University of Duisburg-Essen
Faculty of Business Administration and Economics
Chair of Econometrics

tidyMC

An easy-to-use package for Monte-Carlo Simulations

Report

Submitted to the Faculty of
Business Administration and Economics
at the
University of Duisburg-Essen

from:

Ignacio Moreira Lara, Stefan Linner, Konstantin Lehmann

First Reviewer:	Jens Klenke
Second Reviewer:	Martin C. Arnold
Deadline:	06.09.2022

Name:	Ignacio Moreira	Stefan Linner	Konstantin Lehmann
Matriculation No.:	230658	233565	229994
E-Mail:	ignacio.moreira-lara@tu-dortmund.de	stefan.linner@tu-dortmund.de	konstantin.lehmann@tu-dortmund.de
Study Path:	M.Sc. Econometrics	M.Sc. Econometrics	M.Sc. Econometrics
Semester:	4th	2nd	4th
Graduation (est.):	Summer Term 2022	Summer Term 2022	Summer Term 2022

Introduction

Monte Carlo Simulations (henceforth MCS) are used to study the properties of econometrics inference techniques by simulation. They are almost always part of theoretical econometric research to study the performance of some inference technique. They supplement theoretical research most often in one of four ways. Firstly, they are used to assess the performance of asymptotically valid techniques in smaller samples. Secondly, they are used to assess how robust a technique is to violations of its theoretical assumptions. Thirdly, they allow to compare the performance of different techniques for a given data generating process (DGP). Fourthly, MCS allows to investigate the properties of inference techniques for which no analytic solution exists. Moreover, MCS itself can be used as a statistical inference technique. For example, Bootstrap Inference can be conceptualized as a special case of MCS. Beyond these examples, there are many more areas where MCS is used. However, independent of its use-case, easy and efficient implementation of MCS is currently only sparsely available.

With our `tidyMC` package, we want to enable researchers to easily and efficiently include MCS in their research using the programming language R. For one, `tidyMC` allows to easily speed up MCS by providing efficient parallelization options over a pre-specified parameter grid. Because MCS involves repeated computation, using parallel workers allows for significant computation time reductions. For another, `tidyMC` returns an easy to work with output. The output allows to access the result of all simulations for every parameter combination in the parameter grid. Moreover, `‘tidyMC‘` provides functions to effectively communicate the results of the MCS. Using the output, the simulation results can then be easily visualized using the package’s plot function. Because the function returns a ggplot object, researchers can customize the object to their liking. Additionally, the package provides a function that converts the results into publication-ready LaTex tables. The table is heavily customizable and thereby allows researchers to effectively communicate their results.

This report is structured as follows. Section @ref(monte-carlo-simulation) gives an introduction to why and how Monte Carlo simulations work and provides a simple OLS example. Section @ref(package-principles) explains the coding practices we followed and gives a conceptual overview of how we implemented MCS in R. Section @ref(vignette) presents tidyMC’s vignette. The vignette gives an easy-to-follow guide on how to implement MCS using the `tidyMC` package. Section @ref(comparison-to-other-monte-carlo-approaches-in-r) compares the performance of tidyMC to the performance of the MonteCarlo package and other potential MCS implementations. Section @ref(conclusion) concludes.

Monte Carlo Simulation {MCS}

MCS allows analyzing the properties of classic econometric inference techniques. As a simulation-based technique, MCS applies a statistical procedure to many synthetically created samples. The repeated application of the procedure to a fully known DGP helps to investigate the properties of statistical techniques in a wide set of settings. It is possible to use MCS as a simulation-based inference technique itself. In this section, we want to give a brief overview of how and in which circumstances MCS work , mainly following

Pseudo Random Number Generation and the Data Generating Process

Using MCS on computers has the great advantage that, with the necessary information, the simulation results can be replicated on most machines. This is possible because, given seed and algorithm, the numbers created by a random number generator are completely reproducible. MCS generally start with a synthetically created i.i.d. sample. Computers are, however, not able to generate truly random draws from different distributions. Instead, the computer draws from different distributions that are generated through pseudo-random draws from a uniform distribution. Draws from a uniform distribution are created by using so-called random number generators (RNG). These algorithms are applied to a natural number, called the seed. The resulting values are pseudo-random because they are indistinguishable from truly random numbers but are fully determined by the combination of the algorithm and the seed. The combined knowledge of the algorithm and the seed allows exact replication of a series of pseudo-random numbers generated. The purpose of the `tidyMC`-package is to additionally allow easy parallelization of the MCS. With parallel computing, special attention has to be paid to creating disjoint and sufficiently long sets of draws from a uniform distribution. Our package builds on the L'Ecuyer-CMRG RNG streams implemented in `furrr::furrr_options`. The algorithm creates reproducible sets of numbers for each parallel worker, using an integer seed of at most length 7. Alternatively, a seed of length 1 is accepted, which corresponds to a valid seed of length 7. Because the algorithm creates a subset of random numbers for each worker, the stream and hence the MCS results are only reproducible if the same number of parallel workers is used. Moreover, our function allows supplying user-generated lists of random variables. Complex multivariate distributions can then be simulated as functions of different standard distributions. This allows us to generate and work with distributions that are analytically not manageable.

Monte Carlo Distribution Approximation

MCS is most commonly used to infer properties of the distribution of statistics in a fixed setting, i.e. for a prespecified DGP, hence the simulation results hold only for this. Since it additionally only approximates the distribution numerically, we need to control for the resulting approximation inaccuracies. In MCS we generally simulate the distribution of a statistic q_n. The statistic can often be expressed as a function of a parameter vector β, some deterministic variables D, some random variables v, and a given number of observations, s.t. $q_n = q_n(\beta, D, v, n)$. What a Monte Carlos simulation ultimately achieves is a numerical approximation of the true distribution of q_n for the specified DGP. Let q_n^R be the statistic generated by the R^th Monte Carlo Simulation. Then the histogram of all q_n^R is an unbiased and consistent estimator of q_n. Thus, the precision of the approximation is controlled by setting the number of Monte Carlo estimates. We can then use different statistics such as mean and variance on q_n^R to summarize the distribution of the statistic for the prespecified DGP. The approach can of course be easily extended to a vector of statistics q_n.

Monte Carlo Best Practices

gives a few things that need to be considered when doing and presenting MCS. The results must allow the audience to get a precise picture of the Monte Carlo experiment. MCS generally involves testing the attributes of q_n not only for one DGP but for a wide set of DGPs that may include nuisance parameters that may significantly alter the distribution.

We present one example of a MCS. Instead of going through every use case, we next describe one specific MCS use case. The insights from that use case should however be easily transferable to other MCS applications. One way to perform MCS is mimic analytic derivations. For example, if properties of an asymptotically consistent technique are studied, the MCS should include different nuisance parameters over increasing sample sizes. The report of the results should then include the technique's key performance indicators, such as bias and efficiency, over different combinations of nuisance parameters and sample sizes. As the parameters are the explanatory variables of the MCS cite(dude), their values have to be chosen with care to appropriately represent the results from the experiment. The reported representative selection of sample size and nuisance parameters should then show the - potentially nonlinear- behavior of the key indicators of the performance of the technique. As the later sections will show, `‘tidyMC‘` provides easy options to define an arbitrarily large parameter grid to use for the MCS as well as user-defined performance indicators The packages summary functions allow the researcher to get a quick overview of the results of the combinations defined in the parameter grid. The plot and table functions allow afterward to easily communicate the desired results. Due to the flexible design of the `‘tidyMC‘`, the parameter grid can easily be extended to include different estimators for comparisons. It also allows using the distribution of the data itself to generate a new sample, from which we can then infer the properties of an estimator.

OLS consistency

To demonstrate how a MCS works, we provide an example using the estimation of a linear model using the ordinary least squares(OLS) estimator. Given a dependent variable y and a set of i.id. regressors $X = \{x_1, x_2\}$ we consider the linear DGP

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon, \tag{1}$$

where $\beta = \{\beta_0, \beta_1, \beta_2\}$ are the coefficients and ε is a normally distributed error term with variance σ^2, i.e. $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. The corresponding OLS estimators $\hat{\beta}$ and s^2 for β and σ^2 are

$$\hat{\beta} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'Y, \tag{2}$$

$$s^2 = \frac{1}{n-k-1}\hat{\varepsilon}'\hat{\varepsilon}, \tag{3}$$

where $k = 3$ is the number of regressors and $\hat{\varepsilon}$ is the vector of residuals. Both estimators are consistent and unbiased. Accordingly OLS should be able to correctly estimate the coefficients for every sample size, while the precision of the estimates should increase in the sample size.

Using a MCS we can demonstrate the properties of the OLS estimator. Fixing the coefficients $\beta = \{1, 4, 5\}$ and $\sigma^2 = 3$, we generate different DGPs using different normal distributions to generate the dependent variables x_1 and x_2. Additionally, we generate ε from its respective distribution, and we estimate OLS on this sample. The number of observations we use is set to $n = \{100, 200, 300\}$, which will be used to assess the convergence of the OLS estimators. We repeat the estimation 10000 times for each n.

The results demonstrate the performance of OLS estimation with increasing sample sizes. As expected figures **??**, **??** and table 1 confirm that a bigger sample size leads to a more precise estimation of the underlying population values. Comparing both panels, we see that this behavior is just confirmed, as the estimated values do not differ and they become more precise. Moreover, the densities present show the unbiasedness of the estimator and the decrease of its standard deviation with more observations.

