The Journal of Open Source Software

DrWatson: the perfect sidekick for your scientific inquiries

George Datseris[4], **Jonas Isensee**[1], **Sebastian Pech**[5], **and Tamás Gál**[2, 3]

1 Max Planck Institute for Dynamics and Self Organization **2** Erlangen Centre for Astroparticle Physics **3** Friedrich-Alexander-Universität Erlangen-Nürnberg **4** Max Planck Institute for Meteorology **5** Institute for Mechanics of Materials and Structures, TU Wien

DOI: 10.21105/joss.02673

Software

- Review
- Repository
- Archive

Editor: David P. Sanders

Reviewers:

- @jpfairbanks
- @kescobo
- @apdavison

Submitted: 06 July 2020
Published: 29 October 2020

Summary

Doing scientific work always involves a lot of focus and scrutiny, since producing a scientific result requires several levels of depth of analysis, all of which must be as accurate and as reproducible as possible. All this required scrutiny *should* be naturally translated into the codebase of the scientific project. One should strive for a code that is doing what it is supposed to, is reproducible, doesn't break over time, is sufficiently clear of bugs, and with simulation results that are appropriately labelled, and more. The challenges associated with carrying out scientific work should not be made any worse by the difficulties of managing the codebase and resulting data/simulations. An unfortunate but likely outcome of this stress is that scientific codebases tend to be *sloppy*: folders are not organized, there is no version control, data are not provenanced properly, most scripts break over time, and the whole project is very hard, if not impossible, to reproduce. We have created the software DrWatson to make the process of scientific project management easier. In this paper we will describe how DrWatson results in an efficient scientific workflow, taking time away from project management and giving it to doing science.

Statement of need

DrWatson is a **scientific project assistance software**. Its purpose is to help scientists manage their scientific codebase in a simple and clear manner, to make the process of creating the codebase faster and to enable true full reproducibility and project sharing. DrWatson achieves this while being entirely non-invasive throughout the process. It provides a well-tested science-driven framework for managing a scientific project, thus removing the unnecessary stress and giving more time to doing actual science.

Technically DrWatson is a package for the Julia language. It is likely that Julia is the only language that can allow DrWatson to be the powerful framework that will be presented here, because Julia combines multiple dispatch, an integrated package manager, macros and code introspection. That said, however, we believe that a large part of the design of DrWatson can be applied to other languages as well.

In this paper we provide a summary of what DrWatson can do (detailed documentation that is regularly updated is hosted on GitHub). We will then highlight some examples of how using DrWatson speeds up the scientific workflow in real-world scenarios. We close with a comparison with existing software.

Features and Functionality

DrWatson has an opt-in design. This means that DrWatson's features can be grouped into the following few main categories, which remain independent of each other (and thus you "opt-in" which to use).

- **Project setup & navigation**: A universal project structure and functions that allow you to consistently and robustly navigate through your project, no matter where it is located.
- **Naming schemes**: A robust and deterministic scheme for naming and handling your data structures.
- **Saving tools**: Tools for safely saving and loading your data, automatically tagging the Git commit ID to your saved files, and more.
- **Running & listing simulations**: Tools for producing tables of existing simulations/results, adding new simulation results to the tables, preparing batch parameter containers, and more.

The next section illustrates these categories. For a more thorough explanation, the reader is referred to DrWatson's main documentation.

Please note that DrWatson is not a data management system and provides only basic data management functionality that remains self-contained in a single scientific project. One of our main future goals is to integrate DrWatson with a Relational Data Management System, specifically CaosDB (Fitschen et al., 2019), which has been developed specifically to handle large data bases connecting several scientific projects.

Typical workflow with DrWatson

In this section we demonstrate how using DrWatson makes the typical scientific workflow faster, more robust, and easily reproducible. This section is a brief summary of the DrWatson Workflow Tutorial, which in itself showcases a subset of DrWatson's functionality. There the workflow is discussed and demonstrated more thoroughly via explicitly running every code command.

Typically, one starts a scientific project with the function `initialize_project`. This creates a project folder that contains sensible default structure (e.g. folders for data, papers, scripts, etc.), while also making the project a Julia project. This allows the scientific project to be tied with the full hierarchy of exact package versions used, which remains entirely independent from the main Julia installation (or any other project). The project is also a git repository, which allows code versioning and reproducibility, and DrWatson provides functions that make this process seamless (see below).

Within the context of DrWatson, all project-related code runs *after* the corresponding Julia project has been activated. Several DrWatson functions like `projectdir`, `datadir`, `plotsdir` and similar are then made available. When these functions are called they always return the absolute path to the directory (or the appropriate subdirectories) in the active project, independently of the current working directory or the script directory these functions are called from. This establishes a relative-only path relationship within the project, which allows it to naturally run on other machines when shared or synced via e.g. a cloud service. Adding the command `@quickactivate "ProjectName"` to the start of every script automatically activates the appropriate project and thus enables all DrWatson features with minimal effort.

