts_data`

Creates a univariate time series by combining elements of a linear or exponential trend, additive or multiplicative seasonal adjustment, and white noise. The function generates a time series object with specified parameters, including the length of the series, trend presence and coefficients, seasonal adjustment (with coefficients), and error terms (with coefficients). The resulting time series is structured as a `ts_data` object, allowing for further analysis and exploration.

```
# Generate a time series
ts_eg <- ts_data(end = 20, trend = TRUE, trend.coeff = c(1, 0.5),
                                season = TRUE, season.coeff = c(0.2, 0.1),
                                error = TRUE, error.coeff = 0.1, digits = 2)
print(ts_eg)
#> $t
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
#>
#> $s
#>  [1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
#>
#> $xt
```

```
#>  [1]  1.67  2.08  2.57  3.09  3.70  4.03  4.70  5.05  5.78  6.19  6.78  7.14
#> [13]  7.74  8.12  8.67  9.26  9.71 10.23 10.74 10.96
#>
#> attr(,"class")
#> [1] "ts_data"
```

as_ts

Transforms a `ts_data` object into a time series object (`ts`).

```
ts <- ts_data(12, trend.coeff= c(sample(0:10, 1), sample(1+(1:10)/20, 1)))
as_ts(ts)
#> Time Series:
#> Start = 1
#> End = 12
#> Frequency = 1
#>  [1]  2.2  3.4  4.6  5.8  7.0  8.2  9.4 10.6 11.8 13.0 14.2 15.4
```

ts_moving_average

The `ts_moving_average` function calculates the moving average for a `ts_data` object. This function takes a `ts_data` object (`ts`) and a user-defined order for the moving average (`order`). The result is an extended `ts_data` object containing information about the filter used for the moving average (`filter`) and the computed moving average values (`moving.average`).

```
# Create a time series data object with sinusoidal fluctuations
ts <- ts_data(20, trend.coeff = c(2))

# Compute the moving average with an order of 5
result_ts <- ts_moving_average(ts, 5)

# Display the original and extended time series data objects
cat("Original Time Series Data:\n")
#> Original Time Series Data:
str(ts)
#> List of 3
#>  $ t : num [1:20] 1 2 3 4 5 6 7 8 9 10 ...
#>  $ s : num [1:20] 1 1 1 1 1 1 1 1 1 1 ...
#>  $ xt: num [1:20] 2 2 2 2 2 2 2 2 2 2 ...
#>  - attr(*, "class")= chr "ts_data"

cat("\nExtended Time Series Data with Moving Average:\n")
#>
#> Extended Time Series Data with Moving Average:
str(result_ts)
#> List of 5
#>  $ t             : num [1:20] 1 2 3 4 5 6 7 8 9 10 ...
#>  $ s             : num [1:20] 1 1 1 1 1 1 1 1 1 1 ...
#>  $ xt            : num [1:20] 2 2 2 2 2 2 2 2 2 2 ...
#>  $ filter        : num [1:5] 0.2 0.2 0.2 0.2 0.2
#>  $ moving.average: num [1:20] NA NA 2 2 2 2 2 2 2 2 ...
#>  - attr(*, "class")= chr "ts_data"
```

ts_trend_season

The `ts_trend_season` function estimates a trend and season model from a time series data object (`ts_data`). It allows for flexible modeling, enabling the specification of linear or exponential trends and additive or multiplicative seasonality. The function returns an extended `ts_data` object with various components, including the estimated trend, season, combined trend and season, as well as relevant coefficients. It also provides information about the variance of residuals and the goodness of fit (R^2) for the final model.

```
# Create a time series data object with a linear trend
ts <- ts_data(12, trend.coeff = c(sample(0:10, 1), sample(1 + (1:10)/20, 1)))

# Estimate trend and season
result_ts <- ts_trend_season(ts)

# Display the extended time series data object
str(result_ts)
#> List of 8
#>  $ t           : num [1:12] 1 2 3 4 5 6 7 8 9 10 ...
#>  $ s           : num [1:12] 1 1 1 1 1 1 1 1 1 1 ...
#>  $ xt          : num [1:12] 11.1 12.1 13.2 14.2 15.2 ...
#>  $ trend.linear: logi NA
#>  $ season      : num(0)
#>  $ trend.season: logi NA
#>  $ var         : num NA
#>  $ r.squared   : num NA
#>  - attr(*, "class")= chr "ts_data"
```

Confidence Intervals

CImulen_data

This function generates data to determine the required sample size for constructing a confidence interval for the population mean with minimal rounding operations. Either the stimation error (`e`) or the length of the interval (`l`) must be provided. The relationship between `l` and `e` is given by $l = 2 \times e$. The function ensures that the computed standard deviation (`s`) differs from the known population standard deviation (`sigma`).

$$n = \left\lceil \frac{\sigma^2 \cdot z^2_{1-\frac{\alpha}{2}}}{e^2} \right\rceil = \left\lceil \frac{\sigma^2 \cdot z^2_{1-\frac{\alpha}{2}}}{4 \cdot l^2} \right\rceil$$

```
# Generate data for a confidence interval with estimation error ranging from 0.1 to 1.0
result <- CImulen_data(sigma = 1:10, e = (1:10)/10)
str(result)
#> List of 10
#>  $ e         : num 0.2
#>  $ sigma     : int 7
#>  $ conf.level: num 0.99
#>  $ l         : num 0.4
#>  $ x         : num 0.995
#>  $ q         : num 2.58
#>  $ q2        : num 6.63
#>  $ n         : num 8128
#>  $ N         : num 8128
#>  $ s         : num 6.97
result <- CImulen_data(sigma = 1:10, e = (1:10)/10, full=TRUE)
head(result)
#>      e sigma conf.level   l    x        q       q2          n    N         s
#> 1  0.1     1        0.9 0.2 0.95 1.644854 2.705543  270.55435  271 1.0143428
#> 2  0.2     1        0.9 0.4 0.95 1.644854 2.705543   67.63859   68 0.9202364
#> 3  0.3     1        0.9 0.6 0.95 1.644854 2.705543   30.06159   31 1.0385804
```

```
#> 11 0.1     2         0.9 0.2 0.95 1.644854 2.705543 1082.21738 1083 2.0129845
#> 12 0.2     2         0.9 0.4 0.95 1.644854 2.705543  270.55435  271 2.0676217
#> 13 0.3     2         0.9 0.6 0.95 1.644854 2.705543  120.24638  121 2.0788363
```

CIpilen_data

The `CIpilen_data` function is designed for generating data to determine the necessary sample size of a confidence interval for the population proportion using z^2/l^2). The estimation error (`e`) or the length of the interval (l) must be provided, where the relationship between them is defined as ($l = 2 * e$). The function ensures that the computed sample proportion (`p`) deviates from the known population proportion (`pi`).

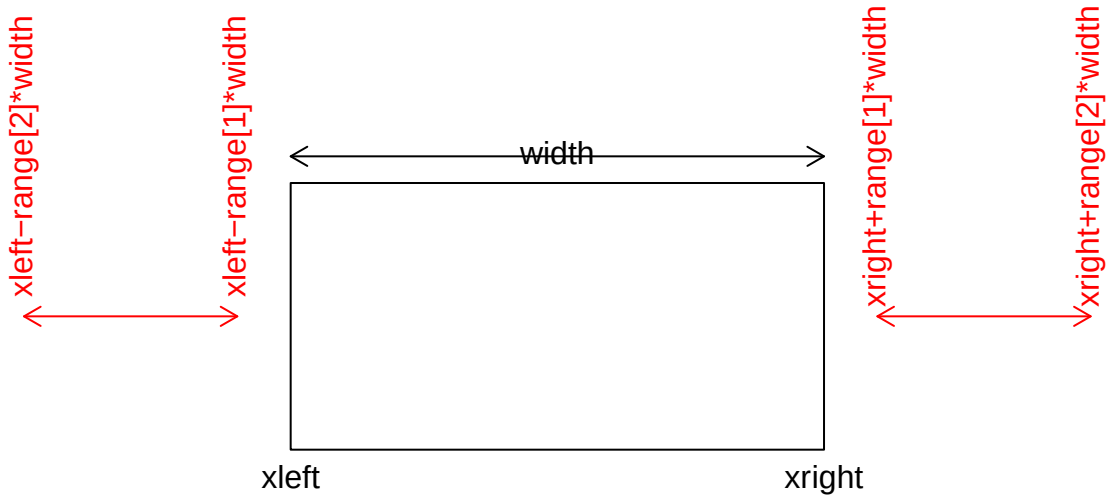
```
# Generate data for a confidence interval with estimation error 0.1
result <- CIpilen_data(pi = (1:9/10), e = (1:9)/10)

# Display the result
str(result)
#> List of 10
#>  $ e         : num 0.2
#>  $ pi        : num 0.8
#>  $ conf.level: num 0.99
#>  $ l         : num 0.4
#>  $ x         : num 0.995
#>  $ q         : num 2.58
#>  $ q2        : num 6.63
#>  $ n         : num 41.5
#>  $ N         : num 42
#>  $ p         : num 0.87
```

Data Generation

add_data

`add_data` adds data point(s) to the left and/or the right of a given data vector `x`.

1. A box and its width is determined by
   - `box="range"` gives a box width of `width=max(x)-min(x)` and two points `xleft=min(x)` and `xright=max(x)`
   - `box="box"` gives a box width of `width=IQR(x)` and two points `xleft=quantile(x, 0.25)` and `xright=quantile(x, 0.75)`
   - `box=c(xleft, xright)` gives a box width of `width=xright-xleft` and two points `xleft` and `xright`
2. The numbers of additional data points is determined by `n`
   - `n=c(nleft, nright)` gives the number of points to generate at the left and right
   - `n=1` is a short form of `c(0,1)` (the default)
3. Within the interval [`xleft-range[2]*width`; `xleft-range[1]*width`] are `nleft` points uniformly drawn and within the interval [`xright+range[1]*width`; `xright+rang[2]*width`] are `nleft` points uniformly drawn (both intervals are colored in red)




```
x <- runif(7, 165, 195)
xr <- add_data(x, "range", n=c(0,1), range=c(1,1.5))
round(xr)
#> [1] 190 178 192 171 191 183 174 216
xb <- add_data(x, "box", n=c(0,1), range=c(1,1.5))
round(xb)
#> [1] 190 178 192 171 191 183 174 206
x1 <- add_data(x, box=c(165,195), n=c(0,1), range=c(1,1.5))
round(x1)
#> [1] 190 178 192 171 191 183 174 231
```

`cor_data` and `meanint_data`

In this exercise, researchers aim to determine which variable, "number of absences in high school" (X) or "monthly income of parents" (Z), better predicts students' average grade points (Y) in the baccalaureate exam.

```
n <- sample(seq(25,50,5),1)
y <- meanint_data(n, c(2,12))
x <- meanint_data(n, c(36, 50))
z <- meanint_data(n, c(2,6))
yx <- cor_data(y, x, r=sample((5:9)/10, 1))
yz <- cor_data(y, z, r=sample((5:9)/10, 1))
```

Here, the function `meanint_data` is used to generate random data for the variables X and Z. It takes two arguments: `n`, the number of the observations and `v`, a vector with two elements specifying the range for the data where the `n` values are allowed. The first element of the vector specifies the lower limit of the data and the second one specifies the upper limit for the data.

The second function from the `exams.forge` package we are dealing with in this exercise is `cor_data`. This function is used in order to create a data set with two variables and a desired correlation coefficient `r`. It consists of 3 arguments: the dependent variable Y , the independent variables X and Z and the correlation coefficient `r`. The function contains 2 more arguments not shown in this exercise, namely the argument `method` and the argument `maxit`. `method` indicates which correlation coefficient is to be computed, in this case it is going to be the default pearson correlation coefficient, and `maxit` presents the maximal number of iterations that is set by default on 1000.

Overall, these two functions help in generating random data that simulate the relationships between variables as described in the exercise.

Number of Observations

data_n, data_n25 and data_nsq

The `data_n`, `data_nsq`, and `data_n25` functions are designed to generate sequences of sample sizes within a specified range, from `min` to `max`. Each function serves a unique purpose:

- `data_n` generates a sequence of sample sizes in the specified range.
- `data_n25` generates a sequence of sample sizes in the specified range that are divisible only by 2 and 5.
- `data_nsq` generates a sequence of sample sizes in the specified range whose square root is an integer.

```
# Generate a sequence of sample sizes from 5 to 10
data_n(10)
#> [1]  5  6  7  8  9 10

# Generate a sequence of sample sizes whose square root is an integer, from 9 to 961
data_nsq(1000)
#>  [1]   9  16  25  36  49  64  81 100 121 144 169 196 225 256 289 324 361 400 441
#> [20] 484 529 576 625 676 729 784 841 900 961

# Generate a sequence of sample sizes divisible only by 2 and 5, from 5 to 1000
data_n25(1000)
#>  [1]    5    8   10   16   20   25   32   40   50   64   80  100  125  128  160
#> [16]  200  250  256  320  400  500  512  625  640  800 1000
```

Number Properties

all_integer

Proves whether all xs are integer.

```
numbers_check <- c(4, 10, 7.00001)
all_integer(numbers_check)
#> [1] FALSE
```

divisor_25

Checks if a number can be represented as a product of powers of 2 and 5.

```
number_check <- 0.3125
result <- divisor_25(number_check)
```

has_digits

Verifies whether the decimal part of a number consists only of digits within a specified tolerance.

```
# Taken from the exercise "Club_Raucher2"
maxn <- 100
repeat {
  n  <- sample(seq(5, maxn, 5),1)
  p  <- sample((1:20)/100, 1)
  x  <- n*c(p, 1-p)
  if (all(has_digits(x, 0))) break
}
print(has_digits(x, 0))
#> [1] TRUE TRUE
```

prime_numbers

Generates a list of prime numbers up to a specified limit.

```
prime_numbers(20)
#> [1]  2  3  5  7 11 13 17 19
```

primes

Computes the prime factorization of each element in a numeric vector, providing a matrix that delineates the power of each prime number.

```
primes(1:5)
#>      2 3
#> [1,] 0 0
#> [2,] 1 0
#> [3,] 0 1
#> [4,] 2 0
#> [5,] 0 0
```

Result Generation with Rounding

as_result, rounded, tol, val and digits

This set of functions is designed to facilitate precise rounding of a numerical input `x` based on specified `digits` and a user-defined rounding function (`FUN`). Additionally, the functions offer a convenient way to set a tolerance for the result. If a tolerance is not explicitly provided, it defaults to the maximum of 2 times 10 to the power of negative `digits`.

- `as_result (x, digits, tol = NA, FUN = round2)`: Rounds the input `x` with specified `digits` using the specified rounding function (`FUN`), and allows for setting a tolerance (defaulting to the maximum of 2 times 10 to the power of negative `digits` if not provided).
- `tol(x)`: Computes the tolerance for a given input `x`.
- `rounded(x)`: Returns the rounded value of `x`.
- `val(x)`: Returns the value of `x`.
- `digits(x)`: Returns the specified digits for rounding `x`.

```
x <- as_result(1/3, "prob")
tol(x)
#> [1] 2e-04
rounded(x)
#> [1] 0.3333
digits(x)
#> [1] 4
val(x)
#> [1] 0.3333333
```

Tables

as_table

Transforms a vector into a horizontal table, facilitating a more structured representation of the data. The parameters are the same as in `xtable` which is used internally. Intended for the use as (class) frequency table.

```
x   <- runif(3)
tab <- vec2mat(x, colnames=1:length(x))
```

```
as_table(tab)
#> % latex table generated in R 4.4.1 by xtable 1.8-4 package
#> % Tue Jul  9 11:29:44 2024
#> \begin{table}[ht]
#> \centering
#> \begin{tabular}{rrrr}
#>   \hline
#>  & 1 & 2 & 3 \\
#>   \hline
#> 1 & 0.21 & 0.21 & 0.86 \\
#>    \hline
#> \end{tabular}
#> \end{table}
#>  [1] "% latex table generated in R 4.4.1 by xtable 1.8-4 package"
#>  [2] "% Tue Jul  9 11:29:44 2024"
#>  [3] "\\begin{table}[ht]"
#>  [4] "\\centering"
#>  [5] "\\begin{tabular}{rrrr}"
#>  [6] "  \\hline"
#>  [7] " & 1 & 2 & 3 \\\\ "
#>  [8] "  \\hline"
#>  [9] "1 & 0.21 & 0.21 & 0.86 \\\\ "
#> [10] "   \\hline"
#> [11] "\\end{tabular}"
#> [12] "\\end{table}"
tab <- vec2mat(x, colnames=sprintf("%.0f-%0.f", 0:2, 1:3))
as_table(tab)
#> % latex table generated in R 4.4.1 by xtable 1.8-4 package
#> % Tue Jul  9 11:29:44 2024
#> \begin{table}[ht]
#> \centering
#> \begin{tabular}{rrrr}
#>   \hline
#>  & 0-1 & 1-2 & 2-3 \\
#>   \hline
#> 1 & 0.21 & 0.21 & 0.86 \\
#>    \hline
#> \end{tabular}
#> \end{table}
#>  [1] "% latex table generated in R 4.4.1 by xtable 1.8-4 package"
#>  [2] "% Tue Jul  9 11:29:44 2024"
#>  [3] "\\begin{table}[ht]"
#>  [4] "\\centering"
#>  [5] "\\begin{tabular}{rrrr}"
#>  [6] "  \\hline"
#>  [7] " & 0-1 & 1-2 & 2-3 \\\\ "
#>  [8] "  \\hline"
#>  [9] "1 & 0.21 & 0.21 & 0.86 \\\\ "
#> [10] "   \\hline"
#> [11] "\\end{tabular}"
#> [12] "\\end{table}"
```

assoc_data

Reorders observations in a frequency table to approximate a specified target association, while maintaining unchanged marginal frequencies. The function utilizes a provided frequency table and computes an association (or correlation) measure using the specified function (`FUN`). The target association may not be achieved entirely, especially for extreme target values like +1 or -1.

- `target`: Specifies the target association to be approximated. If set to `NA`, the original table is returned.
- `zero`: Allows for zero entries in the common distribution.
- `tol`: Sets the maximal deviation of the association measure from the target value.
- `maxit`: Limits the number of optimization steps.

A solution is not assured, necessitating adjustments to parameters such as `maxit`, `tol`, or a reconsideration of the chosen target value. The resulting association value is stored in the attribute `attr ("target")`.

```
# Reordering observations in a frequency table to approximate a target association

# Creating a frequency table (2x2) with arbitrary values
frequency_table <- matrix(c(10, 20, 30, 40), nrow = 2, byrow = TRUE)

# Defining a target association value
target_association <- 0.5

# Applying assoc_data to reorder the frequency table to approximate the target association
result_table <- assoc_data(frequency_table, target = target_association, zero = TRUE, tol = 0.1, maxit =

# Displaying the resulting reordered table
print(result_table)
#>      [,1] [,2]
#> [1,]    2   28
#> [2,]   38   32
#> attr(,"iterations")
#> [1] 13
#> attr(,"target")
#> [1] 0.4068942
```

Vector Generation and Transformation

random

The `random` function generates a random permutation of indices from 1 to the length of the input vector `v`.

```
random(-1:6)
#> [1] 8 3 2 4 6 5 1 7
```

refer

The `refer` function facilitates the generation of names for elements within a vector. It provides a mechanism for assigning customized names based on a specified format, allowing us to enhance the interpretability of vector elements.

```
# Generating a vector of 5 random uniform values
x <- runif(5)

# Applying refer with LaTeX default format
latex_result <- refer(x)
str(latex_result)
```

```
#>  chr [1:5] "x_{1}" "x_{2}" "x_{3}" "x_{4}" "x_{5}"

# Applying refer with R default format
r_default_result <- refer(x, fmt = "%s[%.0f]")
str(r_default_result)
#>  chr [1:5] "x[1]" "x[2]" "x[3]" "x[4]" "x[5]"
```

In the first example, a vector `x` is created with 5 random uniform values using `runif(5)`. The `refer` function is then applied to `x` without specifying a custom format (`fmt`). By default, the LaTeX format is used, resulting in names that follow the pattern "x_{1}", "x_{2}", ..., "x_{n}", where n is the length of the vector. In the second example, the refer function is applied to the vector x with a custom format specified as "fmt="%s[%.0f]". This R default format results in names following the pattern"x[1]","x[2]", ...,"x[n]", where n is the length of the vector.

transformif

The `transformif` function offers conditional transformations for a vector `x` based on the specified condition `cond`. When the condition holds true, the transformation is applied to each element of `x`. The dynamic transformation is determined by parameters `a`, `b`, and `p`, allowing for versatile adjustments. Specifically, if the condition is met and `p` is set to 0, the transformation becomes $\log(a + b \cdot x)$; otherwise, it is $(a + b \cdot x)^p$.

```
# Generate a vector with a mix of positive and negative values
v <- c(2, -3, 1, 0, 5, -4)

# Transform only negative values using a custom shift (a) and scale (b)
transformed_vector <- transformif(v, v < 0, a = 2, b = 0.5)

# Display the original and transformed vectors
cat("Original Vector: ", v, "\n")
#> Original Vector:  2 -3 1 0 5 -4
cat("Transformed Vector: ", transformed_vector, "\n")
#> Transformed Vector:  2 3 1 0 5 3
```

vec2mat

The `vec2mat` function transforms a vector into either a horizontal or vertical matrix, allowing users to specify new column and row names. Existing names can be overwritten if `colnames` or `rownames` are provided.

```
# Generate a vector
vec <- c(1, 2, 3, 4, 5)

# Convert the vector to a horizontal matrix with custom column names
mat_horizontal <- vec2mat(vec, colnames = c("A", "B", "C", "D", "E"))

# Display the resulting matrix
print(mat_horizontal)
#>      A B C D E
#> [1,] 1 2 3 4 5

# Convert the vector to a vertical matrix with custom row names
mat_vertical <- vec2mat(vec, rownames = c("First", "Second", "Third", "Fourth", "Fifth"), horizontal = F

# Display the resulting matrix
print(mat_vertical)
#>        [,1]
```

```
#> First     1
#> Second    2
#> Third     3
#> Fourth    4
#> Fifth     5
```

Statistical Analysis

Approximations

binom2norm, clt2norm and t2norm

`binom2norm` checks if the sample size and parameters of a binomial distribution are suitable for approximating it with a normal distribution. Returns TRUE if conditions based on the binomial distribution parameters (`size`, `prob`, and optionally `type`) are met. The default threshold `c` is 9. The default value of $c = 9$ can be overwritten with `options(distribution.binom2norm=5)` or explicitly set.

```
# Single type
size <- 421
prob <- 0.5
cutoff <- 9
result_single <- binom2norm(size, prob, c=cutoff, type="single")
cat("Single type:", result_single, "\n")
#> Single type: TRUE

# Double type
result_double <- binom2norm(size, prob, c=cutoff, type="double")
cat("Double type:", result_double, "\n")
#> Double type: TRUE
```

`clt2norm` examines if the sample size (`n`) is large enough for the Central Limit Theorem to provide a reasonable approximation to a normal distribution. Returns TRUE if n is greater than a specified threshold (`c`), with the default threshold being 30. The default value of $c = 30$ can be overwritten with `options(distribution.clt2norm=5)` or explicitly set. Note that this function does not verify the existence of the expectation and variance, which are required by the Central Limit Theorem.

```
# Check for a broader range of observations
observations <- c(20, 40, 80, 120, 200, 300, 500, 1000)

# Assess whether each observation size is suitable for CLT approximation
clt_approximation_results <- clt2norm(n = observations)

# Display the results
print(clt_approximation_results)
#> [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

`t2norm` determines if the sample size (`n`) is large enough for a t-distribution to be reasonably approximated by a normal distribution. Returns TRUE if `n` is greater than a specified threshold (`c`), where the default threshold is 30. The default value of 30 can be overwritten with `options(distribution.t2norm=50)` or explicitly set.

