Reproducibility Limits of Mixed-Integer Linear Programming-based methods

Rémi Garcia

Reproducibility Limits of Mixed-Integer Linear Programming-based methods

RÉMI GARCIA, Univ Rennes, Inria, CNRS, IRISA, France

"Do not trust computers," this sentence has probably been uttered by most, if not all, computer scientists at some point. At the same time, computer scientists are building version control tools, databases and many other tools to help them reproduce certain behavior of a software library, with the ultimate goal of trusting the results of computations. In this paper, we propose to look at issues that can arise when trying to reproduce the results of Mixed-Integer Linear Programming (MILP)-based methods, from the user perspective. These methods rely on external tools which introduce new parameters when dealing with reproducibility issues. In particular, the phenomenon of performance variability has a significant impact on execution times, which is currently underestimated. In this work, we question the meaning of reproducibility in the context of the MILP approaches and we propose ways to reduce output dependence on solver uncertainties.

CCS Concepts: • **Computing methodologies** → *Modeling and simulation.*

Additional Key Words and Phrases: Mixed-Integer Linear Programming, Solvers, Reproducibility, Mathematical modeling

1 INTRODUCTION

For the last few decades, research, including computer science research, has been going through a so-called "replication crisis" [8, 10]. The awareness around this issue has a positive impact on research results, and more and more published papers pay greater attention to reproducibility. Giving more details about the proposed algorithms and experiment parameters is important for reproducibility, but not sufficient. In some applications it is difficult to specify and obtain the full set of parameters required to ensure replicability. Hence we will mostly focus on reproducibility aspects in this paper and we will follow the definition of McDougal *et al.* [21] when using the terms reproducible and replicable:

- Replication consists in finding the exact same results as a previous work and/or experiment;
- Reproduction consists in obtaining similar results.

We could expect that, with all the tools at their disposal to archive and share a certain version of a software, computer scientists should have replicable results, but even when combined with open-source code, published results are often neither replicable nor reproducible [22].

Multiple papers [13, 16, 19] address the question of reproducibility in models and simulations, but we believe that specific issues that arise with Mixed-Integer Linear Programming (MILP) approaches have not yet been discussed. Basically, solving a problem with this approach consists in writing a mathematical model, *i. e.*, a definition of the problem instead of an algorithm, which is given to an external tool to obtain a solution. Hence, an important particularity of MILP-based methods is that they are meant to evolve with solvers.

This raises the question of reproducibility for MILP-based methods. We propose to loosely define it based on the following reasonable expectations:

(1) Running the same method under similar experimental conditions should give similar results.
(2) Results should not degrade when using better hardware and/or solvers, *e. g.*, when trying to reproduce the approach years later.

Actually, this does not really differ from the expectations we would have for algorithms. The main difference is that, over time, algorithms do not change while the methods used to solve for mathematical models will evolve independently of the models. Indeed, even when they rely on subroutines that could have been updated, algorithms' performance

Author's address: Rémi Garcia, remi.garcia@irisa.fr, Univ Rennes, Inria, CNRS, IRISA, Rennes, France.

Table 1. Published and reproduced solving times in seconds for 11 instances. Values are rounded up to the nearest integer and TO for timed-out of 600s.

Instances	1	2	3	4	5	6	7	8	9	10	11
Published	<1	29	<1	<1	5	<1	2	66	TO	<1	6
Reproduced	<1	476	<1	2	562	<1	18	TO	TO	<1	180

should only differ marginally whereas solving the same mathematical model decades later might lead to very different solutions and solving times [18].

On the basis of the above definition, we propose experiments showing that, in some cases, our expectations are not met when actually reproducing our original experiments. For these experiments, we use two MILP-based models for the Single and Multiple Constant Multiplication problems [3, 6, 14]. The exact definition of our models is of little relevance for this work and we will refer to them as "Model A" and "Model B" in the following[1].

First, we will show that the phenomenon of performance variability [9, 17, 20] can have a huge unexpected impact, limiting reproducibility of MILP-based approaches. Then, we will show that even using open-source code for the mathematical model generation does not prevent unpredictable behaviors.

