Jug: Software for parallel reproducible computation in Python

Luis Pedro Coelho

September 19, 2017

Abstract

As computational pipelines become a bigger part of science, it is important to ensure that the results are reproducible, a concern which has come to the fore in recent years. All developed software should be able to be run automatically without any user intervention.

In addition to being valuable to the wider community, which may wish to reproduce or extend a published analysis, reproducible research practices allow for better control over the project by the original authors themselves. For example, keeping a non-executable record of parameters and command line arguments leads to error-prone analysis and opens up the possibility that, when the results are to be written up for publication, the researcher will no longer be able to even completely describe the process that led to them.

For large projects, the use of multiple computational cores (either in a multi-core machine or distributed across a compute cluster) is necessary to obtain results in a useful time frame. Furthermore, it is often the case that, as the project evolves, it becomes necessary to save intermediate results while down-stream analyses are designed (or redesigned) and implemented. Under many frameworks, this causes having a single point of entry for the computation becomes increasingly difficult.

Jug is a software framework which addresses these issues by caching intermediate results and distributing the computational work as tasks across a network.

Jug is written in Python without the use of compiled modules, is completely cross-platform, and available as free software under the liberal MIT license.

Jug is available from `http://github.com/luispedro/jug`.

Keywords

Parallel programming; Python; Memoization; Reproducible computation; High performance computing; Data analysis; Computational science

1 Introduction

The value of reproducible research in computational fields has been recognized in several areas, including fields as different computational mathematics, signal processing

[18, 59], neuronal network modeling [45], archeology [41], or climate science [20]. This has lead researchers to realize that computational reproducibility is an issue that spans across fields [19, 21, 23, 38].
Besides the benefits to the wider scientific community and society, reproducible practices can be advantageous to the individual researcher as the resulting research process is faster and less error prone [40].
Several implementations of reproducible papers (or executable papers) have been proposed towards the goal of reproducing published analyses [3, 36, 48, 64]. These solutions do not necessarily scale to large problems, those that take several days, months, or years of CPU time. For very large problems, specialized solutions are needed to fully leverage high performance computing platforms [15, 65]. Nonetheless, there is a range of medium-sized problems that can be successfully tackled on a computer cluster with a small number of nodes or even taking advantage of a single multicore machine. It is for these medium-sized problems that Jug is best suited.
A typical ad hoc approach to this problem is to save intermediate files on disk. A limitation of this approach is that often the design of the computation itself takes several iterations as intermediate steps are improved. Thus, some intermediate results need to be recomputed. This involves a large amount of human management of the state of the computation, breaking it up into pieces, and, when using a cluster, scheduling jobs on the batch computing system.
Jug is a task-based framework, which supports saving and sharing of intermediate results and parallelisation on computer clusters (or multi-core machines).
Intermediate results are cached with a key which takes into consideration all input parameters of that computation. Thus, any change in the parameters immediately triggers a recomputation of all dependent results. The basic model is similar to Make, which has been used before for implementing reproducible research pipelines [53].
However, unlike Make, Jug is written in Python, a general purpose programming language which is widely used in scientific programming [49]. A Task in Jug can consist of any Python function and its arguments. The task can include running external commands and calling routines written in other languages as Python has many tools to interface with the rest of the system [4, 5, 6].
A Jugfile (the file which specifies which tasks need to be run—named by analogy to the Makefiles of Make) is a simple Python script with some added notations. Below, we show how the only a few small changes are needed to transform a conventional Python script into a Jugfile.