Once the project structure and navigation has been established, there are several functions that help the scientific workflow. For example, the function `dict_list` provides a convenient

and consistent way of defining containers of parameter values. `savename` can be used for preparing a file name or a figure title. Using it would transform the following dictionary

```
parameters = Dict(:phi => 3, :pos_z => 0.5, :date => Date(2020,5,23))
```

into

```
savename(parameters, "jld2")
```

with output:

```
"date=2020-05-23_pos_z=0.5_phi=3.jld2"
```

Once a simulation script is created, taking advantage of e.g. `projectdir, dict_list, savename` among other functions, the user will typically want to save numeric results on disk. DrWatson offers many functions that help the workflow at this level. `safesave` ensures that saved data will never overwrite existing files, but make a new version instead (and back up the original file). `tagsave` allows one to add git-related information to saved data. `tagsave` is a function that excellently highlights how DrWatson is a minimally invasive framework. Continuing from the dictionary `parameters` defined above, we would save any kind of `data` wrapped in a dictionary with the command `save(savename(parameters, "jld2"), data)` (ending `.jld2` is used as an example). By only replacing the function `save` with `@tagsave`, it is possible to attach important information to the saved data

```
@tagsave(savename(parameters, "jld2"), Dict("data" => [1,2,3]));

load(savename(parameters, "jld2")) # load back saved data
```

yielding the output:

```
Dict{Symbol,Any} with 6 entries:
  :gitcommit => "v1.13.0-1-g3a5364f"
  :script    => "docs/build/string#3"
  :data      => [1, 2, 3]
  :gitpatch  => ""
```

The fields `:gitcommit, :script, :gitpatch` were added automatically and provide the necessary information for reproducibility. In case of uncommitted modifications (also called a "dirty git repository"), a patch is saved under the `:gitpatch` key which can be applied to the `:gitcommit` to restore the exact state of the repository. Calling `@tagsave` without extra arguments assumes that you use DrWatson's suggested folder structure using `initialize_project`, and thus that it can find all git-related information automatically. However this is not necessary: you can instead provide a keyword argument `gitpath` to `@tagsave` and explictly specify a git path. Finally, the `parse_savename(filename; kwargs...)` function can be used to obtain the parameters dictionary from the filename.

A last step is data aggregation. The function `collect_results` can traverse the data folder and collect all saved files into a `DataFrame` (the major Julia tabular datastructure) for further analysis. The function is adaptive in that it expands the table as needed when adding new simulations with potentially different parameters.

Sharing and reproducing a DrWatson project is in every respect trivial. The entire project folder is simply sent to a different machine, and in a Julia session the user does the following:

```
using Pkg
Pkg.activate("path/to/project")
Pkg.instantiate()
```

and all necessary dependencies are installed automatically. Since the project uses only relative paths because of the function `projectdir`, every script runs as it did on the original machine. If all saved data are tagged with a git-commit, one can potentially re-create previous results by simply checking out the appropriate commit. Finally, since all package versions used are "baked" into the project, a DrWatson project does not break over time even if the main Julia installation is regularly updated.

One thing that we hope to have highlighted in this section is how DrWatson's functionality is not only minimally invasive, but is also achieved with minimal effort. All functionality is contained within the Julia programming language and within the script files naturally belonging to the project: no command-line usage or special commands are necessary, and neither is the preparation of additional configuration files outside of the scripts. DrWatson's functionality comes directly from using the functions and macros exported by the package. While this is a great advantage in many use cases, it does come with a natural limitation: When running and interacting with code from different languages, the relevant file-IO logic needs to happen in Julia to leverage the full power of DrWatson.

Comparison with existing software

There are numerous tools, software packages and language extensions that provide functions to improve the scientific workflow and allow full reproducibility. They contain features like version control, templates for folder structures, management of external code dependencies and data provenance. Although the tools we investigated perform well in their respective domains and in some parts, such as data provenance, even outperform DrWatson, none of them supports the wide range of functions required for a scientific project, while being non-invasive and easy to use. Therefore, several tools must be combined to provide a similar set of functions as those provided by DrWatson.

One main aspect, and the entry point to every scientific project, is a consistent folder structure. While DrWatson comes with a predefined structure, packages like `rrtools` (Sills, 2004), `prodigenr` (Johnston, n.d.) and `starters` (Locke, n.d.) for R or `Cookiecutter` (Roy, n.d.) (with a template for scientific projects like `Cookiecutter-data-science` (Inc., n.d.)) for Python, allow having a user-defined one. Like DrWatson, most of the tools that initialize a folder structure also initialize a Git repository for version control. In order to gain advantage from having code in version control, e.g. extracting diffs or commit ids, additional software packages, focused on data provenance, are needed.