2 TRYING TO REPRODUCE MILP-BASED MODELS

In the following we will show that, while actually reproducing the execution of an MILP-based model, it is possible to obtain unexpected results. In practice, such results would probably lead researchers to believe that they did not reproduce the original model, or that the published results may have been falsified.

For our purpose, we will use two computers, both running on Ubuntu 22.04.3 LTS. Computer $\mathcal{A}$ has an Intel® Core™ i7-1365U CPU and Computer $\mathcal{B}$ has an Intel® Core™ i3-8100T CPU. Model A will be solved on Computer $\mathcal{A}$ using 4 threads and a limit of 8 Go of RAM with time limit of 600s for the solver. As we will not focus on performances when solving Model B, we do not limit resources. In both cases, we will use the MILP solver CPLEX [7], in particular versions 22.1.1 and 12.10. To generate our two models, we will rely on Julia 1.6.7 and the modeling language JuMP [11] v1.18.1 associated with the Julia package CPLEX v1.0.2. Both models we will be using solve Constant Multiplication problems [3, 6, 14].

2.1 New implementation

The first case we consider is the following: suppose we want to reproduce published results implementing the mathematical model provided in a paper. Our tools and hardware specifications might differ, as well as the solvers. Obviously, we do not expect to obtain the exact same results but at least a similar trend. With the next example, we show that this is not necessarily true, even when only applying seemingly performance-neutral changes. All the following experiments were conducted on Computer $\mathcal{A}$ using CPLEX 22.1.1.

In Table 1, we report what could be published results for Model A on 11 instances and what could be our results when reproducing the experiments. To obtain these results, we used the same computer and software environment. The only difference is the order in which the constraints are passed to the solver. To do so, we implemented a Julia package, ShuffleConstraints[2] and solved each instance multiple times, shuffling the order of the constraints each time.

[1] Models and experimental results are available at https://remigarcia.xyz/research_experiments/reproducibility_solvers2024.zip
[2] https://github.com/remi-garcia/ShuffleConstraints

By definition, we reproduced the MILP model, all the results have been obtained using the same experimental environment and the same mathematical model. Nevertheless, our results are clearly worse than the published ones. Although for 6 instances out of 11 we obtained similar results, this is only the case for very simple problems (< 1) or the most difficult ones (TO). For the other instances, our solving times, on the same experimental environment were 6 times slower in the best case and up to 100 times slower for the 5th instance. Even if less than half of the instances were impacted, in our example, the impacted ones were those that were not very simple nor too difficult to solve, corresponding to use cases that are most likely to occur in practice.

Note that, for the sake of this demonstration, we actually cherry-picked best and worst solving times over 50 constraint orderings. However, large differences are not impossible over a single run. Hence, it is necessary to consider that unexpected large differences in terms of solving time are possible between the original published results and our attempt at reproducing them.

Furthermore, we should envisage that a certain constraints' order leads to a large solving time for a very simple instance or similarly to greatly simplify a very difficult instance. While not shown in Table 1, instance 5 was solved in less than a minute for 36 orderings of the constraints, whereas it took more than 4 minutes for 2 other runs. Similarly, instance 8 timed-out or reached the memory limit 41 times and was solved to optimality only 9 times. Hence, a very impressive result on an instance could be serendipity and should not be discarded as falsification if not reproducible.

Called performance variability [9, 17, 18], the phenomenon of "unexpected changes in performance that are triggered by seemingly performance-neutral changes" [17] is not sufficiently considered when publishing or reproducing results based on MILP solvers. As MILP users, trying to figure out the internal behavior of the solver and the origin of these differences is not realistic. Actually, the term "performance variability" was coined by Danna [9] when working on CPLEX, and even with this deep knowledge about solvers, performance variability remained impossible to completely understand and mitigate. The operations research community explains it by the fact that some changes will impact branching order, at least for tie-breaking cases [18]. The fact that the solver was not able to determine best the branching order, *i. e.*, that tie-breaking is necessary, illustrates that it is very difficult to predict the effect of the branching. This leads to our first proposal:

Recommendation 1: Authors reporting run time results relying on MILP solvers should instead report statistics on run times. For example by randomizing the order of constraints, or using other seemingly performance-neutral techniques that could trigger run times differences.