2 Implementation and architecture

2.1 Task-based architecture

Jug is designed around tasks. A task is defined as a Python function and a set of arguments, which may be Python values or the output of other tasks.
For example, consider the following toy problem: given a set of text files, count the number of lines in each, and report the average number of lines. Conceptually, we can already see that each input file can be processed independenly (to count the number of lines) with the results combined at the end.
This can be implemented with Jug, using the following code:

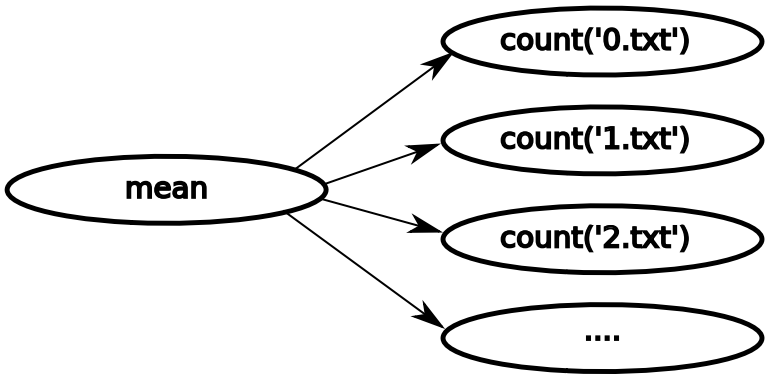

Figure 1: Simple Dependency Structure for Example in the Text. This assumes that the directory had a collection of files named 0.txt, 1.txt,...

```
from jug import Task

def linecount(fname):
    'Return the number of lines in a file'
    n = 0
    for line in open(fname):
        n += 1
    return n

def mean(args):
    return sum(args)/float(len(args))

inputs = glob('*.txt')
counts = []
for tf in inputs:
    counts.append( Task(linecount, tf) )
final = Task(mean, counts)
```

This code defines the task dependency structure represented in Figure 1. As we can see from the graph, all of the `linecount` operations can potentially be run in parallel, while the `mean` operation must wait the result of all of the other computation. The dependency structure is always a DAG (directed acyclic graph). We will later see how Jug exploits this structure to achieve parallelism.
The code above has the construct `Task(f, args)` repeated several times. Using the `TaskGenerator` decorator this can be simplified to a more natural syntax.

```
from jug import TaskGenerator

@TaskGenerator
def linecount(fname):
    'Return the number of lines in a file'
    n = 0
    for line in open(fname):
        n += 1
```

```
    return n

@TaskGenerator
def mean(args):
    return sum(args)/float(len(args))

inputs = glob('*.txt')
counts = []
for tf in inputs:
    counts.append( linecount(tf) )
final = mean(counts)
```

As the reader can appreciate, this is identical to a traditional Python script, except for the `@TaskGenerator` decorators. With a few limitations (which unfortunately can give rise to complex error messages), scripts can be written as if they were sequential Python scripts.
By default, Jug looks for a file called `jugfile.py`, but any filename can be used. Generically, we refer to the script being run as the Jugfile.

2.2 Jug subcommands

Based on a Jugfile defining the computational structure, Jug is invoked by calling the `jug` executable with a subcommand. The most important subcommands are *execute*, *status*, and *shell*.
Execution is used for actually running the tasks. It performs a more complex version of the following pseudo-code:

```
tasks = alltasks in topological sort order
while len(tasks) > 0:
    next = task.pop()
    if not next.has_run() and not next.is_running():
        while not next.can_run():
            sleep(waiting_period)
        with locked(next):
            if not next.has_run():
                next.run()
```

If run on a single processor, this will just run all of the tasks in order. It is most interesting when it is run on multiple processors. Because of the lock synchronisation, tasks can be run in parallel to take advantage of the multiple cores.
The actual code is more complex than what is shown above, particularly to ensure that the locking is performed correctly and that the waiting step eventually times out (in order to handle the situation where another process is hung).
The `status` subcommand prints out a summary of the status of all of the tasks. Figure 2 shows the output of this command using the example Jugfile above. We assume that the Jugfile was called `jugfile.py` on disk and that there were 20 textfiles in the directory. We can see that there are 20 tasks ready to run, while the `mean` task is still waiting for the results of the other tasks.

```
$jug status jugfile.py
Task name         Waiting      Ready   Finished    Running
-----------------------------------------------------------
jugfile.mean            1          0          0          0
jugfile.linecount       0         20          0          0
...........................................................
Total:                  1         20          0          0
```

Figure 2: Output of `jug status`. The $ sign shown is the command line prompt, and the status subcommand was run. At this point, nothing has been run. The output has been edited for space reasons (spacing columns were removed).