Table 1: Ommited variable bias MC results

N. Observations	x_2 Included or not	$\hat{\beta}_0$	S.d. $\hat{\beta}_0$	$\hat{\beta}_1$	S.d. $\hat{\beta}_1$	$\hat{\beta}_2$	s^2
N = 10							
100	1	0.995	0.227	4.000	0.088	4.986	2.843
200	1	1.096	0.164	3.999	0.060	4.994	2.953
300	1	1.015	0.138	4.021	0.051	5.000	3.015
N = 1000							
100	1	0.987	0.235	4.000	0.088	5.003	2.945
200	1	0.998	0.165	4.000	0.061	5.000	2.948
300	1	0.999	0.135	4.000	0.050	5.000	2.978

Total repetitions = 1000, total parameter combinations = 6

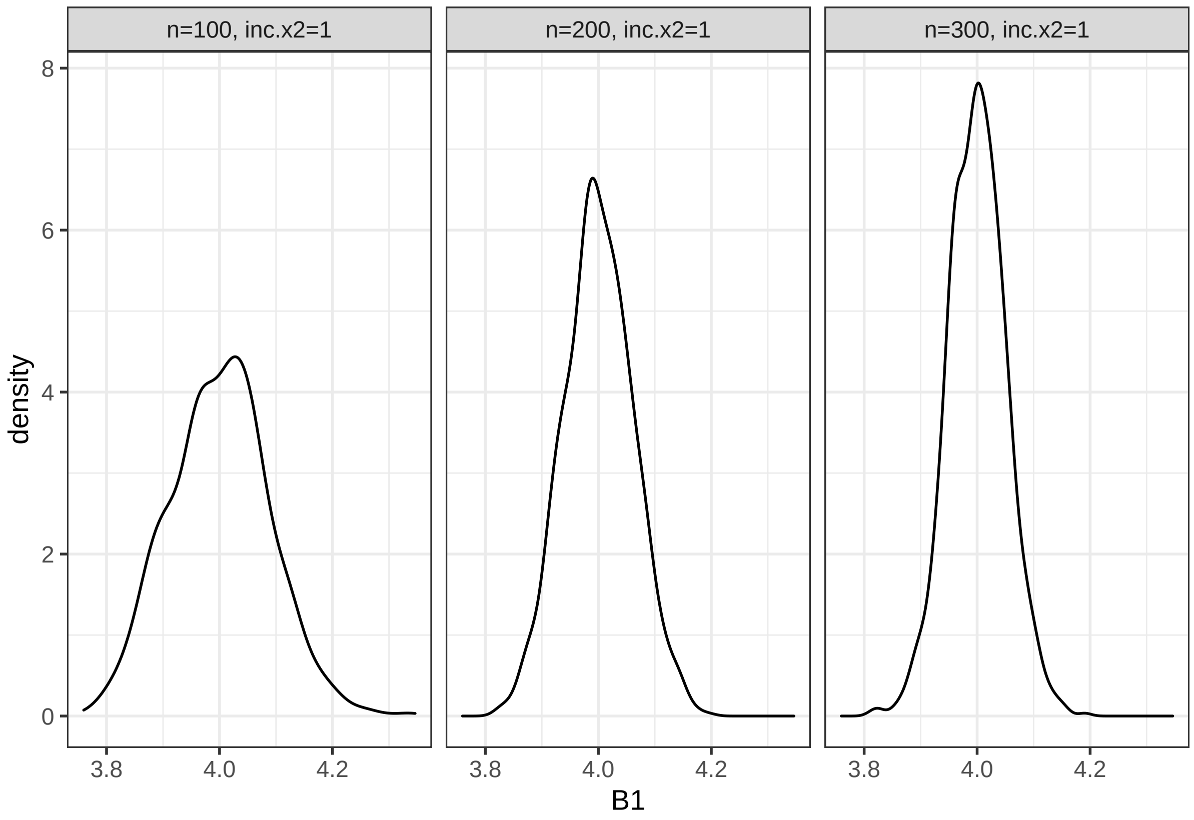


Figure 1: MC β_1 results

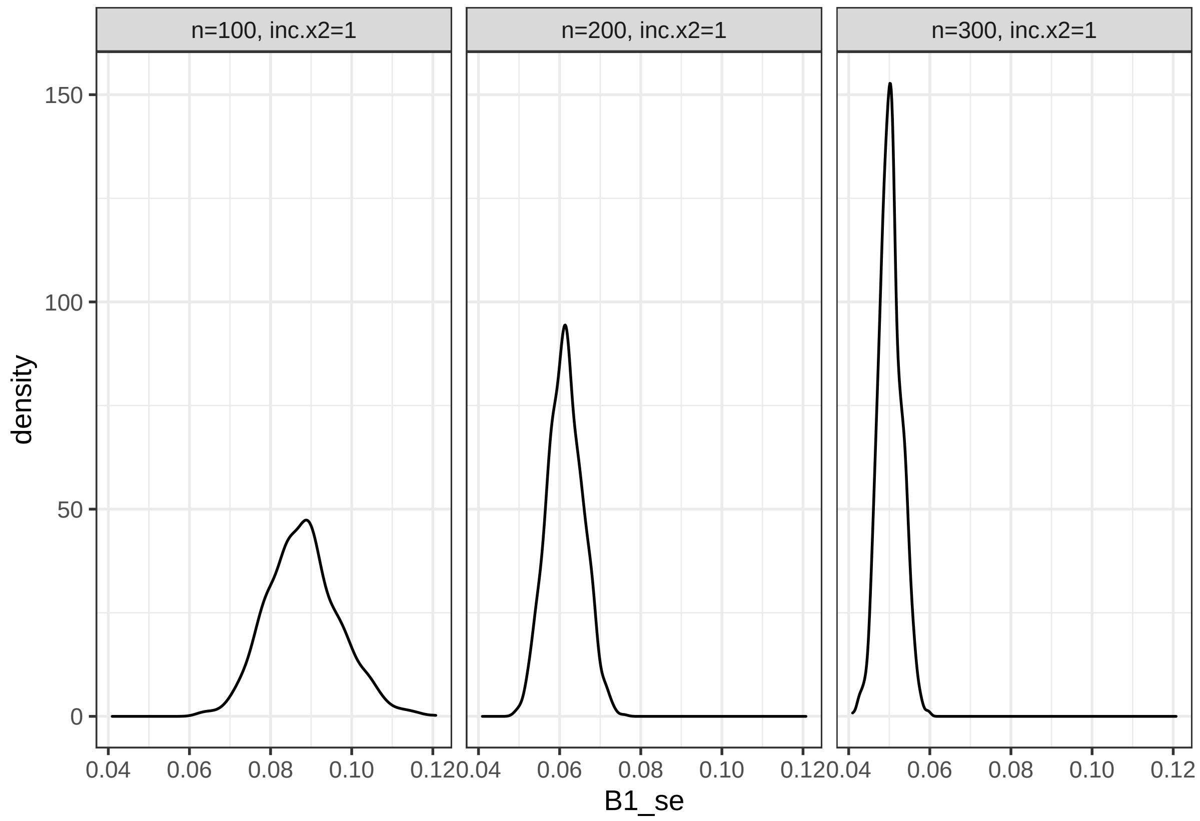


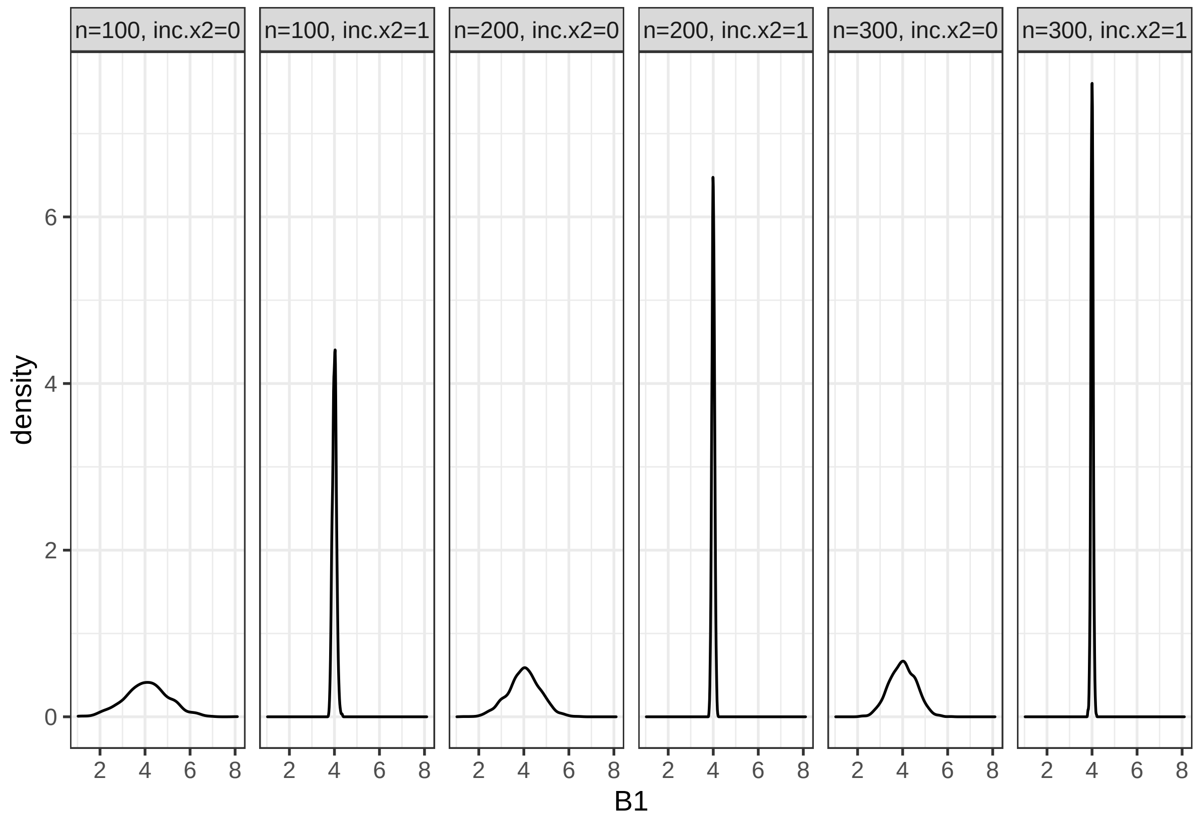
Figure 2: MC s^2 results

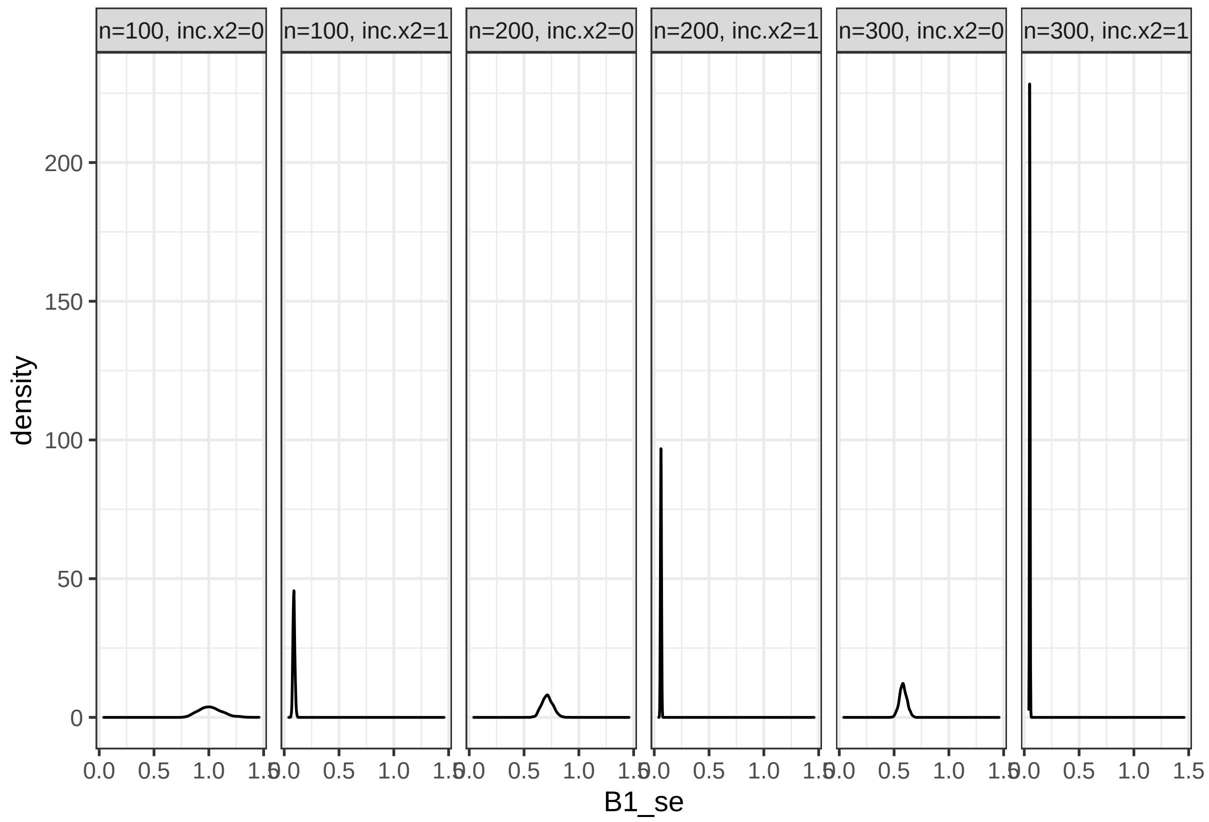
Omitted variable bias

Moreover, a common problem for estimation is the inclusion of suitable regressors into the model. One common problem is the absence of relevant variables in the model, also called omitted variable bias. In this subsection we test the impact of this problem in OLS estimation.

We test the effects of estimating a model using only x_1 when the underlying DGP follows the structure of (1). Since it is necessary that x_1 and x_2 share a degree of correlation between them, the absence of one of them in the regression model will cause the available variable to be correlated with the error term, thus violating one of the main assumptions OLS. This correlation will lead to $\hat{\beta}_1$ to be estimated less efficiently (depending on the sign of the correlation) in comparison to the true parameter.

We set the same structure of the estimated DGP identical to the base case, only that x_2 is not included into the estimation model. To show the loss in efficiency we present the density plots for the Monte Carlo results for $\hat{\beta}_1$ and its standard error in figures **??** and **??** . There is a clear difference in term of the variance of the estimated MC coefficients between the cases for when x_2 is included into the model and not, i.e. the velocity of the variance's decrease does not equal the case for then x_2 is included. As for the standard deviation of $\hat{\beta}_1$, we observe that the density centers itself around values different from zero. This pattern only repeats itself no matter the number of observations used by the model. Nonetheless, we observe that the OLS estimate remains unbiased in any case.





Vignette{vignette}

Package overview

Monte Carlo Simulations aim to study the properties of statistical inference techniques. At its core, a Monte Carlo Simulations works through the application of the techniques to repeatedly drawn samples from a pre-specified data generating process. The `tidyMC` package aims to cover and simplify the whole workflow of running a Monte Carlo simulation in either an academic or professional setting. Thus, `tidyMC` aims to provide functions for the following tasks:

- Running a Monte Carlo Simulation for a user defined function over a parameter grid using `future_mc()`
- Summarizing the results by (optionally) user defined summary functions using `summary.mc()`
- Creating plots of the Monte Carlo Simulation results, which can be modified by the user using `plot.mc()` and `plot.summary.mc()`
- Creating a LaTeX table summarizing the results of the Monte Carlo Simulation using `tidy_mc_latex()`

In the following subsections we will show you how you can implement those tasks using the `tidyMC` package.

Installing tidyMC

Until now, the `tidyMC` package is not on CRAN, thus you need to download the development version from GitHub as follows:

```r
# install.packages("devtools")
devtools::install_github("stefanlinner/tidyMC")
```

Afterwards you can load the package:

```r
library(tidyMC)
```

Moreover, the following packages will be used in this vignette:

```r
# install.packages("magrittr")
library(magrittr)
# install.packages("ggplot2")
library(ggplot2)
```

Run your first Monte Carlo Simulation with `future_mc()`

`future_mc()` allows you to run a Monte Carlo Simulation for a user defined function and given parameters. The first argument of `future_mc()` is `fun` which has to be a function that handles the generation of data, the application of the method of interest and the evaluation of the result for a single repetition and parameter combination. `future_mc()` handles the generation of loops over the desired parameter grids and the repetition of the Monte Carlo experiment for each of the parameter constellations. Consider the following example for fun and note that it performs the required tasks of generating data, applying the method and evaluating the results:

```r
# fun
ols_test <-
  function(b0, b1, b2, n, sigma2, param_x1, param_x2, inc.x2){

    # generation of data
    x1 <- rnorm(n = n, mean = param_x1[1], sd = param_x1[2])
    x2 <- rnorm(n = n,  mean = param_x2[1], sd = param_x2[2])
    e <- rnorm(n, sd = sqrt(sigma2))
    y <- b0 + b1*x1 + b2*x2 + e

    if (inc.x2 == 0){
      x2 <- x2 * inc.x2
    }

    # application of method
    estim <- lm(y ~ x1 + x2)

    # evaluation of the result for a single repetition and parameter combination
    out <- list(b0 = estim$coefficients[1],
                b1 = estim$coefficients[2],
                b2 = estim$coefficients[3],
                sigma2 = var(estim$residuals))
    return(out)
  }
```

The second argument of future_mc() is repetitions which should be an integer specifying the number of Monte Carlo iterations. While the third argument param_list should be a list whose components are named after the parameters of fun which should vary for the different Monte Carlo Simulation and each component is a vector containing the desired grid values for the parameter. Consider the following example for param_list and note that its components are named accordingly to the parameters of ols_test: n and inc.x2, respectively:

```r
# param_list
param_list_ols <-
  list(n = c(100, 200, 300), inc.x2 = c(0,1))
```

future_mc() takes care of creating all possible parameter combinations of param_list and runs the Monte Carlo Simulation over all of these for all possible combinations. If you don't want to run a Monte Carlo Simulation for every possible parameter combination you can alternatively define param_table with a data.frame or data.table containing the parameter combinations you are interested in. The ... argument can be used to specify further arguments of fun which are not contained in param_list. Those arguments will be held fixed for all parameter combinations. In our OLS example the arguments are b0, b1, b2, sigma2, param_x1, and param_x2.

Moreover, there are four formal requirements that fun and thus ols_test have to fulfill:

- The arguments of `fun` which are present in `param_list` have to be scalar values. Note that the arguments of `ols_test` which are contained in `param_list_ols`: `n` and `inc.x2` are scalar values. The remaining arguments of `ols_test` are allowed to take non-scalar values.
- Every variable used inside `fun` has either to be defined inside `fun` or given as an argument through the `...` argument.
- The value returned by `fun` has to be a named list. In our example the names of the returned list are `b0`, `b1`, `b2`, and `sigma2`.
- The names of the returned values and those of the arguments contained in `param_list` need to be different. Moreover, they cannot be `params`, `repetitions` or `setup` as these names are already occupied. Note that `b0`, `b1`, `b2`, and `sigma2` are the names of the returned values as well as names of arguments of `ols_test`. However, none of those arguments is contained in `param_list_ols`. If we would add either of those variables to `param_list_ols` we would need to change the name of the returned value for the respective variable.

We recommend to even further restrict the return value of `fun` to be a named list of scalars. This allows you to use all comfort functions of the `tidyMC` package. As you can see, we did that for `ols_test`.

The argument `parallelisation_plan` allows the user to set a parallelisation plan. While the argument `parallelisation_options` allows the user to fine tune functions, such as `furrr::future_map()` by `furrr::furrr_options()`. The argument `seed` for `furrr::furrr_options()` can be specified in `parallelisation_options` following the formal requirements of its respective documentation. Moreover, the user can also decide not to run the Monte Carlo in parallel by setting `parallel = FALSE`. To construct `parallelisation_plan` the user needs to provide a list named after the arguments of `future::plan`. The main argument `strategy` needs to provide the specific type of parallelisation the user would like to use and the number of cores which are used in the function. Some of the options for strategy are: `multisession`, `multicore` and `cluster`. We strongly recommend the user to read the documentation of the `future` package for a more detailed instruction of how to set up the different strategies.