Many solving time results have already been published without providing statistics and, in the future, we will probably continue to do so. Even though statistical significance is harder to assess, when trying to reproduce results we should also do multiple runs and test whether the published results could come from our implementation or not. To our knowledge, we still do not have a procedure to test statistical differences over MILP models, taking into account performance variability. We do not even know if performance variability induces normally distributed solving times or not and this still needs to be verified with normality tests such as Spiegelhalter's [24] or Shapiro-Wilk [23]. This is important as we cannot use certain significance tests if we do not have normally distributed solving times [12].

With the first case we considered, we actually could have expected that it would not be easy to reproduce the performance results of another person. In the following, we will discuss two other cases where we want to reproduce our own experiments and we will see that new issues arise.

2.2 Same implementation, same computer, new solver

As stated above, we expect that results should not degrade when using better hardware and/or solvers, *e. g.*, when trying to reproduce the approach years later. We observed that different implementations could lead to large solving time differences. Here, we consider a single implementation of Model B that we solve using CPLEX 12.10 and CPLEX 22.1.1 on Computer $\mathcal{A}$, all other thing being equal. We only report the result for one instance, thus we will loosely use "Model B" to refer to it directly.

Let us assume we have published a paper with Model B and obtained an optimal solution in less than a second with CPLEX 12.10. We still have everything available in the same state but we just installed CPLEX 22.1.1 and we want to check the experiments of our previous publication with the new version of the solver. Surprisingly, it takes less than a second for the new solver to return that Model B is infeasible. If we did not closely check our solutions at the time, we might trust the newest version of the solver and impute to the former version the error. Actually, solvers have multiple integrality tolerance parameters which could have led to a solution which did not fit some constraints.

Recommendation 2: Authors should always check feasibility of the returned solutions.

In the case of Model B, the issue is actually the opposite: the latest version introduced a bug. Stumbling across a solver bug is not uncommon but these bugs are often bypassed by chance and left misunderstood. Changing the order of the constraints of Model B or *adding* new constraints solves the problem: simply trying to find a bug usually leads to seemingly effect-neutral changes that fix it.

This raises the question of reproducibility when relying on MILP solvers, or closed-source software. To be fair, from the user perspective, we believe that the exact same problem of thinking that the issue came from the model, and fixing it by chance, would occur even if using open-source MILP solvers such as SCIP [4] or GLPK[3]. Indeed, while CPLEX 22.1.1 has *only* 189 parameters, SCIP has almost 3 thousands of them and this illustrates that it is certainly impossible to look into the origin of a certain behavior of the solver, even open-source ones, for nonspecialists. Thus, the problem is more global than just a closed-source issue.

In any case, our first recommendation would probably permit to avoid this: a different order of the constraints actually solved the problem every time we encountered it. Yet, some outliers can occur.

Recommendation 3: We should be aware of the existence of bugs in MILP solvers for which we have no control over. Thus, we should accept that absurd results will occur from time to time.

This recommendation does not mean that nothing should be done about erroneous results, but that it is important to consider that errors could come from solvers. If solving the same model, with various order of constraints, leads to different results, chances are the error comes from the solver. Hence, this illustrates the importance of recommendation 1.

We also suggest to experiment with deactivating the presolve step, in particular for simple instances which would probably still work well without presolving. Indeed, if the model is feasible but the solver returns that it is infeasible, this means that at some point the solver reduced the search space too much and, basically, presolving [2] is the only step that reduces the search space for the complete solving process.

An in-depth analysis of the model and instances is necessary to understand what is an absurd response from the solver. We should develop and use tools to check if the returned solution is actually valid and to discard necessarily

[3] https://www.gnu.org/software/glpk/glpk.html

incorrect answers. At the same time, these answers are important and the models that generated them should be stored and shared with the solver developers.