```
# Check for a range of observations
observations <- c(10, 30, 50, 100, 200)

# Assess whether each observation size is suitable for t-distribution approximation
approximation_results <- t2norm(n = observations)
```

```
# Display the results
print(approximation_results)
#> [1] FALSE FALSE  TRUE  TRUE  TRUE
```

Bivariate Descriptive Statistics

grouped_data

Determines the mean, mode, quantile or median for data that has been grouped.

```
turnier <- ifelse(as.integer(format(Sys.Date(), "%Y")) %% 4 >= 2, "welt", "europa")
popSize     <- 100
classbreaks <- c(0, 50, 100, 200)
gd <- grouped_data(classbreaks, popSize*ddiscrete(runif(length(classbreaks)-1)), 0.5)
print(gd)
#> $result
#> [1] 100
#>
#> $group
#> [1] 2
#>
#> $tab
#>   lower upper mid width absfreq relfreq cumfreq density
#> 1     0    50  25    50      10     0.1     0.1   0.002
#> 2    50   100  75    50      40     0.4     0.5   0.008
#> 3   100   200 150   100      50     0.5     1.0   0.005
#>
#> $compute
#> [1] 0.5
```

In this example we can observe how the `grouped_data` function calculates the median from the data that has been grouped, namely it takes the `classbreaks`, the product of `popSize` and a random discrete uniform distribution created using `ddiscrete(runif(length(classbreaks)-1))`, and a weighting factor of 0.5 as parameters.

lcmval

This function computes the least common multiple for a numeric vector `x`.

```
lcmval(c(144, 160, 175))
#> [1] 50400
```

mcval

The function computes all the modes (most common value) of data.

```
# Numeric
x <- sample(1:5, size=25, replace = TRUE)
table(x)
#> x
#> 1 2 3 4 5
#> 5 4 7 4 5
mcval(x)
#> [1] 3
# Character
x <- sample(letters[1:5], size=25, replace = TRUE)
```

```r
table(x)
#> x
#> a b c d e
#> 5 5 4 4 7
mcval(x)
#> [1] "e"
# Histogram
x <- hist(runif(100), plot=FALSE)
mcval(x)
#> [1] 0.55
#> attr(,"pos")
#> [1] 6
mcval(x, exact=TRUE)
#> [1] 0.5857143
#> attr(,"pos")
#> [1] 6
```

nom.cc, nom.cramer, ord.spearman and ord.kendall (Association)

A set function which determines a compute association measure based on a contingency table:

- `nom.cc` (Corrected Contingency Coefficient): Computes the corrected contingency coefficient, a statistical measure assessing the association between two categorical variables. This coefficient is an enhancement of the contingency coefficient, correcting for potential biases.
- `nom.cramer` (Cramer's V or Phi): Calculates Cramer's V or Phi, providing a measure of association between categorical variables. Cramer's V is an extension of the phi coefficient, suitable for contingency tables with more than 2x2 cells.
- `ord.spearman` (Spearman's Rank Correlation): Computes Spearman's rank correlation, a non-parametric measure of association between two ordinal variables. It assesses the monotonic relationship between the variables, providing insights into their degree of association.
- `ord.kendall` (Kendall's Rank Correlation): Computes Kendall's rank correlation, a non-parametric measure evaluating the strength and direction of the association between two ordinal variables. This method is particularly suitable for detecting monotonic relationships.

```r
tab <- matrix(round(10*runif(15)), ncol=5)
nom.cc(tab)
#> [1] 0.4208204
nom.cc(tab, correct=TRUE)
#> [1] 0.4924329
nom.cramer(tab)
#> [1] 0.2678304
ord.spearman(tab)
#> [1] 0.1970919
ord.kendall(tab)
#> [1] 0.1636523
```

pearson_data

The following exercise asks for the calculation of the Bravais- Pearson correlation coefficient from the scores recorded from selected students in the mathematics and statistics exam.

```r
data(sos)
n     <- sample(4:8, 1)
rseq  <- seq(-0.95, 0.95, by=0.05)
```

```
r     <- sample(rseq, size=1, prob=rseq^2)
xy0   <- pearson_data(r=r, nmax=n, n=100, xsos=sos100)
str(xy0)
#>  num [1:4, 1:2] -5 -5 5 5 7 1 -1 -7
#>  - attr(*, "interim")= num [1:7, 1:5] -5 7 -5 7 25 49 -35 -5 1 -5 ...
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : chr [1:7] "$x_i$" "$y_i$" "$(x_i-\\bar{x})$" "$(y_i-\\bar{y})$" ...
#>   .. ..$ : chr [1:5] "1" "2" "3" "4" ...
```

The `pearson_data` function is used to generate an integer data set that can be used to compute a correlation, specifically the Pearson correlation coefficient. It is designed to create a data set with a specified desired correlation value `r` while following using the function `sumofsquares`.

We define 4 arguments for this function in our example:

1. `r`: as mentioned previously it is our desired correlation we want to achieve in the generated data set. The Pearson correlation coefficient measures the linear relationship between two variables and ranges from -1 to 1.
2. `n`: marks the number that we want to decompose as a sum of squares. The generated data set will consist of integer values, and this argument specifies how many data points should be included in the data set. Here `n` is set to 100, meaning that the data set will have 100 data points.
3. `nmax`: presents the maximal number of squares in the sum of squares. The `sumofsquares` function is used internally to generate the data set, and `nmax` controls the number of squares allowed in the decomposition.
4. `xsos`: is a precomputed matrix here set to 100.

`maxt`, not mentioned in this exercise, specifies the maximal number of seconds that the `pearson_data` routine should run. It sets a time limit on how long the function can take to generate the data set.

`sumofsquares`

This function endeavors to express an integer, denoted as `n`, as a summation of squared integers ($n = \sum_{i=1}^{k} x_i^2$), where each x_i lies within the range $1 \leq x_i < n$, and the count of terms (k) is bounded by $n_{\max}$. If the parameter `zerosum` is set to true, it ensures that the summation $\sum_{i=1}^{k} c_i x_i$ equals zero, where c_i can take values of either -1 or +1. The computational process is constrained by a specified time limit, `maxt` seconds, which might lead to an incomplete identification of all potential solutions. To optimize efficiency, the use of `rbind` operations within the function has been replaced by the allocation of matrices with a defined number of rows, denoted as `size`, to systematically collate the results.

```
# Example: Decomposing the integer 50 into a sum of squared integers
sos_example <- sumofsquares(50, nmax = 8, zerosum = FALSE, maxt = Inf, size = 100000L)
str(sos_example)
#>  num [1:18, 1:8] 1 5 3 1 3 1 1 1 2 1 ...
#>  - attr(*, "full")= logi TRUE
```

In this example, the `sumofsquares` function is employed to decompose the integer 50 into a sum of squared integers. The function allows a maximum of 8 terms in the decomposition (`nmax = 8`), does not enforce a zero sum (`zerosum = FALSE`), and has no time limit (`maxt = Inf`). The result is stored in the `sos_example` variable and then printed to the console.

Univariate Descriptive Statistics

`means` and `means_choice`

`means_choice` computes a list of mean values for a given data vector `x`:

- arithmetic mean,
- median,
- harmonic mean,
- geometric mean,
- (first) mode,
- trimmed mean, and
- winsorized mean.

If the parameter `trim` and/or `winsor` set to `NA` then these means will not be computed.

```
digits <- 2  # round to two digits
repeat {
  x  <- round(runif(7, min=165, max=195), digits)
  ms <- means_choice(x, digits)
  if (attr(ms, "mindiff")>0.1) break   # make sure that all values are different by 0.1
}
ms <- unlist(ms)
sc <- to_choice(ms, names(ms)=='mean') # arithmetic mean is the correct solution
str(sc)
#> List of 2
#>  $ questions: num [1:7] 177 186 186 187 187 ...
#>  $ solutions: logi [1:7] FALSE FALSE FALSE TRUE FALSE FALSE ...
```

The attribute `mindiff` gives the minimal distance between two mean values. This might be important for setting `extol` the tolerance for numeric solutions.

`scale_to`

Given a numeric vector it uses a linear transformation to re-scale the data to a given mean and standard deviation. The default is to standardize the data.

```
x <- runif(21)
y <- scale_to(x, mean=2, sd=0.5)
print(y)
#>  [1] 1.889316 2.274450 2.477659 1.895059 1.326979 2.218764 2.451160 1.248948
#>  [9] 1.844624 1.399239 2.488460 1.154182 1.774009 2.542026 1.775575 1.983191
#> [17] 2.642976 1.280808 2.425486 2.182298 2.724790
```

Combinatorics

`combinatorics`, `permutation`, `variation` and `combination`

Computation of all results for variation, combination and permutation with and without repetition.

```
variation(7,3)         # without replication
#> [1] 210
variation(7,3, TRUE)   # with replication
#> [1] 343
combination(7,3)       # without replication
#> [1] 35
combination(7,3, TRUE) # with replication
#> [1] 84
permutation(7)
#> [1] 5040
permutation(7, c(2,1,4)) # three groups with indistinguishable elements
#> [1] 105
z <- combinatorics(7, 4)
```

```
#> Warning in permutation(n, k): 'sum(k)<n', one element group(s) added
str(z)
#> List of 7
#>  $ permutation.n  : num 5040
#>  $ permutation.k  : num 24
#>  $ permutation.nk : num 210
#>  $ variation      : num 840
#>  $ variation.rep  : num 2401
#>  $ combination    : num 35
#>  $ combination.rep: num 210
#>  - attr(*, "mindiff")= num 0
```

```
permutation(5, c(2, 2))
#> Warning in permutation(5, c(2, 2)): 'sum(k)<n', one element group(s) added
#> [1] 30
```

The warning is raised because the sum of the specified group sizes (`c(2, 2)`) is less than the total number of elements (`n = 5`). This implies that the specified groups do not cover all elements, leaving some elements without a designated group.

In the context of permutations, the `permutation` function calculates the number of permutations of a set with specified group sizes. When there are not enough groups or when the sum of group sizes is less than the total number of elements, it means that some elements will be left unassigned or unmatched in the permutation process.

To account for these unmatched elements, the function automatically adds one or more groups, each containing a single element, to cover the remaining elements. This ensures that every element has a place in the permutations.

In this case, we have 5 elements and specified two groups, each with size 2. However, 1 element remains unassigned. The function adds a one-element group to accommodate the leftover element, and then it calculates the permutations of the entire set.

To summarize, the warning essentially declares that the specified group sizes don't cover all the elements, and the function has automatically adjusted by adding one or more one-element groups to make sure every element is considered in the permutation calculation.

`lfact`, `lfactquot` and `lbinom`

`lfact` calculates the natural logarithm of the factorial of a given number `n`. The factorial of a non-negative integer `n`, denoted as `n!`, is the product of all positive integers less than or equal to `n`. The natural logarithm of the factorial is computed to avoid overflow errors when dealing with large numbers. This function helps in computing large factorial values efficiently by returning their natural logarithms.

`lfactquot` calculates the natural logarithm of a quotient of factorials. It takes a number `n` and additional arguments as factors, and computes the natural logarithm of the quotient of the factorial of `n` and the product of factorials of the additional arguments. This function is useful in scenarios where calculating large factorials or their quotients is required, as it helps avoid numerical instability by working with logarithms.

`lbinom` computes the natural logarithm of the binomial coefficient, also known as "`n` choose `k`". The binomial coefficient `n choose k` represents the number of ways to choose `k` elements from a set of `n` elements without regard to the order of selection. The natural logarithm of the binomial coefficient is computed to handle large values efficiently and to avoid numerical overflow. This function is helpful in scenarios where the exact value of the binomial coefficient is not required, but its logarithm is sufficient for computation or analysis.

```
lfact(5)
#> [1] "5! = 5 \\cdot 4 \\cdot 3 \\cdot 2 \\cdot 1 = 120"
lfactquot(5,3,2)
```

```
#> [1] "\\frac{5!}{3! \\cdot 2!} = \\frac{5 \\cdot 4}{2 \\cdot 1} = 10"
lbinom(6,3)
#> [1] "\\left(\\begin{array}{c} 6 \\\\ 3 \\end{array}\\right)"
```

Distributions

ddunif2, pdunif2, qdunif2 and rdunif2

These functions provide probability mass function, distribution function, quantile function, and random generation for the sum of two independent discrete uniform distributions. The minimum and maximum values for the uniform distributions can be specified using the `min` and `max` parameters.

- `ddunif2`: Distribution function.
- `pdunif2`: Probability mass function.
- `qdunif2`: Quantile function.
- `rdunif2`: Random generation.

```
# Probability Mass Function
pdunif2(1:13)
#>          1          2          3          4          5          6          7
#> 0.00000000 0.02777778 0.08333333 0.16666667 0.27777778 0.41666667 0.58333333
#>          8          9         10         11         12         13
#> 0.72222222 0.83333333 0.91666667 0.97222222 1.00000000 1.00000000

# Distribution Function
ddunif2(1:13)
#>          1          2          3          4          5          6          7
#> 0.00000000 0.02777778 0.05555556 0.08333333 0.11111111 0.13888889 0.16666667
#>          8          9         10         11         12         13
#> 0.13888889 0.11111111 0.08333333 0.05555556 0.02777778 0.00000000

# Quantile Function
qdunif2((0:4)/4)
#> [1]  2  5  7  9 12

# Random Generation
rdunif2(10)
#>  [1] 11  8  9  4  7  9  5  8  8  8
```

distribution

An object of class `distribution` holds a distribution (of a random variable). It is specified by a name and the distribution values. The name is used create quantile (`("q", name)`) and cumulative distribution functions (`("p", name)`), for example

- `binom` hypergeometric distribution with parameters: `size`, `prob`
- `hyper` hypergeometric distribution with parameters: `m`, `n`, `k`
- `geom` geometric distribution with parameter: `prob`
- `pois` Poisson distribution with parameter: `lambda`
- `unif` hypergeometric distribution with parameters: `min`, `max`
- `exp`: exponential distribution with parameter: `rate`
- `norm`: normal distribution with parameters: `mean`, `sd`
- `lnorm`: log-normal distribution with parameters: `meanlog`, `sdlog`
- `t`: Student t distribution with parameter: `df`
- `chisq`: chi-squared distribution with parameter: `df`

- `f`: F distribution with parameters: `df1`, `df2`

The names of the above-mentioned distributions can be abbreviated; for all others the exact name must be given.

```
d <- distribution("t", df=15)
quantile(d, c(0.025, 0.975))
#> [1] -2.13145  2.13145
d <- distribution("norm", mean=0, sd=1)
cdf(d, c(-1.96, +1.96))
#> [1] 0.0249979 0.9750021
d <- distribution("binom", size=9, prob=0.5)
pmdf(d, 5)
#> [1] 0.2460938
```

`distribution` and `prob1`

The `exams.forge` package includes numerous functions designed to aid with exercises involving distributions.

In this exercise, the functions `distribution` and `prob1` serve as fundamental building blocks to unravel the intricacies of a dice game, where chance and probability intersect to determine one's success.

```
# Taken from the exercise "Würfel 2".
d      <- distribution("dunif", min=1, max=6)
border <- sample(1:5, size=1)+1
ptype  <- "point"
lsg    <- prob1(d, border)
sc     <- num_result(lsg, 4)
str(d)
#> List of 3
#>  $ min : num 1
#>  $ max : num 6
#>  $ name: chr "dunif"
#>  - attr(*, "class")= chr [1:2] "distribution" "list"
#>  - attr(*, "discrete")= num 1
print(lsg)
#> [1] 0.1666667
```

In the context of this exercise, the functions `distribution` and `prob1` play a crucial role in determining the probability of success in the dice game "Jule". `distribution` is used to model the outcomes of a six-sided die, while `prob1` calculates the probability of rolling the next required number, making them essential tools for understanding the game's dynamics.

`d <- distribution("dunif", min=1, max=6)`

This line defines a discrete uniform distribution called `d` with minimum value 1 and maximum value 6. Generally, the `distribution` function creates a distribution with a `name` in this case `dunif`.

`lsg <- prob1(d, border)`

This is the key part of the code. It calculates the point probability using the `prob1` function. The `prob1` function takes two arguments:

- `d`: The probability distribution (in this case, the discrete uniform distribution representing the six-sided die).
- `border`: A randomly selected value from the integers 1 to 5 (inclusive) using the `sample` function.

The `prob1` function calculates the probability of rolling the next required number in the game, given the current state of the game (represented by the `border` value). It is an important function for this exercise as

it directly addresses the main question of the probability of rolling the next required number.

`sc <- num_result(lsg, 4)`

This line defines a numerical result named `sc`. It captures the result of the point probability calculation done by the `prob1` function.

is.distribution

Checks if the `object` is a distribution object. If the `name` is given, it checks if the distribution type is the same.

```
# Check if an object is a distribution
x <- distribution("norm", mean=1.4, sd=0,44)
is.distribution(x)
#> [1] TRUE

# Check if an object is a specific distribution type
is.distribution(x, "exp")
#> [1] FALSE
```

binom_param and sqrtnp

The `binom_param` function computes parameters for a binomial distribution based on the number of trials (`n`) and the success probability (`p`). Optionally, it calculates the mean, standard deviation, and other measures. If mean, standard deviation, or other measures are not specified, they default to NA.

```
# Generate binomial parameters for a specific case
params <- binom_param(600, 0.6, mean = 0, sd = 0)

# Display the generated parameters
print(params)
#>     n   p mean sd
#> 1 600 0.6  360 12
```

The `sqrtnp` function calculates the square root of the product of `n`, `p`, and (1-p) for all combinations of given n and p values. If the resulting value has only digits after the decimal point, the corresponding n, p, and sqrt(np(1-p)) are presented in a structured data frame.

```
# Calculate sqrtnp for different combinations of n and p
result <- sqrtnp(n = c(50, 100, 150), p = c(0.25, 0.5, 0.75), digits = 3)

# Display the resulting data frame
print(result)
#>     n   p np snp
#> 1 100 0.5 50   5
```

In this example: - The `sqrtnp` function is employed to compute the square root of the product of n, p, and (1-p) for various combinations of n and p.

- The vectors `c(50, 100, 150)` and `c(0.25, 0.5, 0.75)` represent different observation numbers and probabilities, respectively.
- The `digits` parameter is set to 3, specifying the number of digits to consider.
- The resulting data frame, denoted as `result`, contains the combinations of n, p, and sqrt(np(1-p)) where the computed value has only digits after the decimal point.

This function is particularly useful for exploring the relationships between observation numbers, probabilities, and their respective square roots in a systematic manner. Adjusting the `digits` parameter allows users to

control the precision of the results.

cdf

Computes the cumulative distribution function of a distribution using `paste0('p', name)`.

```
# Create a distribution object for a normal distribution
normal_distribution <- distribution("norm", mean = 0, sd = 1)

# Calculate CDF for normal distribution
quantiles <- seq(-3, 3, by = 0.5)  # Quantiles for which to compute CDF
cdf_values <- cdf(normal_distribution, quantiles)  # Compute CDF values

# Display the results
cat("Quantile\tCDF Value\n")
#> Quantile CDF Value
cat("----------------------------\n")
#> ----------------------------
for (i in 1:length(quantiles)) {
  cat(quantiles[i], "\t\t", cdf_values[i], "\n")
}
#> -3        0.001349898
#> -2.5          0.006209665
#> -2        0.02275013
#> -1.5          0.0668072
#> -1        0.1586553
#> -0.5          0.3085375
#> 0         0.5
#> 0.5       0.6914625
#> 1         0.8413447
#> 1.5       0.9331928
#> 2         0.9772499
#> 2.5       0.9937903
#> 3         0.9986501
```

pmdf

Computes the probability mass/density function of a distribution using `paste0('d', name)`.