As a default `future_mc()` runs a quick check by running a single test-iteration for each parameter combination in order to check for possible errors in `fun`. If a error occurs the user not only receives the error message but also the parameter combinations for which the error occurred:

```
set.seed(101)
first_mc_ols <- future_mc(
  fun = ols_test,
  repetitions = 1000,
  param_list = param_list_ols,
  b0 = 1,
  b1 = 4,
  b2 = 5,
  sigma2 = -2,
  param_x1 = c(0,5),
  param_x2 = c(0,6),
  check = TRUE
)
```

```
## Running single test-iteration for each parameter combination...
## Warning in sqrt(sigma2): NaNs produced

## Warning in rnorm(n, sd = sqrt(sigma2)): NAs produced

## Warning in sqrt(sigma2): NaNs produced

## Warning in rnorm(n, sd = sqrt(sigma2)): NAs produced

## Warning in sqrt(sigma2): NaNs produced

## Warning in rnorm(n, sd = sqrt(sigma2)): NAs produced

## Warning in sqrt(sigma2): NaNs produced

## Warning in rnorm(n, sd = sqrt(sigma2)): NAs produced

## Warning in sqrt(sigma2): NaNs produced

## Warning in rnorm(n, sd = sqrt(sigma2)): NAs produced

## Warning in sqrt(sigma2): NaNs produced

## Warning in rnorm(n, sd = sqrt(sigma2)): NAs produced

## Error in future_mc(fun = ols_test, repetitions = 1000, param_list = param_list_ols, :
##  Function error: NaNs produced
##  At the parameters: n = 100, inc.x2 = 0
##
##  Function error: NaNs produced
##  At the parameters: n = 200, inc.x2 = 0
##
##  Function error: NaNs produced
##  At the parameters: n = 300, inc.x2 = 0
##
##  Function error: NaNs produced
##  At the parameters: n = 100, inc.x2 = 1
##
##  Function error: NaNs produced
##  At the parameters: n = 200, inc.x2 = 1
##
##  Function error: NaNs produced
##  At the parameters: n = 300, inc.x2 = 1
```

The attentive reader might already have noticed that we specified `sigma2 = -2` which doesn't make sense, as the variance of the error term cannot be negative. This results in a failed check for all parameter combinations, as this parameter is held fixed for any combination. Once we correct that mistake, we can run our first Monte Carlo Simulation:

```r
set.seed(101)
first_mc_ols <- future_mc(
  fun = ols_test,
  repetitions = 1000,
  param_list = param_list_ols,
  b0 = 1,
  b1 = 4,
  b2 = 5,
  sigma2 = 2, # correctly specify sigma2
  param_x1 = c(0,5),
  param_x2 = c(0,6),
  check = TRUE
)
```

```
## Running single test-iteration for each parameter combination...

##
##  Test-run successfull: No errors occurred!

## Running whole simulation: Overall 6 parameter combinations are simulated ...

##
##  Simulation was successfull!
##  Running time:  00:00:05.571308
```

`future_mc` returns a list of type `mc` and length 11 consisting of a tibble (`first_mc_ols$output`) containing the return value of `fun` for each iteration and parameter combination. In our case `first_mc_ols$output` contains a column for each output `b0`, `b1`, `b2`, and `sigma2`, as well as a column for each parameter in `param_list_ols` and a column containing the `nice_names` of the parameter combinations. Overall the `first_mc_ols$output` consists of 6000 rows, i.e., for each parameter combination 1.000 rows:

```r
first_mc_ols$output
```

```
## # A tibble: 6,000 x 7
##    params                n inc.x2     b0    b1    b2 sigma2
##    <chr>             <dbl>  <dbl>  <dbl> <dbl> <dbl>  <dbl>
##  1 n=100, inc.x2=0   100      0  1.94    3.68    NA   895.
##  2 n=100, inc.x2=0   100      0 -3.39    4.37    NA  1115.
##  3 n=100, inc.x2=0   100      0  2.19    4.10    NA  1048.
##  4 n=100, inc.x2=0   100      0  4.52    3.28    NA   696.
##  5 n=100, inc.x2=0   100      0 -0.867   4.52    NA   860.
##  6 n=100, inc.x2=0   100      0  1.25    3.91    NA   901.
##  7 n=100, inc.x2=0   100      0 -0.135   2.57    NA   890.
##  8 n=100, inc.x2=0   100      0  1.71    3.86    NA  1093.
##  9 n=100, inc.x2=0   100      0  0.707   4.75    NA   659.
## 10 n=100, inc.x2=0   100      0  6.12    2.85    NA  1123.
## # ... with 5,990 more rows
```

If `ols_test` would not return a named list of scalars, but a named list of non-scalars, then `first_mc_ols$output` would not contain a column for each output, but a single column containing the named list of non-scalars for each iteration and parameter combination.

Moreover, `first_mc_ols` returns much other information about the Monte Carlo Simulation that can be printed in a dense representation:

```
first_mc_ols
```

```
## Monte Carlo simulation results for the specified function:
##
##  function (b0, b1, b2, n, sigma2, param_x1, param_x2, inc.x2)
## {
##     x1 <- rnorm(n = n, mean = param_x1[1], sd = param_x1[2])
##     x2 <- rnorm(n = n, mean = param_x2[1], sd = param_x2[2])
##     e <- rnorm(n, sd = sqrt(sigma2))
##     y <- b0 + b1 * x1 + b2 * x2 + e
##     if (inc.x2 == 0) {
##         x2 <- x2 * inc.x2
##     }
##     estim <- lm(y ~ x1 + x2)
##     out <- list(b0 = estim$coefficients[1], b1 = estim$coefficients[2],
##         b2 = estim$coefficients[3], sigma2 = var(estim$residuals))
##     return(out)
## }
##
##  The following 6 parameter combinations:
##     n inc.x2
## 1 100      0
## 2 200      0
## 3 300      0
## 4 100      1
## 5 200      1
## 6 300      1
## are each simulated 1000 times.
##
##  The Running time was: 00:00:05.571308
##
##  Parallel: TRUE
##
##  The following parallelisation plan was used:
## $strategy
## multisession:
## - args: function (..., workers = availableCores(), lazy = FALSE, rscript_libs = .libPaths(), envi
## - tweaked: FALSE
## - call: NULL
##
##
```

```
##   Seed: TRUE
```

Summarize your results with `summary.mc()`

If `fun` returns a named list of scalars the user can use `summary.mc()` to summarize all Monte Carlo results. The first argument of the function is an object of class `mc` returned by `future_mc()`. The next argument `sum_funs` determines which summarizing functions will be used on the simulation results. The functions can be provided for any combination of: parameter combinations resulting from `param_list`, and the outputs of `fun`. Every specified function can only take one argument, which is the vector (with length `repetitions`) for every output. We will present all customization options of `sum_funs` in a stepwise manner.

The first option of summarizing the results is given by just providing the `mc` object to `summary.mc`. In this case, `mean()` will be applied to all numeric values and `summary()` to all non-numeric data types. When the summarizing functions return one numeric value (like `mean()`) the results are twofold:

- First, a single scalar result of the function evaluated using the complete output vector is returned in the first element.
- Second, a vector with length `repetitions` of numeric results from the step wise calculation of the function's result across the output's vector. We call this resulting vector as the "path" of the summarizing function.

Additionally, to save computation time the parameter `which_path` is available to the user who wants to specify for which ouputs the "path" should be calculated. The user needs to provide a character vector with the output names'. Moreover, the options `"all"` (the default) and "none" are also available.

For the OLS example since all outputs of `ols_test` are numeric, the returned object will be a named nested list composed of four elements named after the `nice_names` returned by `future_mc()`. Each of this elements are itself lists containing the summarized outputs, i.e. `b0`, `b1`, `b2`, and `sigma2`. Lastly each of these are composed by the "path" and scalar result of `mean()`.

```
# Default
summary_default <- summary(first_mc_ols)
summary_default
```

```
## Results for the output b0:
##    n=100, inc.x2=0: 0.9976054
##    n=100, inc.x2=1: 0.997968
##    n=200, inc.x2=0: 1.003923
##    n=200, inc.x2=1: 0.9985081
##    n=300, inc.x2=0: 0.9847289
##    n=300, inc.x2=1: 0.9987762
##
##
## Results for the output b1:
##    n=100, inc.x2=0: 4.046832
##    n=100, inc.x2=1: 4.000378
##    n=200, inc.x2=0: 4.014101
##    n=200, inc.x2=1: 4.00071
##    n=300, inc.x2=0: 4.004808
##    n=300, inc.x2=1: 4.000037
##
##
## Results for the output b2:
##    n=100, inc.x2=0:
## [1] NA
##
##   n=100, inc.x2=1: 5.000759
##    n=200, inc.x2=0:
## [1] NA
##
##   n=200, inc.x2=1: 5.000076
##    n=300, inc.x2=0:
## [1] NA
##
##   n=300, inc.x2=1: 5.00016
##
##
## Results for the output sigma2:
##    n=100, inc.x2=0: 892.9367
##    n=100, inc.x2=1: 1.942735
##    n=200, inc.x2=0: 897.2449
##    n=200, inc.x2=1: 1.974316
##    n=300, inc.x2=0: 896.9941
##    n=300, inc.x2=1: 1.978199
##
##
```

```
str(summary_default[[1]])
```

```
## List of 4
##  $ b0    :List of 2
##   ..$ mean          : num 0.998
##   ..$ mean_over_reps: num [1:1000] 1.942 -0.725 0.247 1.316 0.88 ...
##  $ b1    :List of 2
##   ..$ mean          : num 4.05
##   ..$ mean_over_reps: num [1:1000] 3.68 4.02 4.05 3.85 3.99 ...
##  $ b2    :List of 1
##   ..$ mean: num NA
##  $ sigma2:List of 2
##   ..$ mean          : num 893
##   ..$ mean_over_reps: num [1:1000] 895 1005 1020 939 923 ...
```

This nested list structure should give an idea of the reach and flexibility the `sum_funs` argument is allowed to have, since the user can specify a function for each element in this list.