A category of bugs that we think is hard to catch is suboptimal solutions returned as optimal. The chances of this happening every time, when following recommendation 1, is small. Nevertheless, it might still happen.

Recommendation 4: Whenever possible, always store the best known solution to compare against the solver's.

With this example, we discussed a few possible bugs that could arise due to the solver. Even harder to catch are the ones which only happen with the right, or wrong, hardware/software combination.

2.3 Same implementation, same solver, new computer

We propose to consider the following third situation. Let us suppose that we worked on Computer $\mathcal{B}$ and produced an MILP model to solve a given problem, Model B. At some point, we decided to run our experiments on a different computer, *e. g.*, with more threads to be able to parallelize experiments. Doing so on Computer $\mathcal{A}$, we expect to reduce solving times as our second computer is "better" than the original one. Yet, Model B was solved in less than a second on the first computer and is now considered infeasible.

Basically, the situation is identical to the previous one. Hence, the same recommendations remain. However, we have a new information which could help to understand the origin of this behavior. *A priori*, algorithms are the same and the difference between the two computers do not come from the solver. Thus, our hypothesis is that the difference is at a lower layer: the floating-point arithmetic.

The IEEE-754 standard for floating-point arithmetic [1] leaves some room for variability. Indeed, correct rounding is not always mandatory, nor is faithful rounding for complex functions, and this obviously has an impact on computations [5, 15]. Although this answer is not really satisfactory, there are not many other differences between computers $\mathcal{A}$ and $\mathcal{B}$ that could have this impact.

Recommendation 5: Errors could come from floating-point arithmetic. Thus, we should accept that absurd results will occur from time to time.

We are not aware of any tool that could help researchers determine if two computers handle similarly basic arithmetic operators in floating-point. In the case we just described, this tool would be very useful as it could confirm that the difference is certainly at the CPU level.

Differences between CPUs are not limited to floating-point operator implementations. The number of threads is also relevant for MILP solvers as default behavior of CPLEX is to exactly use the number of available threads[4], 12 for Computer $\mathcal{A}$ and 8 for Computer $\mathcal{B}$. It might be of interest to change the number of threads, in particular to single threading.

Recommendation 6: Doing experiments with a single thread permits to verify that the bug does not come from multi-threading. In many cases, single thread solving is as efficient as using more threads.

We solved Model B using 1, 4, 12 and 16 threads, and only the default, solving the model with 12 threads, led to an erroneous infeasible response. This new information challenges our hypothesis. Instead of coming from the floating-point arithmetic, the error could be a multi-threading bug from the solver. Actually, the error could come

[4]Actually, CPLEX fixes the default number of threads to the minimum between 32 and the number of available threads.

from the precise combination of number of threads and floating-point arithmetic as using 12 threads on Computer $\mathcal{B}$ works fine.

We explored two possible reasons explaining the error we observed. There are many other potential explanations for this bug and, as mathematical model users, we will probably never be able to examine the solver, even if open-source, to understand the definite origin of the error.

3 TOOLS AND TECHNIQUES TO MEET OUR RECOMMENDATIONS

The possibilities of failure are endless and our time to understand and fix them is limited. We suggest the development of tools to compensate for solver errors and uncertainty. Thus, instead of trying to resolve all the bugs and totally remove the performance variability phenomenon from solvers, which is probably not possible in any case, we propose to use tools to detect and mitigate their effect.

In our first recommendation, we suggest to use performance variability to obtain statistics on solving times and to definitely drop one-time run. We see two possible tools that would help to apply this recommendation.

First, we have started to work on ShuffleConstraints. This Julia package takes a JuMP model and, shuffles the constraints by calling the `shuffle!` function on it. Doing multiple runs by shuffling the model before each solving, we are able to obtain a lot of variability in terms of solving times. We partially illustrated this in Table 1, but more work is necessary to determine if ShuffleConstraints gives a good representation of the performance variability phenomenon. Moreover, shuffling constraints actually leads to constraints' orderings that we would not produce by writing the model by hand, *e. g.*, we tend to group similar constraints together. Thus, patterns, which could be detected by solvers, will probably be harder to find after shuffling. For these reasons, ShuffleConstraints is probably not the perfect tool for recommendation 1, but we still view it as a step in the right direction.