```
# Taken from the exercise "Haribo_3"
n    <- sample(2:10, 1)    # Gruppe 1: keine Frösche und Himbeeren
nj   <- 0
m    <- sample(2:10, 1)    # Gruppe 2: Frösche und Himbeeren
mj   <- sample(1:(m-1), 1)
k    <- mj+nj
d    <- distribution(name="hyper", m=m, n=n, k=k)
lsg  <- pmdf(d, k)
str(lsg)
#>  num 0.107
```

sample_size_freq

The `sample_size_freq` function assesses the compatibility of vectors containing possible sample sizes (`n`) and corresponding relative frequencies (`f`). It checks whether the product of sample sizes and relative frequencies results in integer absolute frequencies. This function is particularly useful in scenarios where the requirement for integer absolute frequencies is essential, such as in the design of experiments and statistical sampling.

```r
# Generating a set of random discrete probabilities with a total sum of 200
f <- ddiscrete(runif(6), unit=200)

# Checking compatibility for a sequence of sample sizes from 50 to 300 with a step of 1
result_default <- sample_size_freq(seq(50, 300, 1), f)
str(result_default)
#>  num 200

# Checking compatibility for a sequence of sample sizes from 10 to 700 with a step of 1, with 'which' s
result_specific <- sample_size_freq(seq(10, 700, 1), f, which=200)
str(result_specific)
#>  num 600
```

- `f` is generated using the `ddiscrete` function. It creates a set of discrete probabilities based on a random uniform distribution with six elements. The `unit=200` argument ensures that the total sum of probabilities is 200.
- `sample_size_freq` is applied to a sequence of sample sizes ranging from 50 to 300 with a step of 1.
- The function returns the first sample size in the sequence that results in integer absolute frequencies.
- `sample_size_freq` is applied to a sequence of sample sizes ranging from 10 to 700 with a step of 1.
- The `which=200` argument is specified, meaning the function specifically returns the sample size at the 200th position in the sequence.
- The function returns the sample size 200 from the specified sequence specifically, that satisfies the condition of creating integer absolute frequencies.

In summary, this example demonstrates the use of the `sample_size_freq` function to check the compatibility of different sequences of sample sizes with the given discrete probabilities. The results indicate which sample sizes, under the specified conditions, result in integer absolute frequencies.

q2norm

The `q2norm` function takes two arguments: `x`, which is a numeric vector containing two quantiles, and `probs`, which is a numeric vector containing the corresponding probabilities (defaulting to `c(0.025, 0.975)`). The function calculates the z-scores corresponding to the input probabilities.Based on the quantiles and z-scores, it estimates the mean and standard deviation of the corresponding normal distribution.The results are returned as a list with components `mean` and `sd`.

The example section demonstrates how to use the function with a set of example quantiles and probabilities, providing an estimated mean and standard deviation for the normal distribution.

```r
# Estimate mean and standard deviation for a normal distribution based on quantiles.

quantiles <- c(10, 20)  # Example quantiles
probabilities <- c(0.1, 0.9)  # Example probabilities
result <- q2norm(quantiles, probabilities)
str(result)
#> List of 2
#>  $ mean: num 15
#>  $ sd  : num 3.9
```

Histogram Manipulation and Analysis

histbreaks

The `histbreaks` function is designed to randomly select breakpoints from a given set of `breaks` values. When the outer parameter is set to TRUE, it ensures that the first and last elements of the `breaks` values are always included in the resulting breakpoints. If `size` is provided as a vector, the number of breakpoints is first sampled from this vector, adding flexibility to the selection process.

```
# Always includes 100 and 200 in the breakpoints
histbreaks(seq(100, 200, by = 10), 4)
#> [1] 100 110 180 200

# Always includes 100 and 200; randomly chooses between 3 to 5 breakpoints
histbreaks(seq(100, 200, by = 10), 3:5)
#> [1] 100 190 200

# May not include 100 and 200
histbreaks(seq(100, 200, by = 10), 4, outer = FALSE)
#> [1] 130 150 180 190
```

histdata

`histdata` computes data about the corresponding histogram to a vector like `hist`, but returns more information which might be necessary for exercises. In contrast to `hist` `histdata` requires that `breaks` covers the entire range of `x`.

`histdata` has the additional parameter `probs`. If `breaks="quantiles"` then it determines which quantiles are used.

```
x  <- runif(25)
h1 <- hist(x, plot=FALSE)
str(h1)
#> List of 6
#>  $ breaks  : num [1:6] 0 0.2 0.4 0.6 0.8 1
#>  $ counts  : int [1:5] 6 8 3 2 6
#>  $ density : num [1:5] 1.2 1.6 0.6 0.4 1.2
#>  $ mids    : num [1:5] 0.1 0.3 0.5 0.7 0.9
#>  $ xname   : chr "x"
#>  $ equidist: logi TRUE
#>  - attr(*, "class")= chr "histogram"
h2 <- histdata(x)
str(h2)
#> List of 15
#>  $ breaks  : num [1:6] 0 0.2 0.4 0.6 0.8 1
#>  $ counts  : int [1:5] 6 8 3 2 6
#>  $ density : num [1:5] 1.2 1.6 0.6 0.4 1.2
#>  $ mids    : num [1:5] 0.1 0.3 0.5 0.7 0.9
#>  $ xname   : chr "x"
#>  $ equidist: logi TRUE
#>  $ width   : num [1:5] 0.2 0.2 0.2 0.2 0.2
#>  $ x       : num [1:25] 0.0441 0.7582 0.0536 0.0744 0.2167 ...
#>  $ relfreq : num [1:5] 0.24 0.32 0.12 0.08 0.24
#>  $ cumfbrk : num [1:6] 0 0.24 0.32 0.12 0.08 0.24
#>  $ class   : int [1:25] 1 4 1 1 2 1 5 5 1 5 ...
#>  $ lower   : num [1:5] 0 0.2 0.4 0.6 0.8
```

```
#>  $ upper   : num [1:5] 0.2 0.4 0.6 0.8 1
#>  $ maxdens : int 2
#>  $ maxcount: int 2
#>  - attr(*, "class")= chr "histogram"
```

The returned list contains the following elements:

- `x`: the finite data values used
- `class`: the class number in which a value falls starting with 1 for the first class
- `xname`: the x argument name
- `breaks`: the class borders
- `lower`: the lower class borders
- `upper`: the upper class borders
- `width`: the class widths
- `mid`: the class mids
- `equidist`: if the classes are equidistant or not
- `counts`: the number of observations in each class
- `relfreq`: the relative class frequency
- `density`: the frequency density computed as relative frequency divided by class width

You can compute mean, quantile, median and mode for a histogram:

```
x <- runif(25)
h <- histdata(x)
# mean
mean(h)
#> [1] 0.452
# median & quantile
median(h)
#> [1] 0.42
#> attr(,"pos")
#> [1] 3
quantile(h)
#> [1] 0.0000000 0.2083333 0.4200000 0.6875000 1.0000000
#> attr(,"pos")
#> [1] 1 2 3 4 5
# mode
mcval(h)
#> [1] 0.1 0.3
#> attr(,"pos")
#> [1] 1 2
mcval(h, exact=TRUE)
#> [1] 0.2 0.2
#> attr(,"pos")
#> [1] 1 2
```

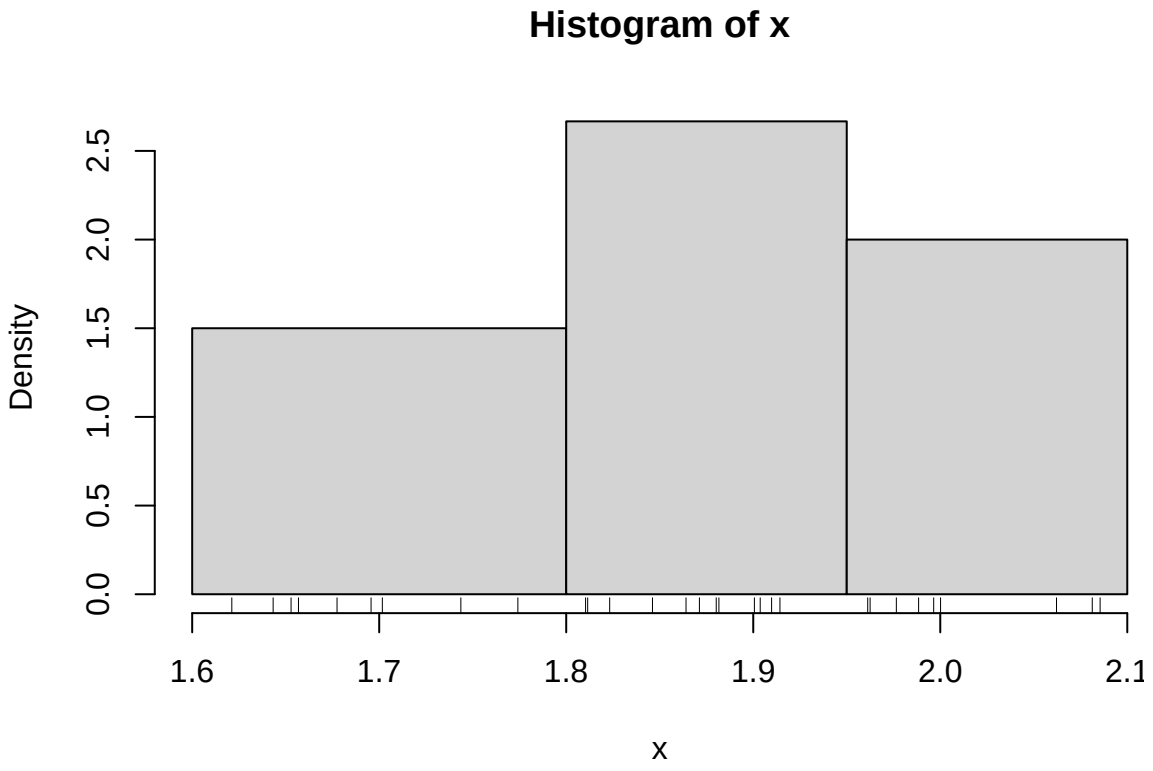
histwidth

Creates histogram data sampled from a set of class widths with following properties:

- the class density has a unique maximum,
- are the class density terminating numbers, and
- the class frequency maximum differs from the class density maximum.

```
hw <- histwidth(1.6, 2.1, widths=0.05*(1:4))
str(hw)
```

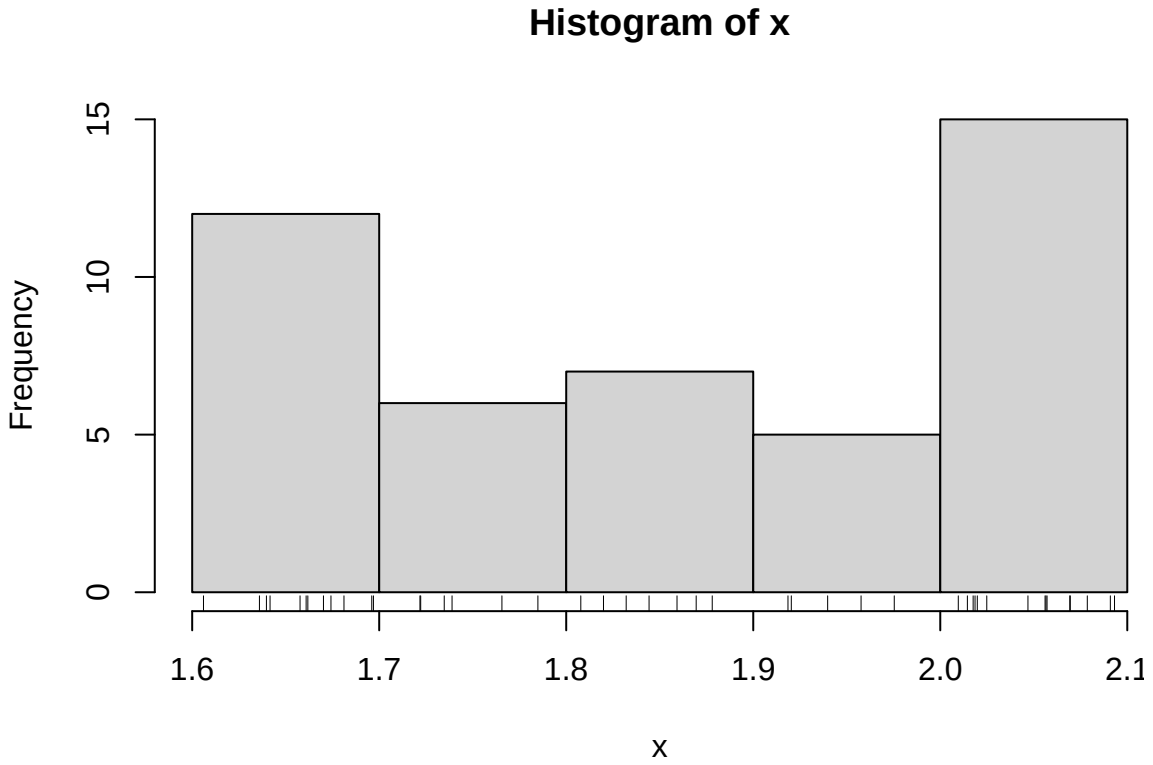
```
#> List of 3
#>  $ breaks : num [1:4] 1.6 1.8 1.95 2.1
#>  $ n      : num [1:3] 9 12 9
#>   ..- attr(*, "unit")= int 10
#>  $ decimal: logi FALSE
x  <- histx (hw$breaks, hw$n)
hist(x, hw$breaks)
rug(x)
```



histx

Generates a data set based on specified class borders (breaks) and the desired number of observations for each class. The resulting data set is structured to distribute data points across the defined classes.

```
breaks <- seq(1.6, 2.1, by=0.1)
x <- histx (breaks, sample(5:15, length(breaks)-1))
hist(x, breaks)
rug(x)
```



In this example, `histx()` is used to generate a data set based on the specified breaks and the number of observations in each class. The resulting data is then plotted using the `hist()` function, and a rug plot is added using the `rug()` function.

Probability Theory

`data_prob2`

The data_prob2 function generates a matrix of probabilities or frequencies based on the specified parameters. If data is provided, it will be normalized so that the sum of finite elements equals 1. If row and column names are not given, event names from the alphabet (`LETTERS`) are used. The resulting matrix has various attributes:

- `marginals`: A list of row and column marginal distributions.
- `byrow`: A matrix with conditional probabilities by row.
- `bycol`: A matrix with conditional probabilities by column.
- `expected`: A matrix with the expected probabilities under independence.
- `prob`: A vector of all computed probabilities (excluding the expected ones).

```
# Generate a data_prob2 object with default parameters
x <- data_prob2()
str(x)
#>  num [1:2, 1:2] 0.1 0.4 0.4 0.1
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : chr [1:2] "A" "!A"
#>   ..$ : chr [1:2] "B" "!B"
#>  - attr(*, "marginals")=List of 2
#>   ..$ : Named num [1:2] 0.5 0.5
#>   .. ..- attr(*, "names")= chr [1:2] "A" "!A"
#>   ..$ : Named num [1:2] 0.5 0.5
#>   .. ..- attr(*, "names")= chr [1:2] "B" "!B"
```

```
#>  - attr(*, "byrow")= num [1:2, 1:2] 0.2 0.8 0.8 0.2
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : chr [1:2] "A" "!A"
#>   .. ..$ : chr [1:2] "B" "!B"
#>  - attr(*, "bycol")= num [1:2, 1:2] 0.2 0.8 0.8 0.2
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : chr [1:2] "A" "!A"
#>   .. ..$ : chr [1:2] "B" "!B"
#>  - attr(*, "expected")= num [1:2, 1:2] 0.25 0.25 0.25 0.25
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : chr [1:2] "A" "!A"
#>   .. ..$ : chr [1:2] "B" "!B"
#>  - attr(*, "prob")= Named num [1:16] 0.1 0.4 0.4 0.1 0.5 0.5 0.5 0.5 0.2 0.8 ...
#>   ..- attr(*, "names")= chr [1:16] "A^B" "!A^B" "A^!B" "!A^!B" ...

# Generate a data_prob2 object with colnames="E"
data_prob2(colnames="E")
#>      E  !E
#> A  0.1 0.6
#> !A 0.2 0.1
#> attr(,"marginals")
#> attr(,"marginals")[[1]]
#>   A  !A
#> 0.7 0.3
#>
#> attr(,"marginals")[[2]]
#>   E  !E
#> 0.3 0.7
#>
#> attr(,"byrow")
#>            E        !E
#> A  0.1428571 0.8571429
#> !A 0.6666667 0.3333333
#> attr(,"bycol")
#>            E        !E
#> A  0.3333333 0.8571429
#> !A 0.6666667 0.1428571
#> attr(,"expected")
#>       E   !E
#> A  0.21 0.49
#> !A 0.09 0.21
#> attr(,"prob")
#>       A^E      !A^E      A^!E     !A^!E         A        !A         E        !E
#> 0.1000000 0.2000000 0.6000000 0.1000000 0.7000000 0.3000000 0.3000000 0.7000000
#>       A|E      !A|E      A|!E     !A|!E       E|A      E|!A      !E|A     !E|!A
#> 0.1428571 0.6666667 0.8571429 0.3333333 0.3333333 0.6666667 0.8571429 0.1428571

# Generate a data_prob2 object with nrow=3
data_prob2(nrow=3)
#>      D   !D
#> A 0.26 0.26
#> B 0.09 0.04
#> C 0.12 0.23
```

```
#> attr(,"marginals")
#> attr(,"marginals")[[1]]
#>    A    B    C
#> 0.52 0.13 0.35
#>
#> attr(,"marginals")[[2]]
#>    D   !D
#> 0.47 0.53
#>
#> attr(,"byrow")
#>           D        !D
#> A 0.5000000 0.5000000
#> B 0.6923077 0.3076923
#> C 0.3428571 0.6571429
#> attr(,"bycol")
#>           D        !D
#> A 0.5531915 0.4905660
#> B 0.1914894 0.0754717
#> C 0.2553191 0.4339623
#> attr(,"expected")
#>        D     !D
#> A 0.2444 0.2756
#> B 0.0611 0.0689
#> C 0.1645 0.1855
#> attr(,"prob")
#>       A^D       B^D       C^D      A^!D      B^!D      C^!D         A         B
#> 0.2600000 0.0900000 0.1200000 0.2600000 0.0400000 0.2300000 0.5200000 0.1300000
#>         C         D        !D       A|D       B|D       C|D      A|!D      B|!D
#> 0.3500000 0.4700000 0.5300000 0.5000000 0.6923077 0.3428571 0.5000000 0.3076923
#>      C|!D       D|A       D|B       D|C      !D|A      !D|B      !D|C
#> 0.6571429 0.5531915 0.1914894 0.2553191 0.4905660 0.0754717 0.4339623
```

ddiscrete

`ddiscrete` generates a finite one-dimensional discrete probability distribution. If the length of `x` is one then `x` is the number of elements. Otherwise `x` is considered a starting distribution and length of `x` is the number of elements.

The parameter `zero` determines if the final distribution can contain the probability entry zero or not. Since, for computation of exercises based on a one-dimensional discrete probability distribution, it is favorable that the entries are fractions having the same denominator, the parameter `unit` can be used for this purpose. Thus, if the smallest non-zero denominator should be `1/7` then use `unit=7`; the default is a power of 10.

```
ddiscrete(6) # fair dice
#> [1] 0.22 0.12 0.10 0.23 0.10 0.23
#> attr(,"unit")
#> [1] 100
x <- runif(6)
ddiscrete(x)
#> [1] 0.05 0.16 0.27 0.21 0.15 0.16
#> attr(,"unit")
#> [1] 100
ddiscrete(x, zero=TRUE)
#> [1] 0.06 0.14 0.27 0.22 0.14 0.17
```

```
#> attr(,"unit")
#> [1] 100
ddiscrete(x, unit=15)
#> [1] 0.06666667 0.20000000 0.26666667 0.20000000 0.13333333 0.13333333
#> attr(,"unit")
#> [1] 15
fractions(ddiscrete(x, unit=15))
#> [1] 1/15 2/15 4/15  1/5 2/15  1/5
#> attr(,"unit")
#> [1] 15
```

The next exercise acts as a second example for better understanding of the `ddiscrete` function:

Exercise: Modify the Discrete Probability Function for a Biased Coin

We consider a biased coin with an initial probability distribution represented as `c(0.8, 0.2, 0, 0, 0, 0)`, where the first element corresponds to the probability of getting heads, and the second element corresponds to the probability of getting tails.

Here: Firstly, we use the `ddiscrete` function to create a discrete probability function for the biased coin. Secondly, we allow zeros in the final probabilities. And thirdly, we experiment with different resolutions by specifying different units.

Hints: - We can use the `ddiscrete` function with the biased coin probabilities. - Set `zero = TRUE` to allow zeros in the final probabilities. - Experiment with different units, for example, `unit = 100` and `unit = 1000`.