For an intermediate level of customization, the user can provide a combination of summarizing functions for every output, which will be used for all parameter combination. In the case of the OLS example, if we want to apply `mean()` on all estimated coefficients, but we want to use `var()` on the MC results of `sigma2`, the `sum_funs` should have the following structure:

```
# summarizing output the same way for each parameter combination with one function combination
sum_funs_ols <- list(b0 = mean, b1 = mean , b2 = mean, sigma2 = var)
```

Moreover, the user can specify any function provided it takes the output vector as only available argument.

Lastly for the last level of customization, a nested list named after the `nice_names` where every element follows the structure of the last example (components named after the outputs and each component is a function) can be specified. We present an example for this:

```
quantile_sum <- function(x) quantile(x, probs = 0.75)

# summarizing output differently for different parameter combinations
sum_funs2 <-
  list(
    list(b0 = quantile_sum, b1 = min, b2 = min, sigma2 = mean),
    list(b0 = mean, b1 = quantile_sum, b2 = mean, sigma2 = mean),
    list(b0 = median, b1 = median, b2 = median, sigma2 = mean),
    list(b0 = max, b1 = max, b2 = max, sigma2 = mean),
    list(b0 = min, b1 = min, b2 = min, sigma2 = quantile_sum),
    list(b0 = mean, b1 = mean, b2 = quantile_sum, sigma2 = quantile_sum)
  )
names(sum_funs2) <- first_mc_ols$nice_names
summary_out_param_spec <- summary(first_mc_ols, sum_funs = sum_funs2)
```

We would like to reiterate that the provided summary functions are not restricted regarding the complexity of their return value. However, the path of the summarized output over all simulation repetitions is only returned if the provided summary functions return a single numeric value. Thus, the following comfort functions `plot.summary.mc()` and `tidy_mc_latex()` will only work in this specific case.

Plot your Monte Carlo Simulation with `plot.mc()` and `plot.summary.mc()`

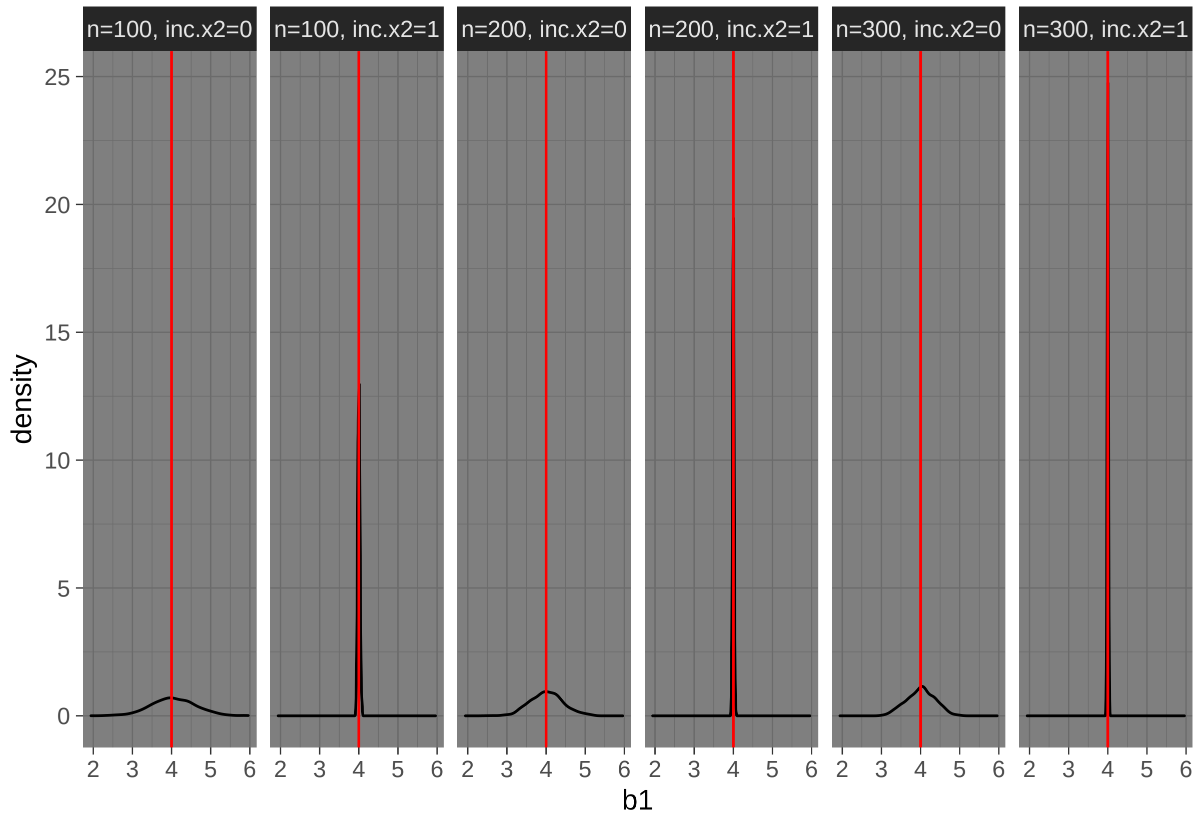
If `fun` returns a named list of scalars the user can use `plot.mc()` to generate a list of objects of class `gg` and `ggplot2` for all Monte Carlo results. The first argument of the function is an object of class `mc` returned by `future_mc()`. Using the argument `plot` the user can indicate whether the generated plots should be printed immediately or only returned in a list. The list will contain one plot for each output of `fun` comparing the results of the different simulation setups. In general, `plot.mc()` generates density plots for numeric outputs and bar plots for non-numeric outputs. In our example a plot for `b0`, `b1`, `b2`, and `sigma2` will be returned in a list of length four and as `b0`, `b1`, `b2`, and `sigma2` are all numeric outputs `plot.mc()` will return density plots for each of those:

```
mc_ols_plot <- plot(first_mc_ols, plot = FALSE)
names(mc_ols_plot)
```

```
## [1] "b0"     "b1"     "b2"     "sigma2"
```

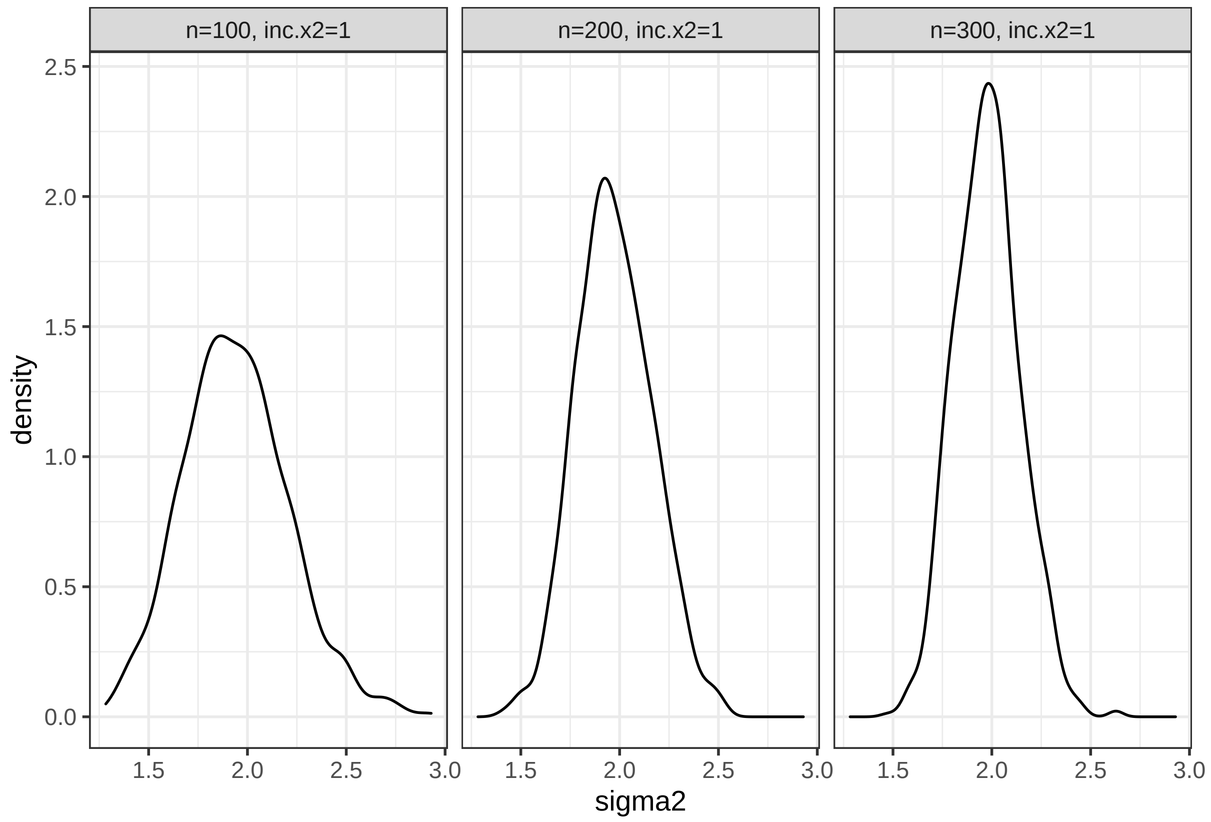
As the single list elements are of class `gg` and `ggplot2`, we can easily customize and extend the single plots using familiar `ggplot2` commands:

```
mc_ols_plot$b1 +
  ggplot2::geom_vline(xintercept = 4, col = "red") +
  ggplot2::theme_dark()
```