A second type of relevant tool when doing multiple solver runs is one for gathering statistics. Comparing single run results is easier, and less meaningful, than rigorously comparing statistics. We are not aware of any easy tool that would work well with Operations Research specific needs which include results with time-out, various instance difficulties, outliers, bugs, etc.

Variability in solving times is not the only issue we encounter when reproducing MILP-based models. As we have highlighted in this paper, many unintended behaviors of the solver could arise. It seems difficult to provide a generic tool to discard incorrect infeasible answers as this would be specific to each problem and for each instance. However, a generic verification tool to check solutions should be possible.

Some numerical tolerances allow the solver to return solutions that might not perfectly fit the constraints. With floating-point variables, this is usually harmless but, in the case of integers it could lead to visible erroneous solutions. We are working on a Julia package built on top of JuMP, CheckSolve[5], to automatically check the difference between mathematical constraints and the solution provided by the solver. Currently, it permits to detect large errors and, in the near future, we are hoping to propagate smaller errors in order to detect more numerical errors.

Finally, we should not completely give up on limiting performance variability and unexpected behaviors using only external tools. Providing open-source code for model generation is a step in the right direction for more reproducibility but fine-tuning solvers could be as important. To limit the performance variability due to the branching order, we suggest to actually fix it. Using CPLEX, the `CPXcopyorder` parameter can be used to set a priority for variable branching, and a similar option usually exists for every MILP solver.

[5] https://github.com/remi-garcia/CheckSolve

Other parameters exist and fine-tuning the solver is probably a good way to reduce variability and to have a more deterministic solving process, which can help when looking under the solver hood. However, it comes at a cost as it requires some expertise in addition to possibly slowing down the solver since default values for the parameters are deduced depending on each instance.

4 CONCLUSION

In this paper, we stumbled across different unexpected behaviors of MILP solvers. By chance, we certainly avoided many other possible issues. In any case, when reproducing someone else's work, and even our own work, chances are that we will meet many strange behaviors that will make us question the reproducibility of published MILP-based results. The main recommendation of this paper is probably the following:

Recommendation 7: Do not trust computers.

Although we do not make a case for giving up on hopes for reproducibility of MILP-based approaches, we believe that we have to re-evaluate our expectations and to take randomness into account. Currently, no out-of-the-box tools exist to statistically compare implementations, to automatically discard erroneous values, to compare solvers behavior depending on the hardware, etc. This paper provides multiple reasons in favor of the development of these tools.

While we provided the state of our work on such tools, multi-disciplinary expertise is required if we want to be able to trust them. Hence, it is important for the Operations Research community to interact with other communities, *e. g.*, we need better knowledge on Computer Arithmetic and Statistics.

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. https://doi.org/10.1109/ieeestd.2019.8766229

[2] Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. 2016. Presolve Reductions in Mixed Integer Programming. *INFORMS Journal on Computing* (2016). https://doi.org/10.1287/ijoc.2018.0857

[3] Robert Bernstein. 1986. Multiplication by integer constants. *Software: Practice and Experience* 16, 7 (July 1986), 641–652. https://doi.org/10.1002/spe.4380160704

[4] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. 2021. *The SCIP Optimization Suite 8.0.* Technical Report. Optimization Online. http://www.optimization-online.org/DB_HTML/2021/12/8728.html

[5] Sylvie Boldo and Guillaume Melquiond. 2008. Emulation of a FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd. *IEEE Trans. Comput.* 57, 4 (2008), 462–471. https://doi.org/10.1109/tc.2007.70819

[6] Andrew D. Booth. 1951. A Signed Binary Multiplication Technique. *The Quarterly Journal of Mechanics and Applied Mathematics* 4, 2 (1951), 236–240. https://doi.org/10.1093/qjmam/4.2.236

[7] CPLEX. 2024. CPLEX User's Manual. https://www.ibm.com/analytics/cplex-optimizer

[8] Sharon M. Crook, Andrew P. Davison, and Hans E. Plesser. 2013. *Learning from the Past: Approaches for Reproducibility in Computational Neuroscience.* Springer New York, 73–102. https://doi.org/10.1007/978-1-4614-1424-7_4

[9] Emilie Danna. 2008. Performance variability in mixed integer programming. In *Workshop on Mixed Integer Programming.* https://coral.ise.lehigh.edu/mip-2008/talks/danna.pdf

[10] Chris Drummond. 2009. Replicability is not reproducibility: nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, Vol. 1. National Research Council of Canada Montreal, Canada.