```
# Exercise: Modify the discrete probability function for a biased coin

# Given biased coin probabilities (Heads, Tails)
biased_coin_prob <- c(0.8, 0.2, 0, 0, 0, 0)

# 1. Create a discrete probability function for the biased coin
biased_coin_fun <- ddiscrete(biased_coin_prob)
print(biased_coin_fun)
#> [1] 0.79 0.17 0.01 0.01 0.01 0.01
#> attr(,"unit")
#> [1] 100

# 2. Create a modified discrete probability function allowing zeros
modified_coin_fun <- ddiscrete(biased_coin_prob, zero = TRUE)
print(modified_coin_fun)
#> [1] 0.8 0.2 0.0 0.0 0.0 0.0
#> attr(,"unit")
#> [1] 100

# 3. Experiment with different resolutions (units)
unit_100 <- ddiscrete(biased_coin_prob, unit = 100)
unit_1000 <- ddiscrete(biased_coin_prob, unit = 1000)
print(unit_100)
#> [1] 0.76 0.20 0.01 0.01 0.01 0.01
#> attr(,"unit")
#> [1] 100
print(unit_1000)
#> [1] 0.799 0.197 0.001 0.001 0.001 0.001
#> attr(,"unit")
#> [1] 1000
```

This code performs the exercise steps, creating the original biased coin probability function, a modified version allowing zeros, and experimenting with different resolutions (units).

`ddiscrete2`

`ddiscrete2` generates a finite two-dimensional discrete probability distribution.

The generation has two steps:

1. Generate two marginal finite two-dimensional discrete probability distributions. Based on this a joint probability for two independent distributions is generated.
2. Define target measure for association and target value for the association for the joint distribution.

The current available association measure are:

- `nom.cc`: (corrected) contingency coefficient
- `nom.cramer`: Cramer's V or Phi
- `ord.spearman`: Spearman's rank correlation
- `ord.kendall`: Kendall's rank correlation

```
r <- ddiscrete(6)
c <- ddiscrete(6)
ddiscrete2(r, c)
#>        [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
#> [1,] 0.1035 0.1125 0.0135 0.0315 0.0585 0.1305
#> [2,] 0.0161 0.0175 0.0021 0.0049 0.0091 0.0203
#> [3,] 0.0345 0.0375 0.0045 0.0105 0.0195 0.0435
#> [4,] 0.0207 0.0225 0.0027 0.0063 0.0117 0.0261
#> [5,] 0.0529 0.0575 0.0069 0.0161 0.0299 0.0667
#> [6,] 0.0023 0.0025 0.0003 0.0007 0.0013 0.0029
#> attr(,"iterations")
#> [1] 0
#> attr(,"target")
#> [1] 0
#> attr(,"unit")
#> [1] 10000
ddiscrete2(r, c, FUN=nom.cc, target=0.4)
#>        [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
#> [1,] 0.1029 0.1134 0.0137 0.0312 0.0577 0.1311
#> [2,] 0.0169 0.0172 0.0001 0.0035 0.0106 0.0217
#> [3,] 0.0344 0.0384 0.0060 0.0087 0.0186 0.0439
#> [4,] 0.0224 0.0216 0.0008 0.0092 0.0096 0.0264
#> [5,] 0.0533 0.0581 0.0068 0.0151 0.0309 0.0658
#> [6,] 0.0001 0.0013 0.0026 0.0023 0.0026 0.0011
#> attr(,"iterations")
#> [1] 500
#> attr(,"target")
#> [1] 0.1796065
#> attr(,"unit")
#> [1] 10000
ddiscrete2(r, c, FUN=nom.cc, target=1)
#>        [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
#> [1,] 0.1020 0.1122 0.0129 0.0322 0.0599 0.1308
#> [2,] 0.0165 0.0194 0.0007 0.0068 0.0076 0.0190
#> [3,] 0.0360 0.0388 0.0018 0.0089 0.0192 0.0453
#> [4,] 0.0203 0.0200 0.0046 0.0066 0.0136 0.0249
#> [5,] 0.0535 0.0591 0.0073 0.0152 0.0295 0.0654
```

```
#> [6,] 0.0017 0.0005 0.0027 0.0003 0.0002 0.0046
#> attr(,"iterations")
#> [1] 500
#> attr(,"target")
#> [1] 0.1696594
#> attr(,"unit")
#> [1] 10000
```

The units are determined as units of `r` multiplied with the units of `c`. Since a iterative process is used the parameter `maxit` is set to 500. If the attribute `iterations` is equal to `maxit` then the iterative process has not been finished. The attribute `target` gives the association value obtained.

is.prob

The function `is.prob` serves the purpose of verifying whether a given numeric value `x` lies within the bounds of an open or closed interval defined by specified minimum (`min`) and maximum (`max`) values. By default, the function is configured to check if `x` falls within the standard open interval (0, 1), often associated with probability values.

```
is.prob(runif(1))
#> [1] TRUE
```

In this case, the `runif(1)` generates a random numeric value between 0 and 1, and the `is.prob` function confirms that the generated value indeed falls within the standard open interval (0, 1). The result, in this instance, is `TRUE`. The function is particularly useful for scenarios where it is essential to ascertain whether a given numeric value is within the expected range, such as verifying whether a number represents a valid probability within the unit interval (0, 1). The default settings of the function align with the typical interval used for probabilities, facilitating a straightforward validation process.

pprobability

The `pprobability` function is designed to facilitate the generation and estimation of polynomials for discrete random variables. This versatile function allows us to construct polynomials, estimate both least squares and maximum likelihood solutions, and provides flexibility in specifying various parameters.

```
y <- pprobability(0:2, coef=seq(-2, 2, by=0.1))
str(y)
#> List of 6
#>  $ p     :List of 3
#>   ..$ : 'polynomial' num [1:2] 0.1 -0.7
#>   ..$ : 'polynomial' num [1:2] 0.3 1.8
#>   ..$ : 'polynomial' num [1:2] 0.6 -1.1
#>   ..- attr(*, "class")= chr "polylist"
#>  $ ep    : 'polynomial' num [1:2] 1.5 -0.4
#>  $ x     : int [1:3] 0 1 2
#>  $ sample: int [1:3] 0 1 2
#>  $ LS    :List of 5
#>   ..$ pi  :List of 3
#>   .. ..$ : 'polynomial' num [1:3] 0.01 -0.14 0.49
#>   .. ..$ : 'polynomial' num [1:3] 0.49 -2.52 3.24
#>   .. ..$ : 'polynomial' num [1:3] 1.96 3.08 1.21
#>   .. ..- attr(*, "class")= chr "polylist"
#>   ..$ pf  : 'polynomial' num [1:3] 2.46 0.42 4.94
#>   ..$ dp  : 'polynomial' num [1:2] 0.42 9.88
#>   ..$ pest: num -0.0425
#>   .. ..- attr(*, "type")= chr "minimum"
```

```
#>   ..$ p   : num [1:3] 0.13 0.223 0.647
#>  $ ML    :List of 5
#>   ..$ pi  :List of 3
#>   .. ..$ : 'polynomial' num [1:2] 0.1 -0.7
#>   .. ..$ : 'polynomial' num [1:2] 0.3 1.8
#>   .. ..$ : 'polynomial' num [1:2] 0.6 -1.1
#>   .. ..- attr(*, "class")= chr "polylist"
#>   ..$ pf  : 'polynomial' num [1:4] 0.018 -0.051 -0.723 1.386
#>   ..$ dp  : 'polynomial' num [1:3] -0.051 -1.446 4.158
#>   ..$ pest: num -0.0323
#>   .. ..- attr(*, "type")= chr "maximum"
#>   ..$ p   : num [1:3] 0.123 0.242 0.636
```

The `pprobability` function, when called with the arguments `pprobability(0:2, coef = seq(-2, 2, by = 0.1))`, performs the following:

1. Generated Polynomials:
   - Three linear polynomials are generated based on the user-defined coefficients. The coefficients are sampled from the sequence `-2` to `2` in increments of `0.1`. Each polynomial corresponds to a value in the discrete random variable `0:2`.
2. Estimated Polynomial:
   - The estimated polynomial is the sum of the generated polynomials.
3. Values of the Random Variable:
   - The values of the discrete random variable: 0, 1, 2.
4. Sample Structure:
   - The sample structure represents the frequency of each value in the random variable. In this case, each value occurs once (`c(0, 1, 2)`).
5. Least Squares Results:
   - The least squares method is applied to estimate a polynomial. The results include the estimated polynomial, its degree, and coefficients.
6. Maximum Likelihood Results:
   - The maximum likelihood method is applied to estimate a polynomial. The results include the estimated polynomial, its degree, and coefficients.

The purpose of this function call is to generate and estimate polynomials for a discrete random variable (`0:2`) with a specified set of coefficients. The user-supplied coefficients (`seq(-2, 2, by = 0.1)`) influence the shape and characteristics of the generated polynomials. Both the least squares and maximum likelihood methods are used to estimate the polynomial parameters based on the generated data.

`prob`

Computes the probability for an interval between `min` and `max` (`max` included, `min` excluded).

```
# Compute the probability for an interval in a uniform distribution
d <- distribution("unif", min=1, max=7)
prob(d)
#> [1] 1
```

Simple Linear Regression

`lm1_data`

This function is designed to create data suitable for performing a simple linear regression with a predefined correlation coefficient. It accepts various parameters, including the desired correlation, the number of squares to decompose, and other options for data manipulation and scaling.

The steps the function performs are as follows:

1. Generate `x` and `y` data so that the sum of squares of the values equals `n` and the sum of values equals 0 for both `x` and `y`.
2. Re-scale the data using user-defined center and scale values.
3. Conduct a simple linear regression analysis on the transformed data, allowing users to explore the relationship between `x` and `y` with the specified correlation.

```
n     <- sample(4:8, 1)
lm1   <- lm1_data(0.4, nmax=n, xsos=sos100)
print(lm1)
#>
#> Call:
#> lm(formula = xy[, 2] ~ xy[, 1])
#>
#> Coefficients:
#> (Intercept)      xy[, 1]
#>        0.00         0.85
```

`lmr_data`

The `lmr_data` function in R serves the purpose of generating data suitable for conducting a simple linear regression analysis.

Arguments of the function include:

- `xr` and `yr`: The ranges for `x` and `y` values can be defined, allowing for controlled data generation.
- `n`: This parameter specifies the number of observations to create.
- `r`: If desired, a target correlation coefficient can be specified. If not provided, the function defaults to a zero correlation.
- `digits`: There is the option to set the precision for rounding `x` and `y` values individually.

Additional parameters can be passed to the function, which are further used in the underlying "cor_data" function.

The function returns an `lm` object, which includes various components such as the generated `x` and `y` values, sums, means, variations, covariance, correlation, and the coefficients of a linear regression model.

```
n   <- sample(c(4,5,8,10),1)
lmr <- lmr_data(c(1,3), c(2,8), n=n, r=sample(seq(0.1, 0.9, by=0.05), 1))
print(lmr)
#>
#> Call:
#> lm(formula = xy[, 2] ~ xy[, 1])
#>
#> Coefficients:
#> (Intercept)      xy[, 1]
#>       0.836        2.030
```

Tables

`incomplete_table`

The `incomplete_table` function is designed to complete a relative contingency table with missing values in such a way that the overall table entries can be recomputed. If a solution cannot be found, the function will generate an error.

Consider a relative contingency table represented by the matrix `tab`, which has some missing values. 7 missing values must be filled in order to make the table computationally complete.

```
tab <- rbind(c(0.02, 0.04, 0.34), c(0.02, 0.28, 0.3))
result <- incomplete_table(tab, 7)
print(result)
#>      [,1] [,2] [,3] [,4]
#> [1,] 0.02 0.04   NA  0.4
#> [2,]   NA 0.28   NA   NA
#> [3,] 0.04   NA   NA   NA
#> attr(,"fillin")
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    1    1
#> [3,]    2    2
#> [4,]    4    4
#> [5,]    2    2
#> [6,]    3    3
#> attr(,"full")
#>      [,1] [,2] [,3] [,4]
#> [1,] 0.02 0.04 0.34  0.4
#> [2,] 0.02 0.28 0.30  0.6
#> [3,] 0.04 0.32 0.64  1.0
# Here column no. 4 and row no. 3 constitute the summaries of their respective columns and rows.
```

Additionally, the function provides information about the filled-in values in the `fillin` attribute and the fully reconstructed table in the `full` attribute. The `fillin` matrix indicates which cells were filled and corresponds to the missing values in the incomplete table. The `full` matrix is the complete contingency table with all missing values filled.

```
# attr(,"fillin")
#      [,1] [,2]
# [1,]    2    2
# [2,]    2    2
# [3,]    4    4
# [4,]    1    1
# [5,]    3    3
# [6,]    3    3
# [7,]    1    1
```

In summary, the `incomplete_table` function helps to impute missing values in a relative contingency table, ensuring that the resulting table remains consistent and computationally valid.

```
#  attr(,"full")
#      [,1] [,2] [,3] [,4]
# [1,] 0.02 0.04 0.34  0.4
# [2,] 0.02 0.28 0.30  0.6
# [3,] 0.04 0.32 0.64  1.0
```

`table_data`

The `table_data` function is designed to generate a frequency table where each entry can be expressed in the form $2^{p_{ij}} \times 5^{q_{ij}}$. The function enforces the constraints $p_{ij} < m_2$ and $q_{ij} < m_5$. In the event that the algorithm fails to find a solution, an error is raised, prompting us to consider increasing the `unit` parameter for a more refined search. Once a valid table is identified, normalization is performed by dividing all entries by an appropriate factor to maintain integer values. Subsequently, a random multiplier in the format $2^p \times 5^5$

is selected, ensuring that the sum of the entries remains less than or equal to the specified limit `n`.

```
# Generate a frequency table with 4 rows and 3 columns
generated_table <- table_data(nrow = 4, ncol = 3, unit = 20, n = 150, maxit = 5000)

# Display the generated frequency table
print(generated_table)
#>      [,1] [,2] [,3]
#> [1,]   32   16   32
#> [2,]   32   16   32
#> [3,]    8    4    8
#> [4,]    8    4    8
#> attr(,"maxit")
#> [1] 33
```

In this example: - The `table_data` function is applied to create a frequency table with 4 rows and 3 columns.

- The `unit` parameter is set to 20, influencing the granularity of the search for a valid table.
- The `n` parameter is set to 150, indicating the maximum sum of entries.
- The resulting frequency table, denoted as `generated_table`, adheres to the conditions specified by the function, and all entries can be expressed in the form $2^{p_{ij}} \times 5^{q_{ij}}$.

Tests

`proptests`, `proptest_data` and `proptest_num`

- `proptests`

The `proptests` function systematically explores various modifications of the input parameters for `proptest` to generate a comprehensive set of proportion tests. If the `hyperloop` parameter is not specified, it will result in the generation of several hundred tests. The function returns a list of different tests, with the first element being the original `proptest` result. If only a specific element of a `proptest` result is of interest, providing the name of the element in `elem` will return all `proptests` where the specified element is different.

```
# Set up a base proportion test
n <- 150
x <- sum(runif(n) < 0.6)
basetest <- proptest_num(x = x, n = n)

# Generate all different tests
all_tests <- proptests(basetest, hyperloop = TRUE)
str(all_tests)
#> List of 1
#>  $ :List of 17
#>   ..$ pi0         : num 0.5
#>   ..$ x           : int 100
#>   ..$ n           : num 150
#>   ..$ alternative : chr "two.sided"
#>   ..$ X           :List of 3
#>   .. ..$ size: num 150
#>   .. ..$ prob: num 0.5
#>   .. ..$ name: chr "binom"
#>   .. ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   .. ..- attr(*, "discrete")= num 1
#>   ..$ Statistic   :List of 3
#>   .. ..$ mean: num 0
```

```
#>   .. ..$ sd  : num 1
#>   .. ..$ name: chr "norm"
#>   .. ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   .. ..- attr(*, "discrete")= num 0
#>   ..$ statistic   : num 4.08
#>   ..$ p.value     : num 4.46e-05
#>   ..$ stderr      : num 0.0408
#>   ..$ binom2norm  : logi TRUE
#>   ..$ alphaexact  : logi NA
#>   ..$ alpha       : num 0.05
#>   ..$ critical    : num [1:2] -1.96 1.96
#>   ..$ acceptance0 : num [1:2] -1.96 1.96
#>   ..$ criticalx   : num [1:2] 63 87
#>   ..$ acceptance0x: num [1:2] 63 87
#>   ..$ accept1     : logi TRUE
#>   ..- attr(*, "class")= chr [1:2] "proptest" "list"

# Generate all different random sampling functions
x_functions <- proptests(basetest, elem = "X", hyperloop = TRUE)
str(x_functions)
#> List of 1
#>  $ :List of 17
#>   ..$ pi0         : num 0.5
#>   ..$ x           : int 100
#>   ..$ n           : num 150
#>   ..$ alternative : chr "two.sided"
#>   ..$ X           :List of 3
#>   .. ..$ size: num 150
#>   .. ..$ prob: num 0.5
#>   .. ..$ name: chr "binom"
#>   .. ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   .. ..- attr(*, "discrete")= num 1
#>   ..$ Statistic   :List of 3
#>   .. ..$ mean: num 0
#>   .. ..$ sd  : num 1
#>   .. ..$ name: chr "norm"
#>   .. ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   .. ..- attr(*, "discrete")= num 0
#>   ..$ statistic   : num 4.08
#>   ..$ p.value     : num 4.46e-05
#>   ..$ stderr      : num 0.0408
#>   ..$ binom2norm  : logi TRUE
#>   ..$ alphaexact  : logi NA
#>   ..$ alpha       : num 0.05
#>   ..$ critical    : num [1:2] -1.96 1.96
#>   ..$ acceptance0 : num [1:2] -1.96 1.96
#>   ..$ criticalx   : num [1:2] 63 87
#>   ..$ acceptance0x: num [1:2] 63 87
#>   ..$ accept1     : logi TRUE
#>   ..- attr(*, "class")= chr [1:2] "proptest" "list"
```

In this example, a base proportion test (`basetest`) is created using a sample size (`n`) and the number of successes (`x`). The `proptests` function is then used to explore various modifications of the input parameters, generating all different tests in the first case and all different random sampling functions in the second case.

- `proptest_data`

Generates data for a binomial test based on specified test properties. This function is particularly useful for simulating scenarios and conducting binomial tests under different conditions.

```
# Generate binomial test data with default settings
data_d <- proptest_data()

# Generate binomial test data with custom settings
data_c <- proptest_data(
  size = 20:50,                          # Vector of sample sizes
  prob = seq(0.1, 0.9, by = 0.2),        # Vector of probabilities
  reject = FALSE,                        # Determines whether the generated data leads to a rejection
  alternative = "less",                  # Specifies the alternative hypothesis, must be "less" or "g
  alpha = 0.05,                          # Vector of significance levels
  norm.approx = TRUE,                    # Specifies whether a normal approximation should be used
  maxit = 500                            # Maximum number of trials
)
str(data_c)
#> List of 5
#>  $ pi0        : num 0.5
#>  $ x          : int 33
#>  $ n          : int 40
#>  $ alpha      : num 0.05
#>  $ alternative: chr "less"
```

- `proptest_num`

Computes results for a test on proportions using either `stats::binom.test()` or a normal approximation without continuity correction. The function accepts named parameters or an argument list with parameters.

- x: Number of successes.
- n: Sample size (default: `sd(x)`).
- pi0: True value of the proportion (default: 0.5).
- alternative: A string specifying the alternative hypothesis (default: “two.sided”; “greater” or “less” can be used).
- alpha: Significance level (default: 0.05).
- binom2norm: Can the binomial distribution be approximated by a normal distribution (default: NA = use `binom2norm` function).

The results may differ from `stats::binom.test()` as `proptest_num` is designed for hand-computed binomial tests. The p-value computed by `stats::binom.test` may not be reliable.

```
# Example with default parameters
n <- 100
x <- sum(runif(n) < 0.4)
result <- proptest_num(x = x, n = n)
str(result)
#> List of 17
#>  $ pi0         : num 0.5
#>  $ x           : int 45
#>  $ n           : num 100
#>  $ alternative : chr "two.sided"
#>  $ X           :List of 3
#>   ..$ size: num 100
#>   ..$ prob: num 0.5
#>   ..$ name: chr "binom"
```

```
#>   ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   ..- attr(*, "discrete")= num 1
#>  $ Statistic   :List of 3
#>   ..$ mean: num 0
#>   ..$ sd  : num 1
#>   ..$ name: chr "norm"
#>   ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   ..- attr(*, "discrete")= num 0
#>  $ statistic   : num -1
#>  $ p.value     : num 0.317
#>  $ stderr      : num 0.05
#>  $ binom2norm  : logi TRUE
#>  $ alphaexact  : logi NA
#>  $ alpha       : num 0.05
#>  $ critical    : num [1:2] -1.96 1.96
#>  $ acceptance0 : num [1:2] -1.96 1.96
#>  $ criticalx   : num [1:2] 40.2 59.8
#>  $ acceptance0x: num [1:2] 40.2 59.8
#>  $ accept1     : logi FALSE
#>  - attr(*, "class")= chr [1:2] "proptest" "list"
```

In this example, the `proptest_num` function is used to compute results for a binomial test with specified parameters. The function returns a list of relevant values, including test statistics, critical values, acceptance intervals, and p-values.

ttests, ttest_data and ttest_num

The `ttest_data` function generates simulated data tailored for a t-test for a single mean, considering specified test properties. This facilitates the exploration of various scenarios and the evaluation of statistical hypotheses related to the mean.

The `ttest_data` function consists of the following arguments:

- size: a numeric vector specifying sample sizes to be generated, calculated as squares of integers ranging from 3 to 20.
- mean: a numeric vector defining potential mean values for the simulated data, ranging from -5 to 5.
- sd: a numeric vector determining standard deviations for the generated data, with values ranging from 0.1 to 1 in increments of 0.1.
- reject: a logical vector that determines whether the generated values of variable x should result in the rejection of the null hypothesis (default is TRUE). If set to NA, this condition will be disregarded.
- alternative: a character vector specifying the alternative hypothesis for the t-test, with options for "two.sided," "less," or "greater."
- alpha: a numeric vector containing significance levels for hypothesis testing, including common values such as 0.01, 0.05, and 0.1.
- z: a numeric vector defining quantiles for the standard normal distribution, used in hypothesis testing; ranges from -4.49 to 4.49 with increments of 0.01.
- use.sigma: a logical value indicating whether the standard deviation (`sigma`) should be used in generating data; default is `TRUE`.