When creating the plots the user can also subset the setups which he/she would like to see in the plots using the `first_mc_ols$nice_names` in the function argument `which_setup`, or a named list in `parameter_comb`. The single components of the list have to be named after the parameters specified in `param_list` and contain vectors specifying the values of the parameters to filter by. In the ols example we can filter by the parameters `n` and `inc.x2`:

```
# subsetting by nice_names
mc_ols_plot_subset1 <-
  plot(first_mc_ols, plot = FALSE, which_setup = first_mc_ols$nice_names[4:6])
#subsetting by parameter values
mc_ols_plot_subset2 <-
  plot(first_mc_ols, plot = FALSE, parameter_comb = list(inc.x2 = 1))

mc_ols_plot_subset1$sigma2
```

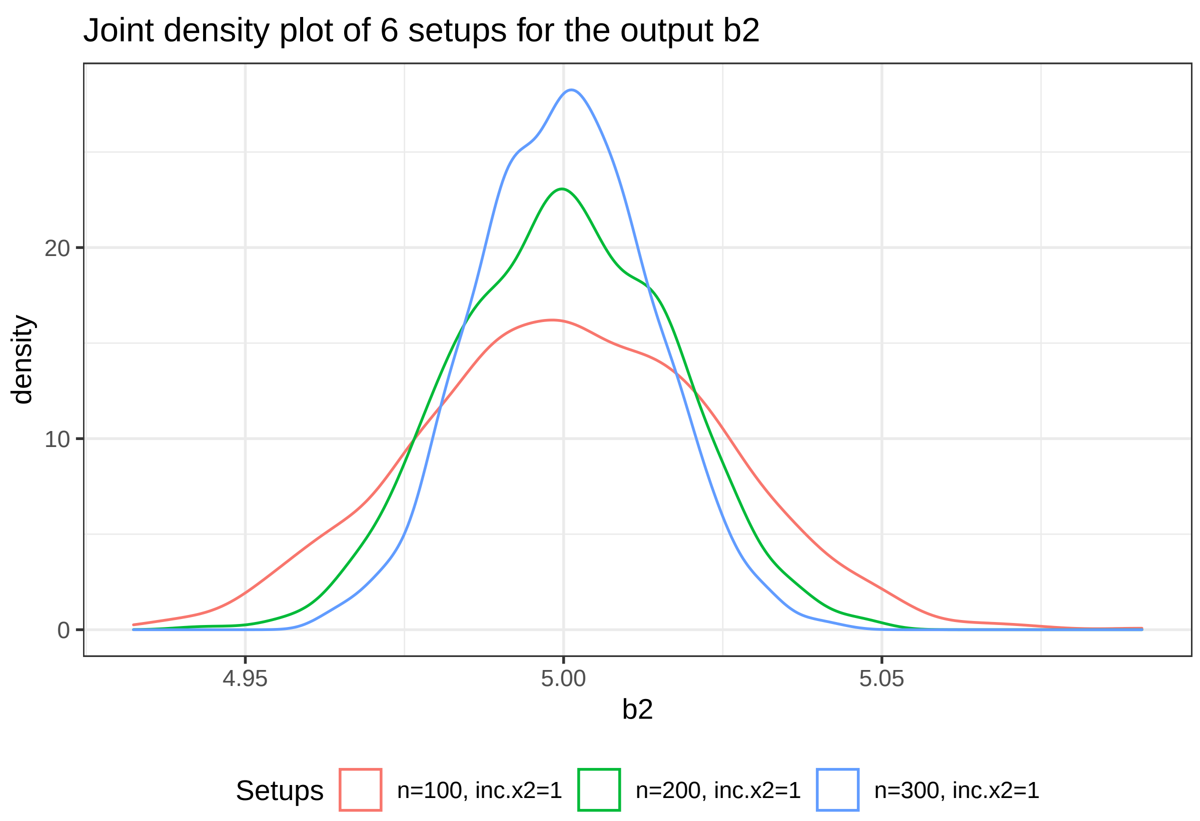


Thus, if the user wants distinct plots for every parameter combination, one needs to subset for the plot for any single setup in `first_mc_ols$nice_names`.

Finally, you can also plot the simulation results for several parameter combination in one single plot by specifying the argument `join` with the respective `first_mc_ols$nice_names`:

```
mc_ols_plot_joint <- plot(first_mc_ols, plot = FALSE, join = first_mc_ols$nice_names)
mc_ols_plot_joint$b2
```

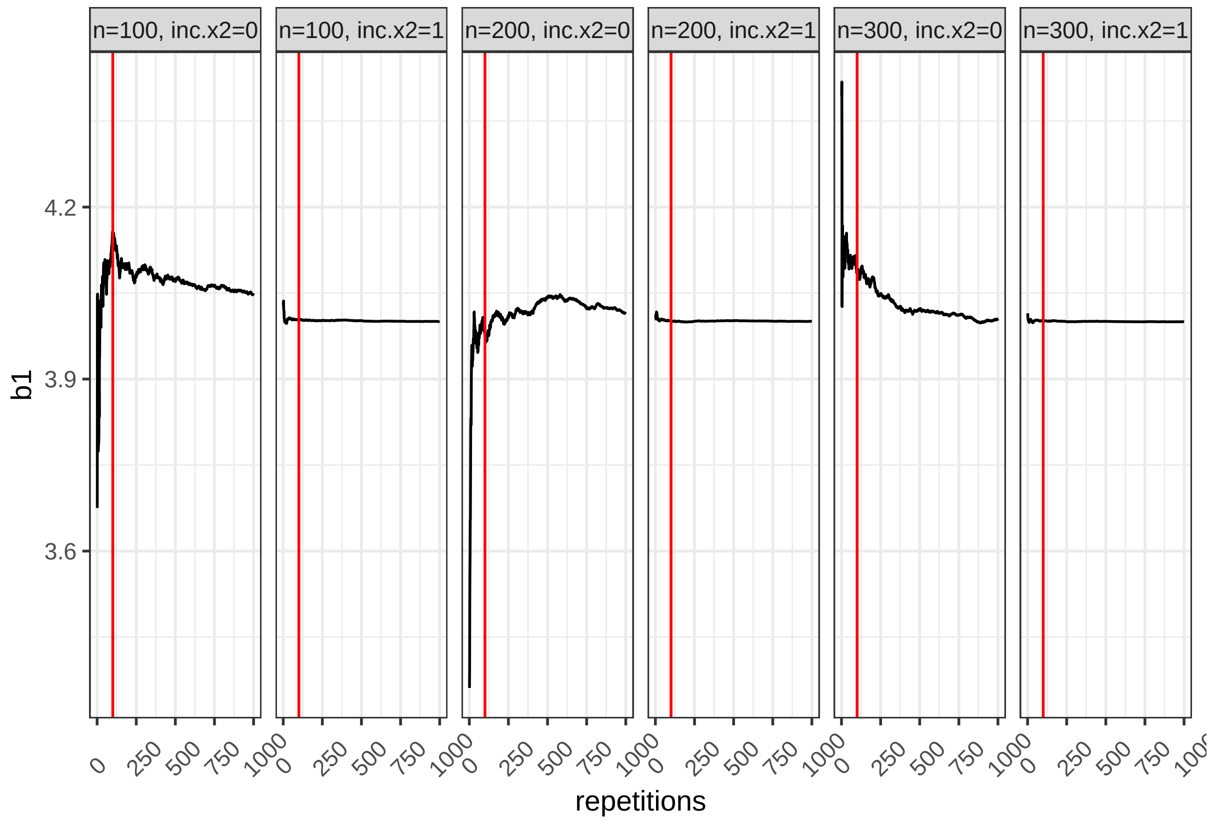
```
## Warning: Removed 3000 rows containing non-finite values (stat_density).
```



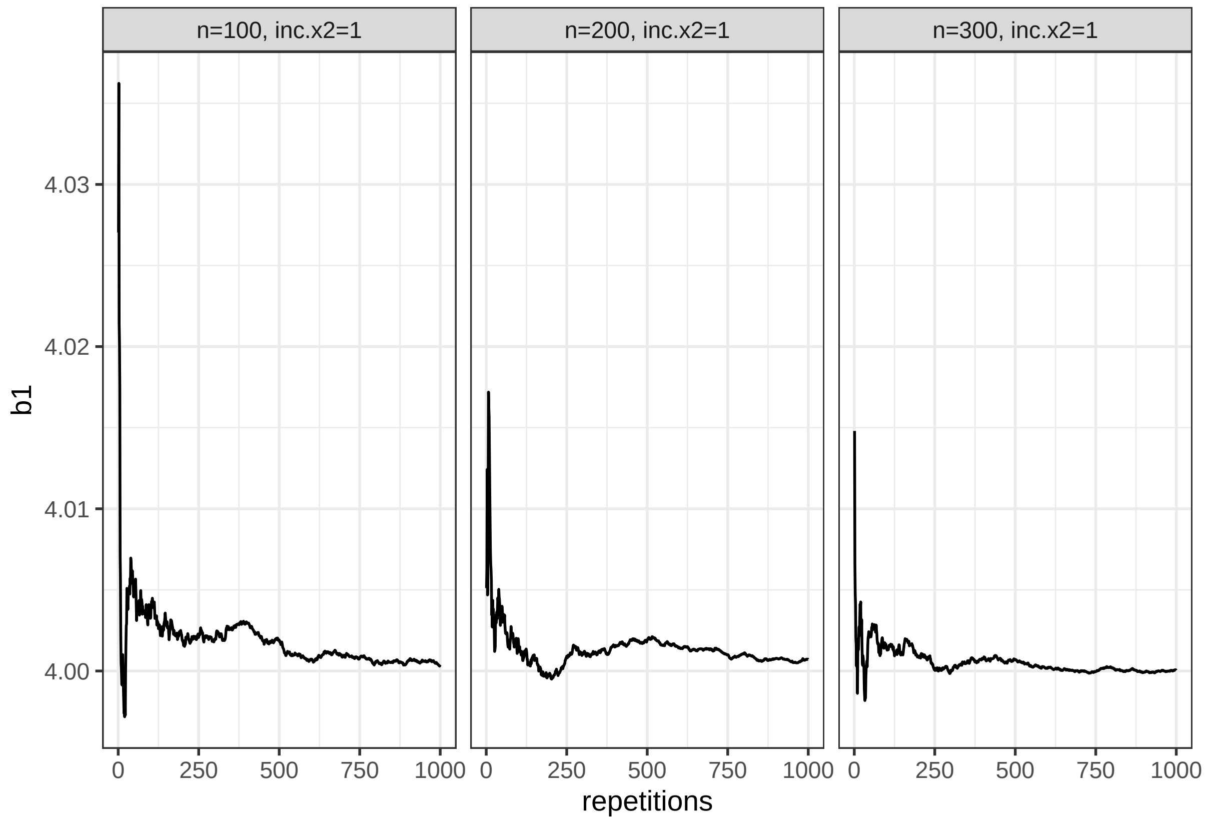
Please be aware that the only one of the three arguments `which_setup`, `parameter_comb`, and `join` can be specified at the same time.