[11] Iain Dunning, Joey Huchette, and Miles Lubin. 2017. JuMP: A Modeling Language for Mathematical Optimization. *SIAM Rev.* 59, 2 (May 2017), 295–320. https://doi.org/10.1137/15M1020575

[12] Stephen E. Edgell and Sheila M. Noon. 1984. Effect of violation of normality on the t test of the correlation coefficient. *Psychological Bulletin* 95, 3 (May 1984), 576–583. https://doi.org/10.1037/0033-2909.95.3.576

[13] Ben G. Fitzpatrick. 2018. Issues in Reproducible Simulation Research. *Bulletin of Mathematical Biology* 81, 1 (Sept. 2018), 1–6. https://doi.org/10.1007/s11538-018-0496-1

[14] Rémi Garcia and Anastasia Volkova. 2023. Toward the Multiple Constant Multiplication at Minimal Hardware Cost. *IEEE Transactions on Circuits and Systems I: Regular Papers* 70, 5 (2023), 1976–1988. https://doi.org/10.1109/tcsi.2023.3241859

[15] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718027

[16] Graham Kendall, Ruibin Bai, Jacek Błazewicz, Patrick De Causmaecker, Michel Gendreau, Robert John, Jiawei Li, Barry McCollum, Erwin Pesch, Rong Qu, Nasser Sabar, Greet Vanden Berghe, and Angelina Yee. 2016. Good Laboratory Practice for optimization research. *Journal of the Operational Research Society* 67, 4 (April 2016), 676–689. https://doi.org/10.1057/jors.2015.77

[17] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. 2011. MIPLIB 2010. *Mathematical Programming Computation* 3, 2 (June 2011), 103–163. https://doi.org/10.1007/s12532-011-0025-9

[18] Thorsten Koch, Timo Berthold, Jaap Pedersen, and Charlie Vanaret. 2022. Progress in mathematical programming solvers from 2001 to 2020. *EURO Journal on Computational Optimization* 10 (2022), 100031. https://doi.org/10.1016/j.ejco.2022.100031

[19] Johannes Lenhard and Uwe Küster. 2019. Reproducibility and the Concept of Numerical Solution. *Minds and Machines* 29, 1 (Jan. 2019), 19–36. https://doi.org/10.1007/s11023-019-09492-9

[20] Andrea Lodi and Andrea Tramontani. 2013. Performance Variability in Mixed-Integer Programming. In *Theory Driven by Influential Applications*. INFORMS, 1–12. https://doi.org/10.1287/educ.2013.0112

[21] Robert A. McDougal, Anna S. Bulanova, and William W. Lytton. 2016. Reproducibility in Computational Neuroscience Models and Simulations. *IEEE Transactions on Biomedical Engineering* 63, 10 (Oct. 2016), 2021–2035. https://doi.org/10.1109/tbme.2016.2539602

[22] Edward Raff and Andrew L. Farris. 2023. A Siren Song of Open Source Reproducibility, Examples from Machine Learning. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability (ACM REP '23)*. ACM. https://doi.org/10.1145/3589806.3600042

[23] Samuel Sanford Shapiro and Martin Bradbury Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3-4 (Dec. 1965), 591–611. https://doi.org/10.1093/biomet/52.3-4.591

[24] David John Spiegelhalter. 1980. An omnibus test for normality for small samples. *Biometrika* 67, 2 (1980), 493–496. https://doi.org/10.1093/biomet/67.2.493

Received May 2024