```r
# Generate t-test data
ttest_data_scenario1 <- ttest_data(
  size = c(25, 64, 121),
```

```
  mean = c(0, 2, -2),
  sd = c(0.5, 0.7, 1),
  reject = TRUE,  # Rejection condition
  alternative = "two.sided",
  alpha = c(0.01, 0.05, 0.1),
  z = seq(-3.49, 3.49, by = 0.01),
  use.sigma = TRUE
)
```

In summary, this example represents a situation where we are generating t-test data for three different sample sizes and mean values, with specific rejection conditions. The generated data is tailored for hypothesis testing with a two-sided alternative hypothesis and varying significance levels. The condition reject = TRUE implies that the null hypothesis will be rejected based on the generated data.

- `ttest_num`

Is a function that helps with the computation of all the results for a t-test. We are testing this function with the following exercise that is intended to produce a one-sample t-test. The exercise is meant to assess whether a new variety of butter is worth launching, based on customers' willingness to pay a certain price.

```
sigma  <- sample(5:30, size=1)
ttest  <- ttest_num(n           = sample((4:8)^2, size=1),
                    mu0         = sample(seq(1.5, 3, by=0.1)+0.5, size=1),
                    mean        = sample(seq(1.5, 3, by=0.1), size=1),
                    alternative = 'greater',
                    sd          =  sample((sigma-3:sigma+3), size=1)/10,
                    sigma       = sigma/10,
                    norm        = TRUE)
str(ttest)
#> List of 21
#>  $ mu0         : num 2.7
#>  $ x           : num(0)
#>  $ sigma       : num 1
#>  $ norm        : logi TRUE
#>  $ mean        : num 1.8
#>  $ sd          : num 0.8
#>  $ n           : num 25
#>  $ alternative : chr "greater"
#>  $ Xbar        :List of 3
#>   ..$ mean: num 2.7
#>   ..$ sd  : num 0.2
#>   ..$ name: chr "norm"
#>   ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   ..- attr(*, "discrete")= num 0
#>  $ Statistic   :List of 3
#>   ..$ mean: num 0
#>   ..$ sd  : num 1
#>   ..$ name: chr "norm"
#>   ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   ..- attr(*, "discrete")= num 0
#>  $ statistic   : num -4.5
#>  $ p.value     : num 1
#>  $ stderr      : num 0.2
#>  $ n.clt       : num 30
#>  $ t2norm      : logi FALSE
```

```
#>  $ critical    : num 1.64
#>  $ acceptance0 : num [1:2] -Inf 1.64
#>  $ criticalx   : num 3.03
#>  $ acceptance0x: num [1:2] -Inf 3.03
#>  $ alpha       : num 0.05
#>  $ accept1     : logi FALSE
#>  - attr(*, "class")= chr [1:2] "ttest" "list"
```

The exercise is set in the context of a butter manufacturer considering the launch of a new butter variety. To determine whether it's worth launching, the manufacturer wants to know if customers are willing to pay at least a specific price per pack of the new butter. This is why we use the `ttest_num` function, in order to make an informed decision with the help of a t-test.

`ttest_num` computes all the results of the t-test as we can observe:

- `n`: The sample size, representing the number of customers randomly selected for the survey.
- `mu0`: The price the manufacturer intends to test as its objective.
- `mean`: The average spending level of the sample's respondents.
- `alternative`: The alternative hypothesis, set to 'greater,' indicating that the manufacturer is interested in testing whether customers are willing to pay more than the target price.
- `sd`: The sample standard deviation, which reflects the range of prices that customers are ready to accept.
- `sigma`: The population standard deviation, representing the standard deviation of prices in the entire population (unknown by default).
- `alpha`: The significance level (set to 0.05).
- `ttests`

The `ttests` function systematically explores various modifications of the input parameters for t-tests, generating a comprehensive set of possible t-tests. Details regarding the specific parameter values employed can be found below. It is important to note that omitting the hyperloop parameter may result in the generation of approximately 5000 t-tests. The function returns only distinct t-tests, with the primary t-test stored as the first element. If there is interest in a specific element of the t-test, users can specify it using the elem parameter, and the function will return all t-tests where that particular element differs.

```r
# Generate a base t-test
base_ttest <- ttest_num(mean = 1.2, sd = 0.8, n = 30, sigma = 1)

# Vary the parameters for hyperloop
hyperloop_variation <- list(
  mean = c(base_ttest$mean - 0.5, base_ttest$mean, base_ttest$mean + 0.5),
  n = c(20, 30, 40),
  sd = c(0.7, 0.8, 0.9)
)

# Obtain different t-tests with varied parameters
different_ttests <- ttests(base_ttest, hyperloop = hyperloop_variation)

# Extract t-tests where the element "Conf.Int" differs
confint_differing_ttests <- ttests(base_ttest, "Conf.Int", hyperloop = hyperloop_variation)
```

- We start by generating a base t-test (`base_ttest`) with specified parameters such as mean, standard deviation, sample size, and population standard deviation using the `ttest_num` function.

- The `hyperloop_variation` parameter is utilized to systematically vary the mean, sample size, and standard deviation in different scenarios.
- The `ttests` function is then employed to generate distinct t-tests by modifying the base t-test with the specified variations. The resulting t-tests are stored in the variable `different_ttests`.
- Additionally, the function is called again, this time focusing on the specific element "Conf.Int," and returning t-tests where this element differs. The results are stored in the variable `confint_differing_ttests`.

This example demonstrates how the `ttests` function can be applied to explore various t-tests by systematically varying parameters, and it highlights the flexibility of extracting t-tests based on specific elements of interest.

Mathematical Computations

Intervals

`dbl`, `pos` and `neg`

The `pos`, `neg`, and `dbl` functions are designed to generate intervals based on powers of ten.

- `pos(pow)`: Generates positive intervals based on powers of ten.
- `neg(pow)`: Generates negative intervals based on powers of ten.
- `dbl(pow)`: Generates intervals that include both positive and negative values based on powers of ten.

```
# Generate double intervals
result_1 <- dbl(2)
print(result_1)
#> [1] -100  100

# Generate positive intervals
result_2 <- pos(3)
print(result_2)
#> [1]    0 1000

# Generate negative intervals
result_3 <- neg(3)
print(result_3)
#> [1] -1000     0
```

Polynomials

`monomial`

The `monomial` function constructs a polynomial in the form of $c \cdot x^d$, where c is the coefficient and d is the degree. The default values are set to create a monomial of degree 1 with a coefficient of 1.

```
degree <- 3
coefficient <- 2

# Generate a monomial with the specified degree and coefficient
result_monomial <- monomial(3, 2)
cat("Monomial:", result_monomial, "\n")
#> Monomial: 0 0 0 2
```

In this example, the `monomial` function is utilized to create a monomial with a degree of 3 and a coefficient of 2. The resulting monomial $2 \cdot x^3$ is then printed using the `cat` function.

pminimum

The `pminimum` function calculates the minimum value of a polynomial within a specified interval $[lower, upper]$. It evaluates the polynomial at critical points within the given interval, including the interval's boundaries, and returns the minimum value.

```
# Creating a polynomial and finding the minimum within a specified range
custom_polynomial <- polynomial(c(2, -1, 4, -2))  # Represents 2x^3 - x^2 + 4x - 2

# Finding the minimum of the polynomial within the range [-1, 2]
minimum_result <- pminimum(custom_polynomial, -1, 2)

# Displaying the result
print(minimum_result)
#> [1] 0
```

In this example, a custom polynomial `custom_polynomial` is created using the `polynomial` function with coefficients `c(2, -1, 4, -2)`, representing the polynomial $2x^3 - x^2 + 4x - 2$. The `pminimum` function is then applied to find the minimum value of the polynomial within the specified range $[-1, 2]$. The result is stored in `minimum_result`, and represents the minimum value of the polynomial within the given range.

Rational Approximation

fractions and is_terminal

To overcome the rounding problem there is a simple approach: try to use (terminal) fractions. A terminal fraction generates a number with a finite number of digits, for example $\frac{1}{10} = 0.1$. The command `fractions` calls simply `MASS::fractions()` to avoid explicitly loading the library `MASS`. The result of calling fractions has an attribute `fracs` which contains a (approximate) fraction as $\frac{numerator}{denominator}$ representation.

```
x <- c(1/5, 1/6)
x
#> [1] 0.2000000 0.1666667
fractions(x)
#> [1] 1/5 1/6
str(fractions(x))
#>  'fractions' num [1:2] 0.2 0.167
#>  - attr(*, "fracs")= chr [1:2] "1/5" "1/6"
```

Therefore, `is_terminal` tests if all entries are terminal fractions which means the denominators must be dividable by two and five only.

```
x <- c(1/5, 1/6)
is_terminal(x)
#> [1]  TRUE FALSE
```

Unfortunately, we use a decimal numeral system limiting the number possible of denominators which lead to terminal numbers; the ancient babylonian cultures using a sexagesimal numeral system had a larger number of denominators which would lead to terminal numbers.

- `fractions`

`fractions` is a copy of `MASS::fractions` to compute from a numeric values fractions.

```
# Create a 5x5 matrix with random values
Y <- matrix(runif(25), 5, 5)

# Display the matrix as fractions using the `fractions` function
fractions(Y)
```

```
#>      [,1]              [,2]              [,3]              [,4]
#> [1,]       14017/20171 24571988/37001831           628/893     165311/167571
#> [2,]        1297/10090    784163/1569929       15275/52563        7574/10905
#> [3,]         1839/2950         2441/3041         3119/4123       11136/20027
#> [4,]         3605/6673 21959656/63138467         3507/7942          507/1828
#> [5,]         3045/3883        3783/10385         2266/6569       9131/184810
#>      [,5]
#> [1,]   2594040/5306581
#> [2,]         3371/4009
#> [3,]         1637/3191
#> [4,]     277724/312045
#> [5,]       23863/45908

# Perform matrix operations and display the results as fractions
fractions(solve(Y, Y/5))
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] 1/5    0    0    0    0
#> [2,]   0  1/5    0    0    0
#> [3,]   0    0  1/5    0    0
#> [4,]   0    0    0  1/5    0
#> [5,]   0    0    0    0  1/5
fractions(solve(Y, Y/5) + 1)
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] 6/5    1    1    1    1
#> [2,]   1  6/5    1    1    1
#> [3,]   1    1  6/5    1    1
#> [4,]   1    1    1  6/5    1
#> [5,]   1    1    1    1  6/5
```

Solving Equations

equal

Compares two numeric values if they are equal given a tolerance (default: 1e-6).

```
x <- pi
y <- pi+1e-4
equal(x, y)
#> [1] FALSE
equal(x, y, tol=1e-3)
#> [1] TRUE
```

equations

The equations function is used to define a set of equations using the formula interface. It also provides a LaTeX representation of the formulae. The resulting equations object includes information about the type of equation, its value, associated text, and the interval if applicable.

```
# Defining a system of economics equations

econ_eq <- equations(
  Y ~ C + I + G + (X - M), "Y = C + I + G + (X - M)",
  C ~ c0 + c1*YD, "C = c_0 + c_1\\cdot YD",
  I ~ I0 - i1*r + i2*Y, "I = I_0 - i_1\\cdot r + i_2\\cdot Y",
  YD ~ Y - T, "YD = Y - T",
```

```
  T ~ t0 + t1*Y, "T = t_0 + t_1\\cdot Y",
  M ~ m0 + m1*Y, "M = m_0 + m_1\\cdot Y",
  X ~ x0 + x1*Y, "X = x_0 + x_1\\cdot Y",
  r ~ r0, "r = r_0"
)
print(econ_eq)
#>        type                      value                                text
#> 1  Equation  C + I + G + (X - M) - Y          Y = C + I + G + (X - M)
#> 2  Equation         c0 + c1 * YD - C           C = c_0 + c_1\\cdot YD
#> 3  Equation I0 - i1 * r + i2 * Y - I I = I_0 - i_1\\cdot r + i_2\\cdot Y
#> 4  Equation               Y - T - YD                       YD = Y - T
#> 5  Equation          t0 + t1 * Y - T           T = t_0 + t_1\\cdot Y
#> 6  Equation          m0 + m1 * Y - M           M = m_0 + m_1\\cdot Y
#> 7  Equation          x0 + x1 * Y - X           X = x_0 + x_1\\cdot Y
#> 8  Equation                   r0 - r                           r = r_0
#> 9         C                    <NA>                               C
#> 10        I                    <NA>                               I
#> 11        G                    <NA>                               G
#> 12        X                    <NA>                               X
#> 13        M                    <NA>                               M
#> 14        Y                    <NA>                               Y
#> 15       c0                    <NA>                              c0
#> 16       c1                    <NA>                              c1
#> 17       YD                    <NA>                              YD
#> 18       I0                    <NA>                              I0
#> 19       i1                    <NA>                              i1
#> 20        r                    <NA>                               r
#> 21       i2                    <NA>                              i2
#> 22        T                    <NA>                               T
#> 23       t0                    <NA>                              t0
#> 24       t1                    <NA>                              t1
#> 25       m0                    <NA>                              m0
#> 26       m1                    <NA>                              m1
#> 27       x0                    <NA>                              x0
#> 28       x1                    <NA>                              x1
#> 29       r0                    <NA>                              r0
#>            interval
#> 1              <NA>
#> 2              <NA>
#> 3              <NA>
#> 4              <NA>
#> 5              <NA>
#> 6              <NA>
#> 7              <NA>
#> 8              <NA>
#> 9  [-1e+05 , 1e+05]
#> 10 [-1e+05 , 1e+05]
#> 11 [-1e+05 , 1e+05]
#> 12 [-1e+05 , 1e+05]
#> 13 [-1e+05 , 1e+05]
#> 14 [-1e+05 , 1e+05]
#> 15 [-1e+05 , 1e+05]
#> 16 [-1e+05 , 1e+05]
```

```
#> 17 [-1e+05 , 1e+05]
#> 18 [-1e+05 , 1e+05]
#> 19 [-1e+05 , 1e+05]
#> 20 [-1e+05 , 1e+05]
#> 21 [-1e+05 , 1e+05]
#> 22 [-1e+05 , 1e+05]
#> 23 [-1e+05 , 1e+05]
#> 24 [-1e+05 , 1e+05]
#> 25 [-1e+05 , 1e+05]
#> 26 [-1e+05 , 1e+05]
#> 27 [-1e+05 , 1e+05]
#> 28 [-1e+05 , 1e+05]
#> 29 [-1e+05 , 1e+05]
```

In this example, the equations represent components of the Keynesian aggregate expenditure model, where Y is the national income, C is consumption, I is investment, G is government spending, X is exports, and M is imports. The model includes consumption functions, investment functions, taxation, and trade balance.

`print.equations`

The `print.equations` function serves as an S3 method designed for displaying an equations object containing equations and associated variables. Internally, it generates a data frame, providing a clear representation of the equations and their dependencies.

```
# The equations describe the formulae for an confidence interval of the mean
e <- equations(o~x+c*s/sqrt(n), "v_o=\\bar{x}+c\\cdot\\frac{s^2}{n}",
               u~x-c*s/sqrt(n), "v_u=\\bar{x}-c\\cdot\\frac{s^2}{n}",
               e~c*s/sqrt(n),   "e  =c\\cdot\\frac{s^2}{\\sqrt{n}}",
               l~2*e,           "l  =2\\cdot e"
               )
print(e)
#>        type                value                                  text
#> 1  Equation x + c * s/sqrt(n) - o v_o=\\bar{x}+c\\cdot\\frac{s^2}{n}
#> 2  Equation x - c * s/sqrt(n) - u v_u=\\bar{x}-c\\cdot\\frac{s^2}{n}
#> 3  Equation     c * s/sqrt(n) - e  e  =c\\cdot\\frac{s^2}{\\sqrt{n}}
#> 4  Equation             2 * e - l                       l  =2\\cdot e
#> 5         x                  <NA>                                   x
#> 6         c                  <NA>                                   c
#> 7         s                  <NA>                                   s
#> 8         n                  <NA>                                   n
#> 9         o                  <NA>                                   o
#> 10        u                  <NA>                                   u
#> 11        e                  <NA>                                   e
#> 12        l                  <NA>                                   l
#>            interval
#> 1              <NA>
#> 2              <NA>
#> 3              <NA>
#> 4              <NA>
#> 5  [-1e+05 , 1e+05]
#> 6  [-1e+05 , 1e+05]
#> 7  [-1e+05 , 1e+05]
#> 8  [-1e+05 , 1e+05]
#> 9  [-1e+05 , 1e+05]
#> 10 [-1e+05 , 1e+05]
```

```
#> 11 [-1e+05 , 1e+05]
#> 12 [-1e+05 , 1e+05]
```

In this example, a set of equations is defined to describe the formulae for a confidence interval of the mean. Let's break down the code and understand each part:

- The `equations` function is used to create an equations object (`e`).
- Four equations are defined in terms of variables (`o`, `u`, `e`, `l`) and involve the variables `x`, `c`, `s`, and `n`.
- Each equation is provided in a formula style, representing a statistical formula related to a confidence interval of the mean.
- The `print` function is used to display the equations object (`e`).
- The output presents the equations and associated variables in a structured format. This example demonstrates how the `equations` function can be utilized to create a set of equations representing statistical formulas.

variables

Is a function that allows the configuration of values, LaTeX representations, and solution intervals for variables within an `equations` object. The first argument must be the `equations` object, followed by named parameters to specify values, intervals, and LaTeX representations for specific variables. This function enables the modification of the `equations` object to incorporate specific variable information.

```
# The equations describe the formulae for a confidence interval of the mean
e <- equations(o~x+c*s/sqrt(n), "v_o=\\bar{x}+c\\cdot\\frac{s^2}{n}",
               u~x-c*s/sqrt(n), "v_u=\\bar{x}-c\\cdot\\frac{s^2}{n}",
               e~c*s/sqrt(n),   "e  =c\\cdot\\frac{s^2}{\\sqrt{n}}",
               l~2*e,           "l  =2\\cdot e"
               )

# Set variable values, intervals, and LaTeX representations
e <- variables(e,
               x=0,    "\\bar{x}",
               c=2.58, dbl(2),
               s=1,    pos(5), "s^2",
               n=25,   pos(5),
               l=pos(5),
               e=pos(5),
               u="v_u", o="v_o")

# Print the modified equations object
print(e)
#>        type                value                                  text
#> 1  Equation x + c * s/sqrt(n) - o v_o=\\bar{x}+c\\cdot\\frac{s^2}{n}
#> 2  Equation x - c * s/sqrt(n) - u v_u=\\bar{x}-c\\cdot\\frac{s^2}{n}
#> 3  Equation     c * s/sqrt(n) - e  e  =c\\cdot\\frac{s^2}{\\sqrt{n}}
#> 4  Equation             2 * e - l                      l  =2\\cdot e
#> 5         x                     0                           \\bar{x}
#> 6         c                  2.58                                 c
#> 7         s                     1                               s^2
#> 8         n                    25                                 n
#> 9         o                  <NA>                               v_o
#> 10        u                  <NA>                               v_u
#> 11        e                  <NA>                                 e
#> 12        l                  <NA>                                 l
#>            interval
```

```
#> 1              <NA>
#> 2              <NA>
#> 3              <NA>
#> 4              <NA>
#> 5  [-1e+05 , 1e+05]
#> 6      [-100 , 100]
#> 7       [0 , 1e+05]
#> 8       [0 , 1e+05]
#> 9  [-1e+05 , 1e+05]
#> 10 [-1e+05 , 1e+05]
#> 11      [0 , 1e+05]
#> 12      [0 , 1e+05]
```

The provided R example involves creating a set of equations representing the formulae for a confidence interval of the mean, including variables such as `o`, `u`, `e`, and `l`. Subsequently, the `variables` function is applied to set specific values, intervals, and LaTeX representations for these variables. For instance, `x` is assigned a value of 0, `c` is set to 2.58 with an interval of [1, 2], and the LaTeX representation for `s` is defined as "s^2". The modified equations object is then printed, showcasing the customized variable settings and representations. This approach demonstrates efficient manipulation and customization of mathematical expressions within the R environment.

num_solve

The `num_solve` function is designed to compute the value of a target variable in a set of equations. The equations, representing relationships between variables, are transformed into root-finding problems, and the function attempts to find the roots using the `stats::uniroot()` function. If successful, the computed value of the target variable is returned; otherwise, `numeric(0)` is returned. If the target variable is not specified (`target==''`), the function returns all computed values and steps. The `compute` attribute contains a data frame with information about the computation steps.