If the provided summary functions in `summary.mc()` return a single numeric value and thus a path of the summarized output over all simulation repetitions is returned, the user can use `plot.summary.mc()` to plot those paths. The first argument of the function is an object of class `summary.mc` returned by `summary.mc()`. Just as `plot.mc()`, `plot.summary.mc()` returns a list of objects of class `gg` and `ggplot2`. The list will contain one line plot for each output of `fun` displaying the paths of the results of the different simulation setups. The arguments `plot`, `which_setup`, `parameter_comb`, and `join` can be specified the same way as for `plot.mc`:

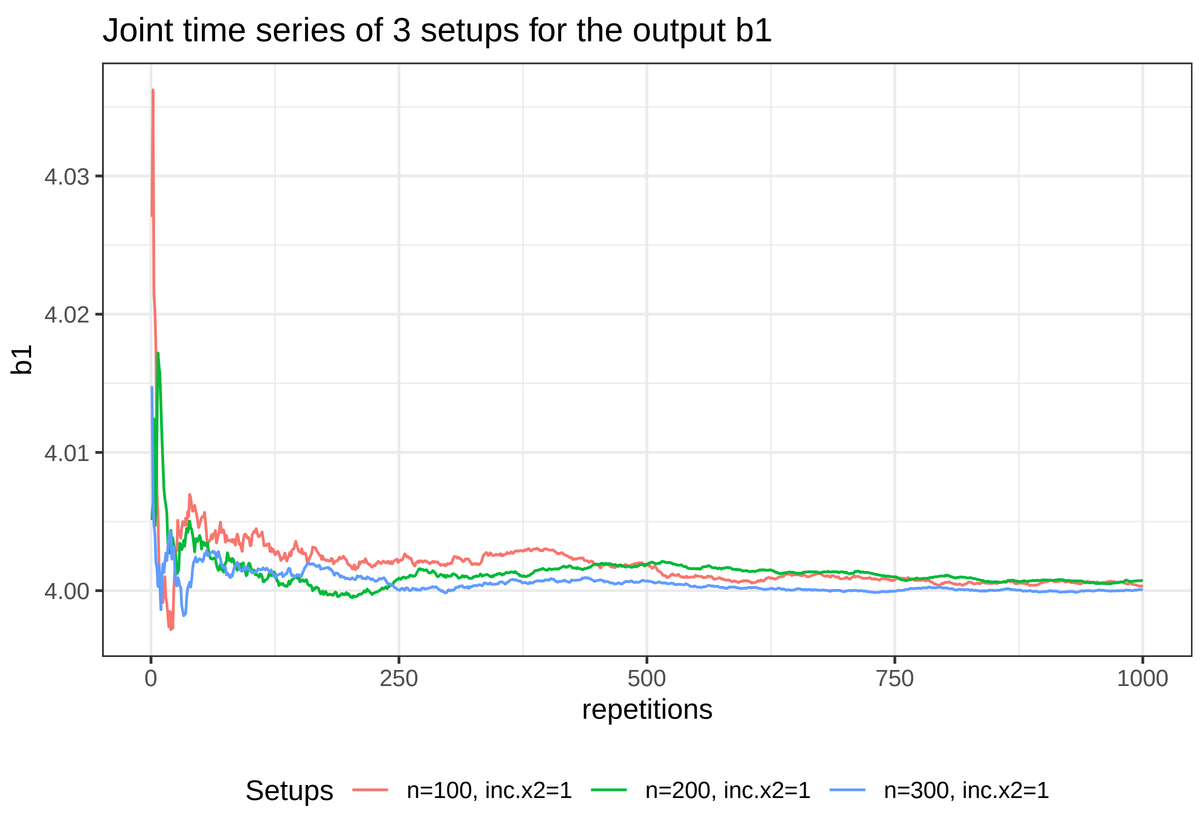
```
sum_mc_plot <- plot(summary_default, plot = FALSE)
sum_mc_plot$b1 +
  ggplot2::geom_vline(xintercept = 100, col = "red") +
  ggplot2::theme(axis.text.x = element_text(angle = 45, hjust = 0.1, vjust = 0.2))
```



```
sum_mc_plot_subset1 <-
  plot(summary_default, plot = FALSE, which_setup = first_mc_ols$nice_names[4:6])

sum_mc_plot_subset2 <-
  plot(summary_default, plot = FALSE, parameter_comb = list(inc.x2 = 1))

sum_mc_plot_subset2$b1
```



```
sum_mc_plot_joint <-
  plot(summary_default, plot = FALSE, join = first_mc_ols$nice_names[4:6])

sum_mc_plot_joint$b1
```



Create a LaTeX table of your results with `tidy_mc_latex()`

Using `tidy_mc_latex` the user can present the results from `future_mc` directly into a LaTeX document using all the benefits from the `kableExtra` package. The first and main argument `x` needed by `tidy_mc_latex` is a `summary.mc` object obtained from `summary.mc()`. To present the results in a comprehensive manner the function requires that all summarized outputs in `summary.mc` be scalar numeric results for all parameter combinations. In case, the summarizing function returns more than one argument then this will be presented in the table as an `NA` value. The second argument of the function is `repetitions_set` which allows the user to see extract the certain values of the "path" of the summarized results of `fun`. To illustrate this we use the MC results for the OLS example:

```
tidy_mc_latex(
  x = summary(first_mc_ols),
  repetitions_set = c(10, 1000)
) %>%
  kableExtra::kable_styling(latex_options = "HOLD_position")
```

Table 2: Monte Carlo simulations results

n	inc.x2	b0	b1	b2	sigma2
N = 10					
100	0	1.404	3.789	NA	927.992
100	1	0.974	4.000	5.007	2.098
200	0	1.143	3.832	NA	886.101
200	1	0.977	4.013	5.003	1.920
300	0	1.458	4.098	NA	874.542
300	1	1.034	3.999	4.990	1.915
N = 1000					
100	0	0.998	4.047	NA	892.937
100	1	0.998	4.000	5.001	1.943
200	0	1.004	4.014	NA	897.245
200	1	0.999	4.001	5.000	1.974
300	0	0.985	4.005	NA	896.994
300	1	0.999	4.000	5.000	1.978

Total repetitions = 1000, total parameter combinations = 6

The resulting table is composed of two panels, which corresponds to the length of `repetitions_set`. In them the columns correspond to the results of the summarizing functions for `b0`, `b1`, `b2`, and `sigma2`, and the rows correspond to specific combinations of the parameters provided in `parameter_list`. The footnote in the table shows the number of repetitions and the total parameter combinations provided to `future_mc`.

Moreover, the next three arguments in `tidy_mc_latex` are comfort options to select which results of `summary.mc` depending on the parameter combinations will be presented in the table. On one hand, The argument `which_setup` allows the user to make use of the `nice_names` of the parameter combinations in the returned object by `future_mc()` to subset the rows in the table. On the other hand, the argument `parameter_comb` is used to directly filter the parameters by their values. This argument requires a named list, containing vector or scalar values of all parameters to be filtered from. The user must only provide one of this arguments at a time. We show how to make use of both parameters to subset the rows of table for $n = 100$ and $inc_{x2} = 1$:

```r
tidy_mc_latex(
  x = summary(first_mc_ols),
  repetitions_set = c(10, 1000),
  which_setup = first_mc_ols$nice_names[1]) %>%
  kableExtra::kable_styling(latex_options = "HOLD_position")
```

Table 3: Monte Carlo simulations results

n	inc.x2	b0	b1	sigma2
N = 10				
100	0	1.404	3.789	927.992
N = 1000				
100	0	0.998	4.047	892.937

Total repetitions = 1000, total parameter combinations = 6

```r
tidy_mc_latex(
  x = summary(first_mc_ols),
  repetitions_set = c(10, 1000),
  parameter_comb = list(n = 100, inc.x2 = 1)) %>%
  kableExtra::kable_styling(latex_options = "HOLD_position")
```

Table 4: Monte Carlo simulations results

n	inc.x2	b0	b1	b2	sigma2
N = 10					
100	1	0.974	4	5.007	2.098
N = 1000					
100	1	0.998	4	5.001	1.943

Total repetitions = 1000, total parameter combinations = 6

The user can also subset the outputs of the original function (columns in the table) using the parameter `which_out`. This is done using a character vector with the names of the outputs, e.g. to only show the columns for `b0` and `sigma2` the user needs to set `which_out = c("b0", "sigma2")`.

Lastly, by providing a named list to the argument `kable_options`, the user can change all arguments of the underlying function `kable::kbl()`. The names of the list have to be equal to the names of the arguments of the function and the contents of every element also has to fulfill its requirements. We provide an example of how this list should be constructed, but for optimal usage we strongly recommend the user to see the documentation of the `kable::kbl()` function.