Applications like `sumatra` (Davison, Mattioni, Samarkanov, & Teleńczuk, 2014), which is written in Python and also supports MATLAB, R, and BASH, while also providing extensibility for other languages, work mainly by executing scripts through a separate standalone tool that captures and tags all files created at runtime. Another example of such an external manager is `ReproZip` (Chirigati, Rampin, Shasha, & Freire, 2016), which traces system calls to identify which files are part of a specific analysis and generates additional metadata to combine everything together into zip-file in a reproducible manner. `ActivePapers` (Hinsen, 2011) falls into the same category and provides a concept and guidelines for reproducible science. There are reference implementations of the ActivePapers concept for Python, JVM (Java Virtual Machine) and Pharo, see (*ActivePapers*, n.d.). Specialised alternatives like `recordr` (Peter Slaughter, n.d.) for R, `explore` for Matlab or `recipy` (Jackson et al., 2018) for Python aim at the non-invasive approach by redefining IO functions for logging metadata during saving.

The outlined tools, however, come with a cost of being limited to certain supported IO functions or the need of additional software to run code or a server infrastructure. Moreover, most of them are tied to a specific programming language and data provenance is only provided in their own context and usually within a single process. Scientific projects, however, often deal with heterogeneous computing environments and pipelines running a multitude of scripts and applications connected to each other, thus the orchestration and data provenance needs to be implemented in a more language-agnostic way. An example for such a framework (with significantly different end-goals compared to DrWatson) is the Common Workflow Language (Amstutz et al., 2016). Notice that in principle DrWatson is tied to Julia, a single programming language. But because Julia has strong interop capabilities, allowing native C/FORTRAN calls and calls to Python or R (for example) via PyCall and RCall, the Julia-based design of DrWatson is much less of a limiting factor than for other languages. In addition, DrWatson is suitable for both making repetitive workflows reproducible (which CWL targets) but also exploratory scientific work.

Therefore, DrWatson only implements basic data provenance features like logging version control information in Julia dictionaries and storing parameter configurations in paths using the `savename` function, which in many cases already covers the basic requirements. The latter approach allows for a simple, universal, file-format-independent method for keeping simulation parameters together with result files.

In terms of portability of scientific projects, management of external code dependencies and packages is crucial. Most of the mentioned languages come with a package manager enabling such functionality. `renv` (Ushey, n.d.) for R implements a feature similar to projects that can be created with `Pkg.jl`, that DrWatson uses. Dependency management is also possible in Python, e.g. by using virtual environments, which are included in the standard library of Python since version 3.5.

Conclusion

In summary, DrWatson combines several functionalities, all communicating excellently with each other and almost all being entirely opt-in, while it goes well beyond only aiding data provenance or simply providing a default folder structure. This results in an efficient scientific workflow, taking time off of project management and giving it to doing science.

References

ActivePapers. (n.d.). Retrieved from http://www.activepapers.org/

Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., et al. (2016). Common Workflow Language, v1.0. doi:10.6084/m9.figshare.3115156.v2

Chirigati, F., Rampin, R., Shasha, D., & Freire, J. (2016). ReproZip: Computational reproducibility with ease. In *Proceedings of the 2016 international conference on management of data*, SIGMOD '16 (pp. 2085–2088). San Francisco, California, USA: ACM. doi:10.1145/2882903.2899401

Davison, A., Mattioni, M., Samarkanov, D., & Teleńczuk, B. (2014). Sumatra: A toolkit for reproducible research. In *Implementing Reproducible Research* (pp. 57–79). doi:10.1201/9781315373461-3

Fitschen, T., Schlemmer, A., Hornung, D., Wörden, H. tom, Parlitz, U., & Luther, S. (2019). CaosDBResearch data management for complex, changing, and automated research workflows. *Data*, *4*(2), 83. doi:10.3390/data4020083

Hinsen, K. (2011). A data and code model for reproducible research and executable papers. *Procedia Computer Science*, *4*, 579–588. doi:10.1016/j.procs.2011.04.061

Inc., D. (n.d.). *Cookiecutter data science*. Retrieved from https://drivendata.github.io/cookiecutter-data-science/#cookiecutter-data-science

Jackson, M., Wilson, R., Zwaan, J. van der, Steinbrook, D. W., Rathgeber, F., Alegre, R., Edwards, T., et al. (2018, October). Recipy. Retrieved from https://github.com/recipy/recipy

Johnston, L. W. (n.d.). *Prodigenr-a component of reproducible and open scientific projects*. Retrieved from https://github.com/lwjohnst86/prodigenr

Locke, S. (n.d.). *Starters*. Retrieved from https://github.com/lockedata/starters

Peter Slaughter, C. J., Matthew B. Jones. (n.d.). *Recordr*. doi:10.5063/F1GF0RF6

Roy, A. (n.d.). *Cookiecutter*. Retrieved from https://cookiecutter.readthedocs.io/en/1.7.2/

Sills, A. V. (2004). RRtools—a maple package for aiding the discovery and proof of finite rogers–ramanujan type identities. *Journal of Symbolic Computation*, *37*(4), 415–448. doi:10.1016/j.jsc.2003.04.002

Ushey, K. (n.d.). *Introduction to renv*. Retrieved from https://rstudio.github.io/renv/articles/renv.html