```
# The equations describe the formulae for an confidence interval of the mean
e <- equations(o~x+c*s/sqrt(n), "v_o=\\bar{x}+c\\cdot\\frac{s^2}{n}",
               u~x-c*s/sqrt(n), "v_u=\\bar{x}-c\\cdot\\frac{s^2}{n}",
               e~c*s/sqrt(n),   "e  =c\\cdot\\frac{s^2}{\\sqrt{n}}",
               l~2*e,           "l  =2\\cdot e"
               )
# Setting variables and their values
e <- variables(e, x = 0, c = 2.58, s = 1, n = 25, l = pos(5), e = pos(5), u = "v_u", o = "v_o")

# Finding confidence interval length ('l')
ns <- num_solve('l', e)

# Computing all possible values
ns <- num_solve('', e)
print(ns)
#> $x
#> [1] 0
#>
#> $c
#> [1] 2.58
#>
#> $s
#> [1] 1
#>
#> $n
```

```
#> [1] 25
#>
#> $o
#> [1] 0.516
#>
#> $u
#> [1] -0.516
#>
#> $e
#> [1] 0.516
#>
#> $l
#> [1] 1.032
#>
#> attr(,"latex")
#>  [1] "\\begin{align*}"
#>  [2] "& x  =  0\\\\"
#>  [3] "& c  =  2.58\\\\"
#>  [4] "& s  =  1\\\\"
#>  [5] "& n  =  25\\\\"
#>  [6] "v_o=\\bar{x}+c\\cdot\\frac{s^2}{n}  & \\rightarrow  v_o  =  0.5160000000017812\\\\"
#>  [7] "v_u=\\bar{x}-c\\cdot\\frac{s^2}{n}  & \\rightarrow  v_u  =  -0.5160000000017812\\\\"
#>  [8] "e  =c\\cdot\\frac{s^2}{\\sqrt{n}}  & \\rightarrow  e  =  0.516\\\\"
#>  [9] "l  =2\\cdot e  & \\rightarrow  l  =  1.032\\\\"
#> [10] "\\end{align*}"
#> attr(,"compute")
#>   variable  value               formula
#> 1        x  0.000                  <NA>
#> 2        c  2.580                  <NA>
#> 3        s  1.000                  <NA>
#> 4        n 25.000                  <NA>
#> 5        o  0.516 x + c * s/sqrt(n) - o
#> 6        u -0.516 x - c * s/sqrt(n) - u
#> 7        e  0.516     c * s/sqrt(n) - e
#> 8        l  1.032             2 * e - l
#> attr(,"class")
#> [1] "equation_solve"
```

In this example, the function is used to find the confidence interval length (`l`) based on a set of equations and variable values. Here, the function is also used to compute all possible values for the variables specified in the equations. In both cases, the resulting `ns` object contains information about the computation, including the values of variables and computation steps. The `compute` attribute provides a data frame with details about each variable's value in the computation process.

Value and Extremes Analysis

`extremes`

Calculates the extrema of real values, including minima, maxima, and saddle points, for a univariate polynomial. The computation can be tailored to focus on specific categories of extrema.

```
p <- polynomial(c(0,0,0,1))
extremes(p)
#> [1] 0
#> attr(,"type")
```

```
#> [1] "saddle"
```

nearest_arg

nearest_arg is a function designed to identify the closest candidate value for each element in the input argument (arg). This function serves as an enhanced alternative to the base R function match.arg, offering improved tolerance for potential typographical errors. However, it's important to note that while nearest_arg enhances error resilience, detecting an incorrect choice may be challenging if one occurs.

```
# Sample usage of nearest_arg
valid_colors <- c("red", "blue", "green", "yellow", "orange")

# Input color names with potential typos
input_colors <- c("rad", "blu", "grien", "yello", "ornge")

# Applying nearest_arg to find the closest valid color names
result_colors <- nearest_arg(input_colors, valid_colors)

# Displaying the result
cat("Input Colors:", input_colors)
#> Input Colors: rad blu grien yello ornge
cat("Nearest Valid Colors:", result_colors)
#> Nearest Valid Colors: red blue green yellow orange
```

- valid_colors: A vector representing the valid color names.
- input_colors: A vector containing color names with potential typos or deviations.
- result_colors: The output of nearest_arg applied to input_colors and valid_colors.

In this example, nearest_arg is utilized to identify the nearest valid color name for each input color. The function demonstrates its effectiveness in handling potential typos or variations in the input color names. The result provides a vector of the nearest valid color names, showcasing how nearest_arg enhances error tolerance and accurately identifies the closest valid candidates in a given set.

unique_max

Checks if the numeric vector x possesses a singular maximum. This function evaluates whether the discrepancy between the largest and second-largest values in x is greater than a specified minimum distance, tol.

```
# Generate a vector with a unique maximum
vec_unique_max <- c(3, 7, 5, 2, 8, 6, 4)

# Check if vec_unique_max has a unique maximum with the default tolerance (1e-3)
result_default_tol <- unique_max(vec_unique_max)

# Check if vec_unique_max has a unique maximum with a larger tolerance (1)
result_large_tol <- unique_max(vec_unique_max, tol = 1)

# Print the results
cat("Default Tolerance Result:", result_default_tol, "\n")
#> Default Tolerance Result: TRUE
cat("Large Tolerance Result:", result_large_tol, "\n")
#> Large Tolerance Result: FALSE
```

Exercise Generation

Structured Exercise Development

all_different

For solutions in multiple choice exercises you want to ensure that the numerical results are not too near to each other. Therefore, `all_different` checks if the differences between the entries in `obj` are larger than some given value `tol`.

```
x <- runif(20)
all_different(x, 1)    # Minimal distance is at least 1
#> [1] FALSE
all_different(x, 1e-4) # Minimal distance is at least 0.0001
#> [1] TRUE
```

calledBy

Checks if the call stack, obtained from `base::sys.calls`, contains a call from the specified function (`fun`).

```
# Define functions funa and funb
funb <- function() { calledBy('funa') }
funa <- function() { funb() }

# Call funa and check if it is called by funb
result <- funa()

# Display the result
str(result)
#>  logi TRUE
```

exercise

The `exercise` function is used to create and modify a data structure for exercise data. `exer` represents an existing exercise data structure or NULL to create a new one.

```
# Create a new exercise data structure
exer <- exercise()

# Add a parameter 'x' to the exercise data structure
exer <- exercise(exer, x = 3)
str(exer)
#> List of 1
#>  $ x: num 3
#>  - attr(*, "class")= chr [1:2] "exercise" "list"
```

Solution Handling and Result Formatting

solutions

- `sol_num` generates a numerical solution object for a given numeric value. The function automatically determines tolerance if not provided, considering the range of values. Additionally, it captures relevant information about the source context, including the script's name or file path.

```
# Example 1: Calculating a solution with default parameters
s <- sol_num(sqrt(2))
str(s)
#> List of 6
```

```
#>  $ type     : chr "num"
#>  $ x        : num 1.41
#>  $ solution : chr "1.41421356237309"
#>  $ digits   : num 3
#>  $ tolerance: num 0.00141
#>  $ name     : chr "exams.forge.Rmd"
#>  - attr(*, "class")= chr [1:2] "solution" "list"
# Example 2: Numeric solution with tolerance and rounding
sol_num(pi, tol=0.001, digits=3)
#> $type
#> [1] "num"
#>
#> $x
#> [1] 3.141593
#>
#> $solution
#> [1] "3.14159265358979"
#>
#> $digits
#> [1] 3
#>
#> $tolerance
#> [1] 0.001
#>
#> $name
#> [1] "exams.forge.Rmd"
#>
#> attr(,"class")
#> [1] "solution" "list"
```

- `sol_int` extends the functionality of the `sol_num` function by rounding the given numeric value to the nearest integer. It generates an integer solution object with optional parameters for tolerance and rounding digits.

```
# Example: Creating an integer solution
integer_solution <- sol_int(7.89, tol=0.01, digits=2)
str(integer_solution)
#> List of 6
#>  $ type     : chr "num"
#>  $ x        : num 8
#>  $ solution : chr "8"
#>  $ digits   : num 2
#>  $ tolerance: num 0.01
#>  $ name     : chr "exams.forge.Rmd"
#>  - attr(*, "class")= chr [1:2] "solution" "list"
```

- `sol_mc` generates a multiple-choice solution object by combining false (x) and true (y) answers. The number of false and true answers to include can be altered, shuffling options can be specified, and a default option when none of the choices apply can be provided. The resulting solution object captures the answer list, solution indicators, and relevant source context information.

```
# Example: Creating a multiple-choice solution for a biology quiz
plants <- c("Moss", "Fern", "Pine", "Rose", "Tulip")
flowering_plants <- c("Rose", "Tulip")
non_flowering_plants <- setdiff(plants, flowering_plants)
```

```
s_plants <- sol_mc(non_flowering_plants, flowering_plants, sample=c(2, 2), shuffle=FALSE, none="None of
str(s_plants)
#> List of 4
#>  $ type    : chr "mchoice"
#>  $ answer  : chr [1:5] "Fern" "Moss" "Tulip" "Rose" ...
#>  $ solution: logi [1:5] FALSE FALSE TRUE TRUE FALSE
#>  $ name    : chr "exams.forge.Rmd"
#>  - attr(*, "class")= chr [1:2] "solution" "list"
```

- `sol_ans` extracts the answer list from a multiple-choice solution object created using the `sol_mc` function. It facilitates the presentation of correct and potential answer choices in various formats, including LaTeX for exams2pdf compatibility.

```
# Example: Extracting correct answers from a biology quiz
s <- sol_mc(c("Oak", "Maple", "Rose"), c("Tulip", "Sunflower"), sample=c(2, 1), none="No valid options"
sol_ans(s)
#> [1] "\nAnswerlist\n----------\n* Oak\n* Rose\n* Tulip\n* No valid options\n"
```

- `sol_tf` extracts the solution list (True or False) from a multiple-choice solution object created using the `sol_mc` function. It facilitates the presentation of binary representations of correct and incorrect choices in various formats, including LaTeX for exams2pdf compatibility.

```
# Example: Extracting True/False solutions from a chemistry quiz
s <- sol_mc(c("Copper", "Silver", "Gold"), c("Oxygen", "Carbon"), sample=c(2, 1), none="None of the abo
sol_tf(s)
#> [1] "\nAnswerlist\n----------\n* False\n* True\n* False\n* False\n"
```

- `sol_info` generates a Meta-Information block for a given solution object. It provides additional context and details about the solution, including its type, solution values, tolerance, and source context.

```
# Example: Displaying Meta-Information for a statistical analysis
stat_analysis <- sol_num(mean(c(5, 8, 12, 15, 18)), tol = 0.01, digits = 2)
info_stat <- sol_info(stat_analysis)
cat(info_stat)
#>
#> Meta-information
#> ================
#> extype: num
#> exsolution: 11.6
#> extol: 0.01
#> exname: exams.forge.Rmd
```

`int_result` and `num_result`

`num_result` is a function that generates a list containing various elements for numeric results. The key components of this list include:

- `x`: The original numeric values.
- `fx`: The rounded values with the `exams::fmt()` function, represented as characters.
- `tolerance`: The specified tolerance for rounding.
- `digits`: The number of digits used for rounding.

It's important to note that `x` can contain more than one numeric value, and in such cases, ensure using `...$x[1]` for numeric exercises.

If `digits` are not explicitly provided and `length(x) > 1`, the function calculates `ceiling(-log10(min(diff(sort(x)), na.rm=TRUE)))`. If `digits` are not provided and `length(x) == 1`, it uses `3 + ceiling(-log10(abs(x)))`.

If no tolerance is specified, `tolmult * 10^(1 - digits)` is employed.

Additionally, the auxiliary function `int_result` can be used when the result is an integer number. It calls `num_result(x, 0, 0.1, 1, ...)` with a tolerance of 0.1.

As for the exercise provided, it involves generating random values for variables such as `hours`, `lambda`, `busses`, and `border`. The exercise utilizes the exponential distribution and aims to create a scenario related to waiting times for buses. The `num_result` and `int_result` functions are then employed to format and round the results appropriately for use in statistical exercises. The overall goal is to create a dynamic and varied set of exercises with numerical outcomes based on the specified parameters.

```
# Exercise "Bluthochdruck"

alpha <- sample(c(0.01, 0.02, 0.05, 0.1, 0.2), 1)
n     <- sample(5:15, 1)
smean <- 80:160
ssig  <- 1:50
ski   <- sample(smean,1)
sigma <- sample(ssig,1)
a <- ski-sigma
b <- ski+sigma
X <- sample(seq(a,b,1),n,replace=TRUE)
#part a
xBar <- round(mean(X))
s2   <- var(X)
s2   <- round(s2)
s    <- round(sqrt(s2),2)
#part c
c    <- round(qt(1-alpha/2, n-1), 3)
v_u  <- xBar - c * sqrt(s2/n)
v_o  <- xBar + c * sqrt(s2/n)
dig  <- 1-floor(log10((c-qnorm(1-alpha/2))*sqrt(s2/n)))
sc   <- num_result(v_u, digits=dig, tolmult=1)
print(sc)
#> $x
#> [1] 66.41807
#>
#> $fx
#> [1] "66.4"
#>
#> $tolerance
#> [1] 1
#>
#> $digits
#> [1] 1
```

This example demonstrates how to generate random data, perform statistical calculations, and use the `num_result` function to obtain a numerical result for a confidence interval. The focus is on rounding precision and tolerance. Here the `num_result` function is called with the upper confidence limit `v_u`, specifying the desired precision (digits) and a tolerance multiplier (`tolmult`).

File Manipulation and Document Enhancement

makekey

The `makekey` function generates a character key from a vector of integers. It takes a numeric vector `index` as input and converts each element into a character, creating a comma-separated string representation of the indices.

```
makekey(c(3, 7, 10))
#> [1] "3, 7, 10"
```

- The function `makekey` is applied to the numeric vector `c(3, 7, 10)`.
- Each numeric value in the vector is converted to a character.
- The resulting characters are then joined into a single string, separated by commas.
- In this specific example, `makekey(c(3, 7, 10))` generates the key "3, 7, 10".

moodle_m2s

The `moodle_m2s` function addresses a limitation in the `exams` package by enabling support for multiple-choice questions with multiple correct answers, a feature allowed by Moodle but not directly supported by exams. This function processes an XML file created by `exams.forge`, specifically adapting the representation of multiple-choice questions:

- Changes `<single>...</single>` to `<single>true</single>`
- Adjusts the `fraction` attribute in `<answer fraction="...">...</answer>` tags. If the fraction is less than 0, it is set to zero, and if it's greater than 0, it is set to

  100.

If the file does not have a .xml extension, .xml is appended. Finally, the modified XML code is saved in `newfile`.

```
# Modifying a Moodle XML file for multiple-choice questions with multiple correct answers

# Example 1: Using moodle_m2s on a specified file
# Assuming 'my_moodle_file.txt' is the original Moodle XML file
# original_file <- "my_moodle_file.txt"

# Applying moodle_m2s to modify the XML file
# modified_file <- moodle_m2s(original_file)

# Displaying the name of the modified XML file
# cat("Example 1: Modified XML file saved as:", modified_file, "\n")

# Example 2: Using moodle_m2s on a file from the exams.moodle package
# if (interactive()) {
# Creating a temporary file with .xml extension
# newfile <- tempfile(fileext=".xml")
# Using moodle_m2s on the 'klausur-test.xml' file from the exams.forge package
# moodle_m2s(system.file("xml", "klausur-test.xml", package="exams.forge"), newfile=newfile)

# Opening the modified XML file for editing with file.edit(newfile) }
```

In the first example, the `moodle_m2s` function is applied to address the limitation in the exams package regarding multiple-choice questions with multiple correct answers. The original Moodle XML file is assumed to be named `my_moodle_file.txt`. The function processes this file, making necessary adjustments such as changing `<single>...</single>` to `<single>true</single>`. It also adjusts the fraction attribute in `<answer fraction="...">...</answer>` tags, ensuring that it is set to zero if less than 0 and set to 100 if

greater than 0. The modified XML code is then saved in a new file, and the name of the modified XML file is printed. It's important to note that the function automatically appends `.xml` to the file name if it does not already have a `.xml` extension.

The second example demonstrates the interactive use of the `moodle_m2s` function. It creates a temporary file with a `.xml` extension and applies the function to the `klausur-test.xml` file from the `exams.forge` package. The modified XML file is then opened for editing using `file.edit`. If run interactively, the modifications made by the function can also be viewed and edited.

spell

The `spell` function conducts a spell check on RMarkdown files while selectively disregarding specified keywords commonly used in the context of `exams`. This is achieved through the utilization of the `spelling::spell_check_files()` function.

```
# Perform spell check on an RMarkdown file, ignoring specific keywords
# spell_result <- spell("path/to/my/file.Rmd")

# Alternatively, perform spell check on multiple files
# spell_result_multiple <- spell(c("path/to/file1.Rmd", "path/to/file2.Rmd"))

# Display the spell check results
# print(spell_result)
```

In this example: - The `spell` function is used to conduct a spell check on an RMarkdown file located at "path/to/y/file.Rmd" while ignoring specified keywords common in `exams`.

- Alternatively, the function is applied to multiple files by passing a vector of file paths.
- The results of the spell check are stored in the `spell_result` and `spell_result_multiple` variables.

String Manipulation

Conditional String Output

catif

Calls `cat` if the specified condition (`cond`) is TRUE.

```
# Call catif with TRUE condition
catif(TRUE, "PDF")
#> PDF

# Call catif with FALSE condition
catif(FALSE, "Moodle")  # There is no output with this condition
```

nosanitize

The `nosanitize` function allows us to bypass any sanitation procedures on character vectors. It is designed for situations where no additional sanitization or modification of strings is required, providing us with direct access to the original unaltered data.

```
original_strings <- c("Hello, World!", "<script>alert('Danger!');</script>", "1234567890")
# Applying nosanitize to preserve original strings
unsanitized_strings <- nosanitize(original_strings)
print(unsanitized_strings)
#> [1] "Hello, World!"                       "<script>alert('Danger!');</script>"
#> [3] "1234567890"
```

In this example, the `nosanitize` function is used to process a vector of strings (`original_strings`) without performing any sanitation.The resulting `unsanitized_strings` vector preserves the original content, including any potentially unsafe characters or HTML tags.

Number to String Conversion

`fcvt`

The `fcvt` function converts a numeric vector to a string containing either a floating-point or a fractional number. It is particularly useful for representing repeating or recurring decimals as rational numbers. The function supports various options for controlling the output format.

- `x`: Numeric vector to be converted.
- `nsmall`: Number of decimal places for floating-point numbers.
- `plus`: Logical, indicating whether to include a plus sign for positive numbers.
- `denom`: Integer controlling the output format:
  - If negative, always decimal point numbers are used (default).
  - If zero, a mix of decimal point and fractional numbers are used (whichever is shorter).
  - If one, fractional numbers are used except for integers.
  - If larger than one, the denominator is set to `denom` if possible.

```
# Example 1
x3 <- c((0:16)/8, 1/3)
fcvt(x3)
#>  [1] "0"                "0.125"            "0.25"
#>  [4] "0.375"            "0.5"              "0.625"
#>  [7] "0.75"             "0.875"            "1"
#> [10] "1.125"            "1.25"             "1.375"
#> [13] "1.5"              "1.625"            "1.75"
#> [16] "1.875"            "2"                "0.333333333333333"

# Example 2
fcvt(x3, denom=0)
#>  [1] "0"    "1/8"  "1/4"  "3/8"  "0.5"  "5/8"  "3/4"  "7/8"  "1"    "9/8"
#> [11] "5/4"  "11/8" "1.5"  "13/8" "7/4"  "15/8" "2"    "1/3"

# Example 3
fcvt(x3, denom=1)
#>  [1] "0"    "1/8"  "1/4"  "3/8"  "1/2"  "5/8"  "3/4"  "7/8"  "1"    "9/8"
#> [11] "5/4"  "11/8" "3/2"  "13/8" "7/4"  "15/8" "2"    "1/3"

# Example 4
fcvt(x3, denom=8)
#>  [1] "0/8"  "1/8"  "2/8"  "3/8"  "4/8"  "5/8"  "6/8"  "7/8"  "8/8"  "9/8"
#> [11] "10/8" "11/8" "12/8" "13/8" "14/8" "15/8" "16/8" "1/3"
```

`num2str`

Converts a set of numeric variables to a list of string representations, allowing for both decimal and fractional number formats. The function takes numeric variables as arguments and an optional denominator for the fractional representation. The result is a list where each element corresponds to the string representation of a numeric variable.

```
x <- 1
str(num2str(x))
#> List of 1
```

```
#>  $ x: chr "1"
y <- 2
str(num2str(x, y))
#> List of 2
#>  $ x: chr "1"
#>  $ y: chr "2"
str(num2str(x, y, z=c(x,y)))
#> List of 3
#>  $ x: chr "1"
#>  $ y: chr "2"
#>  $ z: chr [1:2] "1" "2"
```

Quote and Prefix and/or Suffix Manipulation

affix, unaffix

- `affix` adds a specified prefix and/or suffix to a character vector.

```
random_values <- runif(5)
new_value <- affix(random_values, prefix = "$", suffix = "$")
```

- `unaffix` removes specified prefixes and/or suffixes from a character vector.

```
random_numbers <- c("$15.3", "$7.9", "$22.6")
new_numbers <- unaffix(random_numbers, prefix = "$", suffix = "")
```

cdata, uncdata

- `cdata` adds a <[CDATA[prefix and]]> suffix to a character vector, ensuring proper encapsulation for XML or HTML data content.

```
new_data <- c(5.5, 12.3, 8.9)
cdata_representation <- cdata(new_data)
```

- `uncdata` removes the <[CDATA[prefix and]]> suffix from a character vector, commonly used in XML and HTML processing.

```
cdata_numbers <- c("<![CDATA[30.5]]>", "<![CDATA[18.2]]>", "<![CDATA[45.7]]>")
new_numbers <- uncdata(cdata_numbers)
```

bracket

Adds a (as prefix and) as suffix to a (character) vector.