To allow for further customization the returned object by `tidy_mc_latex` is of class `knitr_kable`, therefore the user can utilize most functions from the `kableExtra` package in the standard tidyverse manner. For example:

```r
tidy_mc_latex(summary(first_mc_ols),
              repetitions_set = c(10, 1000),
              kable_options = list(
                col.names = c("Number of observations",
                              "$x_2$ included or not",
                              "$\\beta_0$", "$\\beta_1$",
                              "$\\beta_2$", "$s^2$"),
                caption = "Ommited variable bias MC results",
                escape = FALSE
              )
) %>%
  kableExtra::kable_styling(latex_options = "HOLD_position")
```

Table 5: Ommited variable bias MC results

Number of observations	x_2 included or not	β_0	β_1	β_2	s^2
N = 10					
100	0	1.404	3.789	NA	927.992
100	1	0.974	4.000	5.007	2.098
200	0	1.143	3.832	NA	886.101
200	1	0.977	4.013	5.003	1.920
300	0	1.458	4.098	NA	874.542
300	1	1.034	3.999	4.990	1.915
N = 1000					
100	0	0.998	4.047	NA	892.937
100	1	0.998	4.000	5.001	1.943
200	0	1.004	4.014	NA	897.245
200	1	0.999	4.001	5.000	1.974
300	0	0.985	4.005	NA	896.994
300	1	0.999	4.000	5.000	1.978

Total repetitions = 1000, total parameter combinations = 6

Comparison to other Monte Carlo Approaches in R

Design of the functions

As already stated our conceptualization of an MCS implementation is based upon the basic requirements of this technique and framing the functions into the `tidyverse` structure. Previously, the `MonteCarlo` package already developed a convenient framework for the study and paralellised implementation of MCS, to which we compare the structure of our package. The main function for running a MCS in the `MonteCarlo` is the function of the eponymous name.

Both our `future_mc` and the `MonteCarlo` functions have two main divisions on their structure. First, we have the set-up of the arguments, data containers and error checking, parts of the function. Which can be called as the preamble of the simulation. Lastly, the actual simulation of the MC study over a parameter grid specified by the user. The main differences between our implementation and the `MonteCarlo` packages lay in these two steps of the functions.

On one hand, the whole development of our package draws from the benefits of the tidyverse functions. For example: the creation of parameter grids, the structure of the data and repeated calculations, rely heavily in functions provided by the `stringr`, `tibble` and `purrr` packages. Most importantly, for all repeated calculations that normally are handled by for loops we implement them with their `purrr` counterparts. These are all implemented keeping in mind the type of the object to be handled in order to reduce computation time, i.e. the mapping functions are tailored for dealing with specific data types. In contrast, the `MonteCarlo` function is built within a base `R` framework. Moreover, to offer flexibility to the user, the function uses a high number of strings for the creation of for loops, containers and parameter grids which are then parsed and evaluated inside the function.

On the other hand, when it comes to the implementation of the paralellised simulation both functions fifer totally from each other. The `MonteCarlo` function is built around the implementation of the `snow` and `snowfall` packages. These packages let the user perform calculations by in an master/worker scheme, this means that the main function call distributes tasks to the "workers" (CPUs). Thus, all workers resolve the tasks they are given and convey their results to the main `R` session giving the instructions. In contrast, `future_mc` draws its computing power from the `future` package through the `furrr` package API. The `future` package conceptualizes parallel computation based on task processing constructs called `futures` which are tasks that are resolved in background process while the main session continues to run. The benefit of this is twofold, for one all tasks are centralized in a queue and are supplied to the workers as soon as they are available which keeps calculation times consistent accross workers. For the other, the main session can still delegate tasks and perform operations, provided that no results are required from the unresolved futures. In the design of `future_mc` we reap these benefits through the use of the `furrr::pmap()` function, to which we provide the user's function and hes/hers desired parameter grid replicated as many times as the number of MonteCarlo repetitions.

Package Principles {MCS}

In developing the `tidyMC` package, we followed the principle of "fail early and often", i.e. if a function is misspecified, we want our function to fail as early as possible and also return readable and informative error messages. To achieve this goal, the first part of each function in our package consists of assertions to the input parameters, as these are specified by the user and are therefore error-prone. After making sure that the input parameters are correctly specified, the following core of the function should work without problems and can thus be executed. In our package, we ensure the class and general structure of the input parameters by using the `checkmate` package, as the `checkmate` commands can almost always be written in a readable line of code, produce readable error messages and are written in C, so we don't have to worry about execution time overhead.

Since one of our main focuses is to embed our functions in `tidyMC` into the `tidyverse` structure, we have used functions from packages of the `tidyverse` whenever possible. This is particularly evident in the following three cases: First, we use `stringr::str_c()` instead of `paste()` as this is generally faster and more stable. Second, we mostly work with tibbles instead of data.Frames, as they provide more consistent subsetting behaviour and we take advantage of tibbles' higher flexibility by storing lists in columns. Thirdly, for all repeated calculations that are normally handled by for loops, we implement their `purrr` counterparts. These are all implemented taking into account the type of the object being handled to reduce computation time.

To work collaboratively on our package code, we use the version control system `Git` and manage our Git project with `GitHub`. This ensures that our pieces of code are merged correctly and nothing gets lost along the way. It also allows us to track our development over time and easily revert to older versions of our package. We also relied heavily on Hadley Wickham's book, which is a great summary of best practices in package programming. As recommended by Wickham, when we wanted to use functions from another package, we added the appropriate package to the `NAMESPACE` file with `usethis::use_package()` and then used the form `package::function()` when calling a function from another package in our code. This way, all dependencies are clear to the reader. To reduce package dependencies outside the `tidyverse`, we tried to avoid using packages that are not part of the `tidyverse` whenever possible.

Our typical development workflow (inspired by Hadley Wickham) looks as follows:

1. we implement the desired feature in our code relying on `tidyverse` functions
2. we use `devtools::load_all()` to make our (updated) function available for experimentation
3. we run an example to check whether the functionality is implemented correctly

4. we try to take care about the function documentation using `roxgen2` right away, as at this point all requirements for the function and the functionality are best in our mind
5. we run `devtools::check()` and fix all occurring notes, warnings, and errors
6. a group member who was not involved in the coding runs several examples using the function to test its functionality 6.1 Whenever a bug occurs, it is reported to the developer of the function, who takes care of fixing the bug

6.2 the bug is converted into a failing test (using `testthat`) and the developer's task is to make this test pass. This test ensures that the same bug will not occur a second time.

Speed comparison

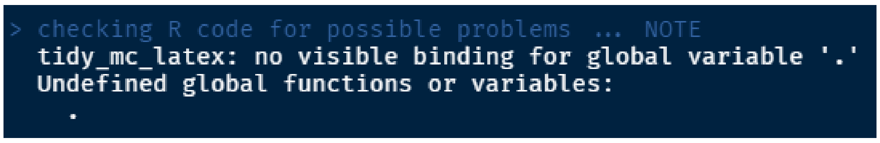
Conclusion

With the `tidyMC` package we provide researchers with an easy and comfortable way to perform their own Monte Carlo Studies. We provide a general setup, that easily allows for Monte Carlo Studies in all possible flavors. In this report, we used OLS to demonstrate how `tidyMC` may be used to efficiently set up Monte Carlo Simulations over a parameter grid with parallelization. Building on the `tidyverse`, the `tidyMC` package uses the powerful `furrr` package to efficiently parallelize the Monte Carlo study over an arbitrarily complex parameter grid. We compared the `tidyMC` approach to R's currently most popular package for Monte Carlos Simulations, the `MonteCarlo`package, as well as two alternative approaches. We find that `tidyMC` generally outperforms the alternatives in terms of speed and memory required. Especially with larger Monter Carlos Studies using more parallel workers, our implementation is much quicker and much less memory intensive. To ease getting a quick overview of the simulation results we provide a comprehensive summary function. Lastly, to allow easy communication of the Monte Carlo results, we provide a build in plot function and a function to generate Latex Tables. Both functions are highly customizable, allowing the user to tailor the output to his needs. Because our package is built solely on top of other `tidyverse` packages, it integrates very nicely into the package collection. However, its easy usage should benefit all types of `R`user.

While we have implemented many different options for the user, there is still room for extensions. Right now the plot function only allows for plotting densities. In a future version, we may add further plotting options. Furthermore, it may be beneficial to add predesigned options for different kinds of flavors of Monte Carlo Simulations. Our current implementation is as general as possible. Specifying the plot and table functions for specific use cases would likely lead to a more comfortable user experience. For example, if different estimators are compared, it might make sense to add preset plot functions that more effectively allows visualizing the differences in performance. However, given the level of generality for which we designed that package, we believe that its current implementation may nonetheless lead to good results for all potential users.

Unresolved Issues

With the help of various online resources, especially Stackoverflow, we were able to fix all errors, warnings and notes returned by `devtools::check()` in a satisfactory way (for us), except for one:



```
> checking R code for possible problems ... NOTE
  tidy_mc_latex: no visible binding for global variable '.'
  Undefined global functions or variables:
    .
```

Figure 3: Unresolved Note.

If this note occurs in the context of a `dplyr` function, the problem can easily be solved by using the `rlang::.data` pronoun after adding `rlang .data` to the `@importFrom` tag in your `roxygen2` header, as explained e.g. in this post. However, if this note does not occur in the context of a `dplyr` function, but as in our case with `purrr::map()`, there is (at least to our knowledge) no such nice solution, as also this stackoverflow post is not answered. Thus, we decided to use the second option presented in the (R-blogger post)[https://www.r-bloggers.com/2019/08/no-visible-binding-for-global-variable/] and set

```
. <- NULL
```

at the respective part in the code. This solution is often used when working with the `data.table` package and deals with the note in the RCMD-check as the variable `.` then has a visible binding (to `NULL`). However, it feels a bit "hacky" to us and we would like to know if there is another more natural solution to that problem.