```
existing_values <- c(10, 20, 30)
new_values <- bracket(existing_values)
```

math

Encloses a character vector with the dollar symbol ($) as both prefix and suffix, often used for mathematical expressions.

```
numeric_vector <- c(3.14, 2.718, 1.618)
math_representation <- math(numeric_vector)
```

unquote

Eliminates double quotes as both prefix and suffix from a character vector.

```r
quoted_values <- c("\"42.0\"", "\"8.8\"", "\"16.5\"")
unquoted_values <- unquote(quoted_values)
```

breaks

Generates a set of breakpoints for a given data vector `x`. The breaks can be either equidistant or non-equidistant. If the `width` parameter is not specified, it defaults to the first difference of the rounded values from `pretty(x)`. The `probs` parameter defines the number of quantiles or a vector of probabilities with values in [0, 1]. If the `width` is too large, using `probs` may result in equidistant breaks.

```r
# Generate breaks for a random normal distribution
x <- rnorm(100, mean = 1.8, sd = 0.1)
breaks(x)
#> [1] 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1

# Generate breaks with specified width for the same distribution
breaks(x, 0.1)
#> [1] 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1

# Generate quantile-based breaks with specified width for the distribution
breaks(x, 0.1, probs = 4)
#>   0%   0%  25%  50%  75% 100% 100%
#>  1.4  1.5  1.7  1.8  1.9  2.0  2.1
```

Vector to String Conversion

as_fraction

Converts numeric values into fractions, optionally in LaTeX format and allowing sorting.

```r
x <- round(runif(5), 2)
as_fraction(x)
#> [1] "13/25"  "99/100" "63/100" "19/100" "93/100"
as_fraction(x, latex = TRUE)
#> [1] "\\frac{13}{25}"  "\\frac{99}{100}" "\\frac{63}{100}" "\\frac{19}{100}"
#> [5] "\\frac{93}{100}"
```

as_obs

Creates a string representing observations with optional sorting and LaTeX formatting.

```r
# Taken from the exercise "Niederschlag"
smean <- 250:350
ssig  <- 1:10
ski   <- sample(smean, 1)
sigma <- sample(ssig, 1)
a <- ski-sigma
b <- ski+sigma
repeat{
  X    <- sample(seq(a,b,1),5,replace=TRUE)
  xbar <- sum(X)/5
  if (abs(xbar-round(xbar))<1e-3) break
}
#part a
sumSize = sum(X)
```

```r
xBar <- round(xbar,2)
S2   <- round(var(X), 2)
sx   <- as_obs(X, last=" und ")
```

as_string

Converts a vector or list of values into a readable string with specified separators.

```r
# Taken from the exercise "Dart 2"
fields  <- c(6, 13, 4, 18, 1, 20, 5, 12, 9, 14, 11, 8, 16, 7, 19, 3, 17, 2, 15, 10)
N       <- 82
ind     <- sort(sample(20, 2))
mname   <- paste0("eines der Felder, die zu den Nummern ", as_string(fields[ind[1]:ind[2]], last=" oder
print(mname)
#> [1] "eines der Felder, die zu den Nummern 16, 7, 19, 3, 17 oder 2 gehören"
```

as_sum

Creates a string representation of a sum expression for numeric values.

```r
x <- round(runif(5), 2)
as_sum(x)
#> [1] "0.51+0.15+0.43+0.51+0.7"
```

Miscellaneous Functions

Function Helper

gapply

The `gapply` function executes a given function (`FUN`) for all combinations of parameters specified in the ellipsis (`...`). This facilitates grid application, where each combination of parameters is applied to the function. The use of `I(.)` allows preventing certain elements from being interpreted as grid values. If an error occurs during the execution of the function, the corresponding result will not be stored, and missing indices may be observed in the returned list.

```r
# Execute 4 function calls: sum(1,3,5:6), sum(1,4,5:6), ..., sum(2,4,5:6)
gapply("sum", 1:2, 3:4, I(5:6))
#>   V1
#> 1 15
#> 2 16
#> 3 16
#> 4 17
```

Formatting

replace_fmt

The `replace_fmt` function is designed to substitute names within a text with values that are formatted either through the `exams::fmt()` function or as strings. This facilitates the integration of formatted values or strings into a given text.

```r
# Formatting numeric values with a list specifying precision for each variable, overriding y's precisio
result1 <- replace_fmt("\\frac{x}{y}", x = 2, y = 3, digits = list(2, y = 0))

# Formatting LaTeX expressions as strings
result1 <- replace_fmt("\\frac{x}{y}", x = "\\\\sum_{i=1}^n x_i", y = "\\\\sum_{i=1}^n y_i")
```

The first example showcases custom precision for each variable using a list, with `y` overridden to have zero digits. The second example illustrates the use of LaTeX expressions as strings, incorporating them into the formatted LaTeX expression.

LaTeX and HTML Functions (Multi-Format Rendering Functions)

Introductory LaTeX Functions

`answercol`

Customizes LaTeX documents by specifying the number of answer columns using the command.

```
# Set the number of answer columns to 2 in the LaTeX document
answercol(2)
```

`hypothesis_latex`

This function generates a structured data frame to represent test hypotheses. The resulting data frame includes various columns:

- `h0.left`: Represents the left value in the null hypothesis, typically denoted as μ or π.
- `h0.operator`: Indicates the operator used in the null hypothesis, selected from eq, ne, lt, le, gt, or ge.
- `h0.right`: Denotes the right value in the null hypothesis, often expressed as μ_0, π_0, or a hypothetical value.
- `h1.left`: Signifies the left value in the alternative hypothesis, typically μ or π.
- `h1.operator`: Specifies the operator in the alternative hypothesis, chosen from eq, ne, lt, le, gt, or ge.
- `h1.right`: Represents the right value in the alternative hypothesis, usually μ_0, π_0, or a hypothetical value.
- `H0`: Provides the LaTeX representation of the null hypothesis.
- `H1`: Presents the LaTeX representation of the alternative hypothesis.
- `match.left`: Indicates whether the left values in the null and alternative hypotheses match.
- `match.right`: Specifies whether the right values in the null and alternative hypotheses match.
- `match.operator`: Determines whether the operators in the null and alternative hypotheses match, covering all real numbers.
- `match.type`: Describes the matching type as wrong, left.sided, right.sided, two.sided, greater, or less. If the null hypothesis is not provided, it is determined from the alternative hypothesis. Valid values for the alternative and null include two.sided, greater, less, eq, ne, lt, le, gt, or ge.

```
hypothesis_latex("\\mu", alternative=c("eq", "ne", "lt", "le", "gt", "ge"),
                         null=c("eq", "ne", "lt", "le", "gt", "ge"))
#>    h0.left h0.operator h0.right h1.left h1.operator h1.right                 H0
#> 1     \\mu          eq    \\mu_0    \\mu          eq    \\mu_0      \\mu = \\mu_0
#> 2     \\mu          ne    \\mu_0    \\mu          eq    \\mu_0 \\mu \\neq \\mu_0
#> 3     \\mu          lt    \\mu_0    \\mu          eq    \\mu_0      \\mu < \\mu_0
#> 4     \\mu          le    \\mu_0    \\mu          eq    \\mu_0 \\mu \\leq \\mu_0
#> 5     \\mu          gt    \\mu_0    \\mu          eq    \\mu_0      \\mu > \\mu_0
#> 6     \\mu          ge    \\mu_0    \\mu          eq    \\mu_0 \\mu \\geq \\mu_0
#> 7     \\mu          eq    \\mu_0    \\mu          ne    \\mu_0      \\mu = \\mu_0
#> 8     \\mu          ne    \\mu_0    \\mu          ne    \\mu_0 \\mu \\neq \\mu_0
#> 9     \\mu          lt    \\mu_0    \\mu          ne    \\mu_0      \\mu < \\mu_0
#> 10    \\mu          le    \\mu_0    \\mu          ne    \\mu_0 \\mu \\leq \\mu_0
#> 11    \\mu          gt    \\mu_0    \\mu          ne    \\mu_0      \\mu > \\mu_0
#> 12    \\mu          ge    \\mu_0    \\mu          ne    \\mu_0 \\mu \\geq \\mu_0
#> 13    \\mu          eq    \\mu_0    \\mu          lt    \\mu_0      \\mu = \\mu_0
#> 14    \\mu          ne    \\mu_0    \\mu          lt    \\mu_0 \\mu \\neq \\mu_0
#> 15    \\mu          lt    \\mu_0    \\mu          lt    \\mu_0      \\mu < \\mu_0
```

```
#> 16    \\mu          le   \\mu_0    \\mu          lt   \\mu_0 \\mu \\leq \\mu_0
#> 17    \\mu          gt   \\mu_0    \\mu          lt   \\mu_0     \\mu > \\mu_0
#> 18    \\mu          ge   \\mu_0    \\mu          lt   \\mu_0 \\mu \\geq \\mu_0
#> 19    \\mu          eq   \\mu_0    \\mu          le   \\mu_0     \\mu = \\mu_0
#> 20    \\mu          ne   \\mu_0    \\mu          le   \\mu_0 \\mu \\neq \\mu_0
#> 21    \\mu          lt   \\mu_0    \\mu          le   \\mu_0     \\mu < \\mu_0
#> 22    \\mu          le   \\mu_0    \\mu          le   \\mu_0 \\mu \\leq \\mu_0
#> 23    \\mu          gt   \\mu_0    \\mu          le   \\mu_0     \\mu > \\mu_0
#> 24    \\mu          ge   \\mu_0    \\mu          le   \\mu_0 \\mu \\geq \\mu_0
#> 25    \\mu          eq   \\mu_0    \\mu          gt   \\mu_0     \\mu = \\mu_0
#> 26    \\mu          ne   \\mu_0    \\mu          gt   \\mu_0 \\mu \\neq \\mu_0
#> 27    \\mu          lt   \\mu_0    \\mu          gt   \\mu_0     \\mu < \\mu_0
#> 28    \\mu          le   \\mu_0    \\mu          gt   \\mu_0 \\mu \\leq \\mu_0
#> 29    \\mu          gt   \\mu_0    \\mu          gt   \\mu_0     \\mu > \\mu_0
#> 30    \\mu          ge   \\mu_0    \\mu          gt   \\mu_0 \\mu \\geq \\mu_0
#> 31    \\mu          eq   \\mu_0    \\mu          ge   \\mu_0     \\mu = \\mu_0
#> 32    \\mu          ne   \\mu_0    \\mu          ge   \\mu_0 \\mu \\neq \\mu_0
#> 33    \\mu          lt   \\mu_0    \\mu          ge   \\mu_0     \\mu < \\mu_0
#> 34    \\mu          le   \\mu_0    \\mu          ge   \\mu_0 \\mu \\leq \\mu_0
#> 35    \\mu          gt   \\mu_0    \\mu          ge   \\mu_0     \\mu > \\mu_0
#> 36    \\mu          ge   \\mu_0    \\mu          ge   \\mu_0 \\mu \\geq \\mu_0
#>                    H1 match.left match.right match.operator        type
#> 1       \\mu = \\mu_0       TRUE        TRUE          FALSE       wrong
#> 2       \\mu = \\mu_0       TRUE        TRUE           TRUE       wrong
#> 3       \\mu = \\mu_0       TRUE        TRUE          FALSE       wrong
#> 4       \\mu = \\mu_0       TRUE        TRUE          FALSE       wrong
#> 5       \\mu = \\mu_0       TRUE        TRUE          FALSE       wrong
#> 6       \\mu = \\mu_0       TRUE        TRUE          FALSE       wrong
#> 7  \\mu \\neq \\mu_0       TRUE        TRUE           TRUE   two.sided
#> 8  \\mu \\neq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 9  \\mu \\neq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 10 \\mu \\neq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 11 \\mu \\neq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 12 \\mu \\neq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 13      \\mu < \\mu_0       TRUE        TRUE          FALSE right.sided
#> 14      \\mu < \\mu_0       TRUE        TRUE          FALSE       wrong
#> 15      \\mu < \\mu_0       TRUE        TRUE          FALSE       wrong
#> 16      \\mu < \\mu_0       TRUE        TRUE          FALSE       wrong
#> 17      \\mu < \\mu_0       TRUE        TRUE          FALSE       wrong
#> 18      \\mu < \\mu_0       TRUE        TRUE           TRUE        less
#> 19 \\mu \\leq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 20 \\mu \\leq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 21 \\mu \\leq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 22 \\mu \\leq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 23 \\mu \\leq \\mu_0       TRUE        TRUE           TRUE       wrong
#> 24 \\mu \\leq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 25      \\mu > \\mu_0       TRUE        TRUE          FALSE       wrong
#> 26      \\mu > \\mu_0       TRUE        TRUE          FALSE       wrong
#> 27      \\mu > \\mu_0       TRUE        TRUE          FALSE       wrong
#> 28      \\mu > \\mu_0       TRUE        TRUE           TRUE     greater
#> 29      \\mu > \\mu_0       TRUE        TRUE          FALSE       wrong
#> 30      \\mu > \\mu_0       TRUE        TRUE          FALSE       wrong
#> 31 \\mu \\geq \\mu_0       TRUE        TRUE          FALSE       wrong
```

```
#> 32 \\mu \\geq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 33 \\mu \\geq \\mu_0       TRUE        TRUE           TRUE       wrong
#> 34 \\mu \\geq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 35 \\mu \\geq \\mu_0       TRUE        TRUE          FALSE       wrong
#> 36 \\mu \\geq \\mu_0       TRUE        TRUE          FALSE       wrong
```

Here the function `hypothesis_latex` is used to generate a data frame that represents different hypotheses related to the population mean (μ). Let's break down the key components of this example:

- `\\mu`: The symbol for the population mean in LaTeX format, which is specified as the first argument to the function.
- `alternative`: A vector specifying the alternative hypotheses. In this example, the alternatives include:
  - `eq`: Equality
  - `ne`: Inequality
  - `lt`: Less than
  - `le`: Less than or equal to
  - `gt`: Greater than
  - `ge`: Greater than or equal to
- `null`: A vector specifying the null hypotheses. It includes the same set of hypotheses as the `alternative` vector.

The function will generate a data frame with columns representing various aspects of the hypotheses, such as left and right values, operators, LaTeX representations, and matching criteria.

The resulting data frame will contain rows corresponding to all possible combinations of operators in the null and alternative hypotheses. Each row represents a unique hypothesis scenario. The `match` columns indicate whether the left and right values, as well as the operators, match between the null and alternative hypotheses.

In essence, this example explores and generates a comprehensive set of hypotheses involving the population mean with different combinations of operators in both null and alternative hypotheses.

latexdef

Enhances LaTeX document customization by adding a command, enabling the inclusion of personalized definitions within the document body.

```
latexdef("myvariable", "42")
```

pdensity and toLatex

The `pdensity` function generates a density function in a specified interval [a, b], where the endpoints a and b are sampled from the input vector x. The function can create either a linear (power=1) or constant (power=0) density function. It samples a specified number of elements (`size`) without replacement and calculates the values of the distribution function.

`toLatex` generates a LaTeX representation of the distribution and its parameters.

```
# Taken from the exercise "Constant_Density"
ops   <- c("\\leq", "<", "\\geq", ">")
sym   <- sample(1:2, size=2, replace=TRUE)
dens  <- pdensity(-5:5, size=4, power=0)
xdens <- toLatex(dens$pcoeff, digits=FALSE)
tdens <- toLatex(dens$pcoeff, digits=FALSE, variable="t")
tdist <- toLatex(integral(dens$pcoeff), digits=FALSE, variable="t")
str(dens)
#> List of 6
```

```
#>  $ a     : int -4
#>  $ b     : int 3
#>  $ x     : int [1:4] -4 -3 2 3
#>  $ fx    : num [1:4] 0 0.143 0.857 1
#>  $ pcoeff: 'polynomial' num 0.143
#>  $ pint  : num [1:3] 1 -0.5 4.33
print(tdist)
#> [1] "\\frac{1}{7}\\cdot t"
```

In this exercise, the `pdensity` function is used to generate a density function within a specified interval. The `pdensity` function is called with the following parameters: - `x`: The vector `-5:5` is provided, from which the endpoints of the interval will be sampled. - `size`: `4` elements will be sampled without replacement. - `power`: `0` specifies that a constant density function should be generated.

The resulting `dens` object contains information about the generated density function. Specifically, `dens$pcoeff` holds the coefficients of the generated density function.

- `toLatex` is used to convert the coefficients of the density function to LaTeX format.
  - `xdens`: The coefficients without any specific variable, essentially the constant terms.
  - `tdens`: The coefficients with the variable "t" specified.
  - `tdist`: The integral of the density function with respect to "t" is converted to LaTeX.

`toLatex`

After getting a glimpse of the `toLatex` function in the previous example, let's now explore it further in detail.

The `toLatex` S3 method is a versatile tool for generating LaTeX representations, focusing on statistical distributions and parameters. Derived functions cover a range of scenarios, including solution paths, matrices, polynomials, and equation solutions through tools like `num_solve()`. This suite provides a practical toolkit for producing LaTeX output across various mathematical and statistical contexts.

`toLatex.distribution` Generates LaTeX representation for statistical distributions and their parameters.

`toLatex.equation_solve` This function retrieves a LaTeX representation of the solution path obtained through the use of `num_solve()`. It inherits parameters from the base `utils::toLatex` function, providing compatibility with its usage.

`toLatex.html_matrix` Produces a LaTeX representation for matrices with limited style options.

`toLatex.polynomial` Generates a LaTeX representation for polynomials.

`toLatex.prob_solve` Presents solution pathways in LaTeX/MathJax using an align* environment.

`toHTMLorLatex`

This function produces either an HTML or LaTeX representation of a matrix, contingent on whether the function is invoked within the context of `exams2pdf`.

```
# Example: Generating HTML or LaTeX representation based on context
matrix_example <- html_matrix(matrix(1:4, nrow = 2))
result <- toHTMLorLatex(matrix_example)
str(result)
#>  chr "<table>\n<tr><td style=\"text-align:left;background-color:#999999;vertical-align:top;font-weig
```

In this example, the `toHTMLorLatex` function is employed to generate either an HTML or LaTeX representation of a matrix. The choice between HTML and LaTeX output depends on whether the function is called within

the context of `exams2pdf`. The resulting representation is then printed to the console. Adjust the matrix content and structure as needed for the specific use case.

Supporting Functions for Math LaTeX Output

`lsumprod`, `lsum`, `lprod`, `lmean`, `lvar`, `lbr`, `lsgn` and `lvec`

1. `lsumprod`: Creates a LaTeX printout of the sum of the products of corresponding elements in vectors `x` and `y`, including brackets if any element in `x` or `y` starts with a minus sign.

```
lsumprod(-2:2, (1:5)/10)
#> [1] "\\left(-2\\right) \\cdot 0.1 + \\left(-1\\right) \\cdot 0.2 + 0 \\cdot 0.3 + 1 \\cdot 0.4 + 2 \
```

This example generates the LaTeX expression for the sum of products:

$$(-2) \cdot 0.1 + (-1) \cdot 0.2 + 0 \cdot 0.3 + 1 \cdot 0.4 + 2 \cdot 0.5$$

2. `lsum`:Creates a LaTeX printout of the sum of elements in vector `x`.

```
lsum(-2:2)
#> [1] "-2-1+0+1+2"
```

This example generates the LaTeX expression for the sum:

$$-2 - 1 + 0 + 1 + 2$$

3. `lprod`:Creates a LaTeX printout of the product of elements in vector `x`.

```
lprod(-3:2)
#> [1] "\\left(-3\\right) \\cdot \\left(-2\\right) \\cdot \\left(-1\\right) \\cdot 0 \\cdot 1 \\cdot 2"
```

This example generates the LaTeX expression for the product:

$$(-3) \cdot (-2) \cdot (-1) \cdot 0 \cdot 1 \cdot 2$$

4. `lmean`: Creates a LaTeX printout of the mean of elements in vector `x`.

```
lmean(-2:2)
#> [1] "\\frac{-2-1+0+1+2}{5}"
```

This example generates the LaTeX expression for the mean:

$$\frac{-2 - 1 + 0 + 1 + 2}{5}$$

5. `lvar`:Creates a LaTeX printout of the variance of elements in vector `x`.

```
lvar(1:5)
#> [1] "\\frac{\\left(1-3\\right)^2 + \\left(2-3\\right)^2 + \\left(3-3\\right)^2 + \\left(4-3\\right)^
```

`lvar(x)` will generate a LaTeX printout for the variance of the vector `x`. The output will be a mathematical representation of the variance formula:

$$\frac{(1 - \bar{x})^2 + (2 - \bar{x})^2 + (3 - \bar{x})^2 + (4 - \bar{x})^2 + (5 - \bar{x})^2}{5}$$

where $\bar{x}$ is the mean of the vector `x`.

6. `lbr`: Creates a LaTeX printout of the vector `x` with brackets if any element starts with a minus sign.

```
 lbr(-2:2)
#> [1] "\\left(-2\\right)" "\\left(-1\\right)" "0"
#> [4] "1"                 "2"
```

This example generates the LaTeX expressions for each element with brackets:

$$(-2), (-1), 0, 1, 2$$

7. `lsgn`: Creates a LaTeX printout of the vector `x` with a plus or minus sign at the beginning.

```
lsgn(-3:1)
#> [1] "-3" "-2" "-1" "+0" "+1"
```

In this example, `lsgn` will generate a LaTeX printout with a plus or minus at the beginning of each element. The output will be a LaTeX representation of the vector:

$$-3, -2, -1, +0, +1$$

8. `lvec`: is a versatile function designed to create a LaTeX printout of a vector `x`. This function allows for the specification of the left and right delimiters for the vector.

```
# Using lvec to create a LaTeX representation of a vector with square brackets
# lvec(c(1, 2, 3), left = "[", right = "]")

# Using lvec to create a LaTeX representation of a vector with angle brackets and custom collapse
# lvec(c("a", "b", "c"), left = "<", collapse = " \\cdot ")
```

`lprob` and `prob_solve`

- `prob_solve`: Given a set of events, it computes the total or conditional probability of the given event. If no solution is found, it returns NA. Events are specified using uppercase letters, and operators include ! (complementary event), | (conditional event), and ^ (intersection of events). The latex attribute of the return value contains computation steps, and if `getprob` is TRUE, it includes the prob vector and compute with all computation steps.
- `print`: Shows the solution in ASCII format.
- `toLatex`: Shows the solution in LaTeX/MathJax with an `align` environment.
- `lprob`: Converts `!A` to $\bar{A}$ and `A^B` to $A \cap B$.

```
# Example: Solving a Genetics Problem
# Consider two genes A and B with the following probabilities:
# P(A) = 0.6, P(B) = 0.4
# P(A|B) = 0.3, P(B|A) = 0.2

# Compute the probability of having both genes A and B (A^B)
result_genetics <- prob_solve("A^B", "A" = 0.6, "B" = 0.4, "A|B" = 0.3, "B|A" = 0.2)

# Print the result
print(result_genetics)
#> [1] 0.12
```

In this genetics example, consider genes A and B. The probabilities of having each gene individually (P(A) and P(B)) and the conditional probabilities (P(A|B) and P(B|A)) are given. The `prob_solve` function is used to compute the probability of having both genes A and B (A^B).

```
# Example: Probability Expression Transformation

# Suppose we have a probability expression in a format using ^ and !:
expression <- "!A^B"

# Apply the lprob function to transform the expression
transformed_expression <- lprob(expression)
```

```r
# Print the original and transformed expressions
cat("Original expression:", expression, "\n")
#> Original expression: !A^B
cat("Transformed expression:", transformed_expression, "\n")
#> Transformed expression: \bar{A} \cap B
```

In this example, we start with a probability expression `!A^B`. We then apply the `lprob` function to transform the expression by replacing `^` with the LaTeX representation for the intersection (`\\cap`) and `!A` with the LaTeX representation for the complement (`\\bar{A}`).

Markdown Functions

`inline`

This function is designed to knit (render) text within an R code chunk. It is utilized to incorporate text-based content into an R Markdown document, providing a convenient way to weave together narrative and code.

```r
result <- inline("2 + 2")
#> 2 + 2
cat("The result of the calculation is:", result, "\n")
#> The result of the calculation is:
```

`rv`

The provided exercise is used to calculate the expected value for a random variable `rvt`.

```r
rateperhour <- sample(10:25, 1)
rate    <- rateperhour/60
sec     <- 60/rate
d       <- distribution("exp", rate=rate)
number  <- rateperhour
length  <- 60
lambda  <- rate
rvt     <- rv("T", "Wartezeit in Minuten auf den nächsten Wähler")
str(rvt)
#>  chr "$T$: \"Wartezeit in Minuten auf den nächsten Wähler\""
```

In order to calculate the random variable `rvt`, we use the function `rv`. Here `rv` formats a random variable and its meaning for R Markdown using a symbol and the explanation to the symbol. The symbol "T" stands for the waiting time in minutes until the next voter arrives at a polling station. In this case, "T" indicates an exponential distribution as we can also observe from the function `distribution`.

`template`

This function creates a text template that allows the incorporation of R code snippets. The template, defined as a character string, can include placeholders marked by backticks, where the ellipsis represents variable names. The R code within these placeholders is then replaced by its corresponding evaluation based on the provided parameter values.

```r
# Example: Creating a dynamic template with embedded R code
tmpl <- "The sum of `r a` and `r b` is: `r a + b`"
result <- template(tmpl, a = 1, b = 2)
cat(result)
#> The sum of 1 and 2 is: 3
```

to_choice

To determine the correct level of measurement of a variable we use an Excel file with two columns with the name of the variable and the level of measurement.

```
# subset of variables we use, variable names are in German
data("skalenniveau")
skalen <- c("nominal", "ordinal", "metrisch")
stopifnot(all(skalenniveau$type %in% skalen))  # protect against typos
skala  <- sample(skalenniveau$type, 1)
exvars <- sample(nrow(skalenniveau), 8)
tf     <- (skalenniveau$type[exvars]==skala)
sc     <- to_choice(skalenniveau$name[exvars], tf)
# Additional answer: Does none fit?
sc$questions <- c(sc$questions, "Keine der Variablen hat das gewünschte Skalenniveau")
sc$solutions <- c(sc$solutions, !any(tf))
sc
#> $questions
#> [1] "Erlernter Beruf"
#> [2] "Finanzierung des Studiums"
#> [3] "Geschlecht"
#> [4] "Klausurpunkte"
#> [5] "Seitenzahl eines Buches"
#> [6] "Semesterzahl"
#> [7] "Temperatur in Grad Celsius"
#> [8] "Wohnsitz"
#> [9] "Keine der Variablen hat das gewünschte Skalenniveau"
#>
#> $solutions
#> [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

The `to_choice` function generates a object such that can be used in `answerlist` and `mchoice2string`. The first parameter is either a vector or data frame. The second parameter is a logical vector containing `TRUE` if the element in the vector (or row in the data frame) contains a true answer.

The parameter `shuffle` samples from the correct and false answers. The following example could replace the main code from the example above.

```
# Subset of variables we use, variable names are in German
data("skalenniveau")
skalen <- c("nominal", "ordinal", "metrisch")
skala  <- sample(skalenniveau$type, 1)
exvars <- sample(nrow(skalenniveau), 8)
tf     <- (skalenniveau$type[exvars]==skala)
# select one true and four false answers
sc     <- to_choice(skalenniveau$name[exvars], tf, shuffle=c(1,4))
sc
#> $questions
#> [1] "Abonnierten Zeitungen" "Aggressivität"         "Geschlecht"
#> [4] "Grundstücksgröße"      "Telefonnummer"
#>
#> $solutions
#> [1] FALSE FALSE FALSE  TRUE FALSE
```

By default the answers are arranged in a certain order, determined by the parameter `order`, which is used to arrange the answers (default: `order`). To use the ordering given, set `order=NULL`.

HTML Functions

html_e2m

The `html_e2m` function facilitates the creation of an HTML page containing the contents of XML tags that match a specified pattern. By default, it displays the contents of all XML tags. The resulting HTML page is stored in the specified HTML file name.

If `name` is set to NULL (default), a temporary file is created. If the specified name does not end with .html, the function appends .html.

When `browseURL` is set to TRUE (default), the HTML page is automatically opened in the default web browser.

If needed, the contents of XML tags are concatenated with `\n`. Users have the flexibility to customize the concatenation for single XML tags using the `merge` parameter.

```
# if (interactive()) {
# Read XML data from an RDS file
# resexams <- readRDS(system.file("xml", "klausur-test.rds", package="exams.forge"))

# Create and display HTML page
# html_e2m(resexams) # Opens HTML file in the browser}
```

html_matrix_sk

A twist on creating a `html_matrix` object. It is important to note that the length of the `fmt` parameter must match either the number of rows (`nrow(m)`) or the number of columns (`ncol(m)`) in the matrix, depending on the `byrow` argument.

```
# html_matrix_sk(m)
#   tooltip(sprintf(tooltip, nrow(m), ncol(m)))
#   hm_cell(fmt=fmt, byrow=byrow)
```

```
# Create a matrix
m <- matrix(1:6, ncol=2)

# Generate and display an html_matrix object
html_matrix_sk(m, title="", fmt=c("%.0f", "%.1f"))
#>
#>      [,1] [,2]
#> [1,]    1  4.0
#> [2,]    2  5.0
#> [3,]    3  6.0

# Another small example taken from the exercise "Mobil Telephone 2"
 a  <- runif(4)
  pa <- ddiscrete(a)
  b  <- dpois(0:3, 1)
  pb <- ddiscrete(b)
  studie <- cbind(pa, pb)
hstudie <- html_matrix_sk(studie, "Studie / $x$", fmt=rep("%3.1f", 2))
print(hstudie)
#>
#> Studie / $x$  pa  pb
#>         [1,] 0.1 0.3
#>         [2,] 0.4 0.3
```

```
#>          [3,] 0.2 0.3
#>          [4,] 0.3 0.1
```

html_matrix, zebra and toHTML

Returns a HTML representation of a matrix as table. Any exercises created for Moodle can be embedded as HTML in an exercise and will be translated by `exams.forge` into HTML.

```
library("magrittr")
x  <- matrix(1:12, ncol=3)
hm <- html_matrix(x)
toHTML(hm)
#> [1] "<table>\n<tr><td style=\"text-align:left;background-color:#999999;vertical-align:top;font-weigh

# hm <- html_matrix(x) %>% zebra() %>%
# sprintf("Table has %.0f rows and %.0f columns", nrow(.), ncol(.))
# toHTML(hm)
```

With parameters the appearance of the table can be influenced:

- `title` entry at the top left (default: `""`)
- `caption` entry for the caption (default: `""`)
- `names$col` entry for the column names (default: `colnames(x)`)
- `names$row` entry for the row names (default: `rownames(x)`)
- `style$table` style for the table (default: `""`)
- `style$caption` style for the caption (default: `""`)
- `style$title` style for the caption (default: `"background-color:#999999;vertical-align:top;text-align:left;fo`
- `style$row` style for the row names (default: `"background-color:#999999;vertical-align:top;text-align:right;f`
- `style$col` style for the col names (default: `"background-color:#999999;vertical-align:top;text-align:right;fo`
- `style$cell` style for the col names (default: `c("background-color:#CCCCCC; vertical-align:top; text-align:right;", "background-color:#FFFFFF; vertical-align:top; text-align:right;")`)
- `style$logical` style for a logical matrix entry (default: `c("background-color:#CCCCCC; vertical-align:top; text-align:right;", "background-color:#FFFFFF; vertical-align:top; text-align:right;")`)
- `style$numeric` style for a numeric matrix entry (default: `c("background-color:#CCCCCC; vertical-align:top; text-align:right;", "background-color:#FFFFFF; vertical-align:top; text-align:right;")`)
- `style$char` style for a character matrix entry (default: `c("background-color:#CCCCCC; vertical-align:top; text-align:right;", "background-color:#FFFFFF; vertical-align:top; text-align:left;")`)
- `format$title$fmt` parameter to format the title via `sprintf` (default: `"\%s"`)
- `format$row$fmt` parameter to format the row names via `sprintf` (default: `"\%s"`)
- `format$col$fmt` parameter to format the col names via `sprintf` (default: `"\%s"`)
- `format$cell$fmt` parameter to format a matrix entry via `sprintf`
- `format$logical$fmt` parameter to format a logical matrix entry via `sprintf` (default: `"\%d"`)
- `format$numeric$fmt` parameter to format a numeric matrix entry via `sprintf` (default: `"\%f"`)

General Purpose Functions

Output Checker

firstmatch

Seeks matches for the elements of its first argument among those of its second. If multiple matches are found then the first match is returned, for further details see `charmatch`.

```
firstmatch("d", c("chisq", "cauchy"))
#> [1] NA
firstmatch("c", c("chisq", "cauchy"))
#> [1] 1
firstmatch("ca", c("chisq", "cauchy"))
#> [1] 2
```

gsimplify

The `gsimplify` function is designed to simplify a hyperloop object, primarily utilized in the context of grid applications. The goal is to reduce the complexity of the hyperloop object if simplification is feasible.

```
# Execute three t-test calls: t.test(x, -1), t.test(x, 0), t.test(x, 1)
ga <- gapply(t.test, x = I(rnorm(100)), mu = -1:1)

# No simplification occurs in this case since `data.name` and `conf.int` have lengths larger than one
str(gsimplify(ga))
#> List of 3
#>  $ :List of 10
#>   ..$ statistic  : Named num 9.48
#>   .. ..- attr(*, "names")= chr "t"
#>   ..$ parameter  : Named num 99
#>   .. ..- attr(*, "names")= chr "df"
#>   ..$ p.value    : num 1.47e-15
#>   ..$ conf.int   : num [1:2] -0.357 -0.0167
#>   .. ..- attr(*, "conf.level")= num 0.95
#>   ..$ estimate   : Named num -0.187
#>   .. ..- attr(*, "names")= chr "mean of x"
#>   ..$ null.value : Named int -1
#>   .. ..- attr(*, "names")= chr "mean"
#>   ..$ stderr     : num 0.0858
#>   ..$ alternative: chr "two.sided"
#>   ..$ method     : chr "One Sample t-test"
#>   ..$ data.name  : chr "structure(c(0.761784560279891, -0.491222740304113, 0.00332197499994945, 0.33
#>  $ :List of 10
#>   ..$ statistic  : Named num -2.18
#>   .. ..- attr(*, "names")= chr "t"
#>   ..$ parameter  : Named num 99
#>   .. ..- attr(*, "names")= chr "df"
#>   ..$ p.value    : num 0.0317
#>   ..$ conf.int   : num [1:2] -0.357 -0.0167
#>   .. ..- attr(*, "conf.level")= num 0.95
#>   ..$ estimate   : Named num -0.187
#>   .. ..- attr(*, "names")= chr "mean of x"
#>   ..$ null.value : Named int 0
#>   .. ..- attr(*, "names")= chr "mean"
#>   ..$ stderr     : num 0.0858
#>   ..$ alternative: chr "two.sided"
#>   ..$ method     : chr "One Sample t-test"
#>   ..$ data.name  : chr "structure(c(0.761784560279891, -0.491222740304113, 0.00332197499994945, 0.33
#>  $ :List of 10
#>   ..$ statistic  : Named num -13.8
#>   .. ..- attr(*, "names")= chr "t"
#>   ..$ parameter  : Named num 99
```

```
#>   .. ..- attr(*, "names")= chr "df"
#>   ..$ p.value    : num 7e-25
#>   ..$ conf.int   : num [1:2] -0.357 -0.0167
#>   .. ..- attr(*, "conf.level")= num 0.95
#>   ..$ estimate   : Named num -0.187
#>   .. ..- attr(*, "names")= chr "mean of x"
#>   ..$ null.value : Named int 1
#>   .. ..- attr(*, "names")= chr "mean"
#>   ..$ stderr     : num 0.0858
#>   ..$ alternative: chr "two.sided"
#>   ..$ method     : chr "One Sample t-test"
#>   ..$ data.name  : chr "structure(c(0.761784560279891, -0.491222740304113, 0.00332197499994945, 0.33
#>  - attr(*, "class")= chr [1:2] "hyperloop" "list"
```

hyperloop and unique_elem

For generating answers for multiple choice exercises it is helpful to run the same routine several times with different input parameters. For example students may forget to divide by n-1 or divide by n instead of n. `hyperloop` runs about all parameter combinations. `unique_elem` removes duplicate elements from a `hyperloop` object by considering specific list elements for comparison. As the outcome in each execution might be a list, the deletion process focuses on maintaining distinct elements within the `hyperloop` structure.

`ttest_num` is a routine which computes all information required for exercises with a t-test.

```
x <- runif(100)
correct <- ttest_num(x=x, mu0=0.5, sigma=sqrt(1/12))
str(correct)
#> List of 21
#>  $ mu0         : num 0.5
#>  $ x           : num [1:100] 0.658 0.224 0.415 0.848 0.246 ...
#>  $ sigma       : num 0.289
#>  $ norm        : logi FALSE
#>  $ mean        : num 0.514
#>  $ sd          : num 0.293
#>  $ n           : int 100
#>  $ alternative : chr "two.sided"
#>  $ Xbar        :List of 3
#>   ..$ mean: num 0.5
#>   ..$ sd  : num 0.0289
#>   ..$ name: chr "norm"
#>   ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   ..- attr(*, "discrete")= num 0
#>  $ Statistic   :List of 3
#>   ..$ mean: num 0
#>   ..$ sd  : num 1
#>   ..$ name: chr "norm"
#>   ..- attr(*, "class")= chr [1:2] "distribution" "list"
#>   ..- attr(*, "discrete")= num 0
#>  $ statistic   : num 0.483
#>  $ p.value     : num 0.629
#>  $ stderr      : num 0.0289
#>  $ n.clt       : num 30
#>  $ t2norm      : logi TRUE
#>  $ critical    : num [1:2] -1.96 1.96
```

```
#>  $ acceptance0 : num [1:2] -1.96 1.96
#>  $ criticalx   : num [1:2] 0.443 0.557
#>  $ acceptance0x: num [1:2] 0.443 0.557
#>  $ alpha       : num 0.05
#>  $ accept1     : logi FALSE
#>  - attr(*, "class")= chr [1:2] "ttest" "list"
```

Now, let us run many t-tests (up to 384) with typical student errors. We extract all different test statistic and choose seven wrong answers and one correct answer with the condition that all solutions differ at least by 0.05.

```
res <- hyperloop(ttest_num,
                 n           = list(1, correct$n, correct$n+1),
                 mu0         = list(correct$mu0, correct$mean),
                 mean        = list(correct$mu0, correct$mean),
                 sigma       = list(correct$sigma, correct$sd, sqrt(correct$sigma), sqrt(correct$sd)),
                 sd          = list(correct$sigma, correct$sd, sqrt(correct$sigma), sqrt(correct$sd)),
                 norm        = list(TRUE, FALSE)
                )
# extract all unique test statistics
stat <- unlist(unique_elem(res, "statistic"))
# select 7 wrong test statistic such that the difference
# between all possible test statistics is at least 0.01
repeat {
  sc <- to_choice(stat, stat==correct$statistic, shuffle=c(1,7))
  if (all_different(sc$questions, 0.005)) break
}
# show possible results for a MC questions
sc$questions
#> [1] -0.04755522 -0.02575125  0.00000000  0.02595336  0.04755522  0.25879683
#> [7]  0.47792405  0.48304650
sc$solutions
#> statistic statistic statistic statistic statistic statistic statistic statistic
#>     FALSE     FALSE     FALSE     FALSE     FALSE     FALSE     FALSE      TRUE
```

Text Processing and Formatting

knitif

The function knitif is designed to evaluate a logical condition and return a knitted result based on the outcome. It takes a text argument and produces the rendered output using R Markdown syntax.

```
knitif(runif(1) < 0.5, 'TRUE' = "`r pi`", 'FALSE' = "$\\pi=`r pi`$")
#> [1] "3.1415927"
```

In the given example, the `knitif` function is employed with the logical condition `runif(1) < 0.5`. This condition evaluates to `FALSE` in this specific instance. As a result, the function selects the text argument associated with `FALSE`, which is "
$pi = 3.1415927$". Therefore, the output of the `knitif` function in this example is "
$pi = 3.1415927$".

now

If we randomize the task and the stories then we may have a lot of different tasks. If questions arise then we need to identify the exact task a student has.

Therefore we embed a:

```
substring(now(), 10)
#> [1] "75926921367645263671 9"
```

The `now` function uses: `gsub('.', '', sprintf("%.20f", as.numeric(Sys.time())), fixed=TRUE)` and ensures that every time called a different number is returned.

nsprintf (round_de and schoice_de)

The `nsprintf` function generates text based on the value(s) provided in n. Specifically, it includes two sub-functions:

- `round_de`: Returns text indicating rounding instructions, such as "Round your result to the nearest whole number," "Round your result to one decimal place," or "Round your result to n decimal places."
- `schoice_de`: Returns text indicating that there can be one or more correct answers. It emphasizes that providing one correct answer is sufficient. If multiple answers are given and at least one is incorrect, the task is considered incorrectly answered.

```
# Example taken from the exercise "DSL 4"
repeat {
  border  <- sample(3:10, 1)-1
  lambda  <- sample(seq(0.5, 6, by=0.1), 1)
  if (ppois(border, lambda = lambda)>1e-3) break
}
d       <- distribution("pois", lambda=lambda)
ptype   <- "less"
sc      <- num_result(cdf(d, border), 4)
txt     <- nsprintf(border, "%i Netzunterbrechungen",
                    '0'="keine Netzunterbrechung",
                    '1'="eine Netzunterbrechung")
str(txt)
#>  chr "8 Netzunterbrechungen"
```

In this exercise, the `nsprintf` function is used to create a text message based on the value of `border`, which represents the number of network interruptions in a specific context. The resulting text is then embedded in the question text for the exercise. Here, `nsprintf` is used with the following parameters:

- `border`: The value to be included in the text.
- `"%i Netzunterbrechungen"`: The format string indicating where the value from `border` should be inserted. `%i` is a placeholder for an integer.

The following arguments provide alternative text depending on the value of `border`:

- `'0'="keine Netzunterbrechung"`: If `border` is 0, the text "keine Netzunterbrechung" (no network interruption) will be used.
- `'1'="eine Netzunterbrechung"`: If `border` is 1, the text "eine Netzunterbrechung" (one network interruption) will be used.

The resulting `txt` variable will contain a formatted text message that includes the value of `border` and provides context-specific information about network interruptions.

MIME

mime_image

The `mime_image` function returns the MIME type of an image based on the provided filename extension. In cases where a corresponding MIME type for a given file extension is not identified, the function returns the extension itself.

```r
image_file <- "example_image.jpg"

# Retrieve MIME type for the given image file
mime_type <- mime_image(image_file)

# Display the result
cat("MIME Type for", image_file, ":", mime_type, "\n")
#> MIME Type for example_image.jpg : jpeg
```

In this example, the `mime_image` function is used to obtain the MIME type for an image file named "example_image.jpg." The resulting MIME type is then printed using the `cat` function.