2.3 Backends

A basic feature of Jug is its ability to save and load results so that values computed by one process can be loaded by another. Each task of the form `Task(f, args)` is represented by a hash of `f` and `args`. Jug assumes that the result of a function is uniquely defined by its arguments. Therefore, Jug does not work well with functions which are not pure or which access (non-constant) global variables.
A Jug backend must then support four basic operations:

save Saving a Python object by its hash name.

load Loading a Python object by its hash name.

lock Creating a lock by hash name. Naturally, this lock must be created atomically.

release Releasing the lock.

A few other operations, such as deletion and listing of names are also supported.
The filesystem can support all of the above operations if the backend is coded correctly to avoid race conditions. This is the default backend, identified simply by a directory name. Inside this directory, files named by a hexadecimal representation of their hashes. Objects are saved using Python's `pickle` module with zlib compression. As a special case, `numpy` arrays [62] are saved to disk directly. This special case was introduced as `numpy` arrays are a very common data type in scientific programming and saving them directly allows for very fast saving and loading (they are represented on disk as a header followed by the binary information they contain).
Another backend currently included with Jug is a redis backend. Redis a name-key database system.[1] Redis is particularly recommended for the case where there are many small objects being saved. In this case, keeping each as a separate file on disk would incur a large space penalty, while redis keeps them all in the same file.
Finally, there is an in-memory backend. This was initially developed for testing, but can be useful on its own.

[1] See the redis webpage, at redis.io, for detailed information about redis.

```
$jug shell images.py

Available jug functions:
    - value() : loads a specific object
    - load_all() : loads all objects


In [1]: value(final)
Out[1]: 7.5

In [2]: value(counts)
Out[2]: [7, 7, 7, 7, 7, ... 8, 8, 8, 8, 8, 8, 8, 8, 8]
```

Figure 3: Interaction with `jug shell`. The `value` function loads and returns the result from any task. The final line has been edited (marked with ...) for presentation purposes.

2.4 Asynchronous function

Parallelisation is achieved by running more than one Jug process simultaneously. All of the synchronisation is outsourced to the backend. As long as all of the processes can access the backend, there is no need for them to communicate directly. It can even be the case that processors start working on tasks in mid-processing. This makes Jug usable in batch-based computer cluster environments, which are quite common in research institutions.

2.5 Software development with Jug

The output of the computation can be obtained from Jug in several ways. One can write a task that writes the output to a file in the desired format. Alternatively, outputs can be inspected interactively using the *shell* subcommand. It is expected that the first option will be used for the final version of the computation, whilst the second one is most helpful during development.

The shell subcommand, invokes an IPython shell with all the objects in the Jugfile loaded. The IPython console is an enhanced interactive shell for Python [48]. A few functions are added to the namespace, in particular, `value` will load the results of a task object if it is available.

Figure 3 shows a possible interaction session with the `jug shell` subcommand. While having to explicitly load all the results may be bothersome, it is both much faster at start up and the user might not load more than a few objects throughout their session. In some cases, loading all of the objects simultaneously might even be impossible due to memory constraints. Furthermore, this allows exploration of the task structure for debugging.

Jug can also be loaded as a library from a Python script and Jug computation outputs can serve as inputs for further computation. This can be performed from inside a Jupyter notebook [33], for example, for interactive exploration of the computational results.

2.6 Result invalidation

If a researcher improves an intermediate step in a pipeline (e.g., fixes a bug) and wishes to obtain new results of the computation, then all outputs from that step and downstream must be recomputed, but results from upstream and unrelated processes can be reused. Formally, in the task DAG, affected tasks and their descendants must be recomputed. Without tool support, this can be a very error-prone operation: by not removing all relevant intermediate files, it is easy to generate an irreproducible state where different blocks of the computation output were generated using different versions of the code.
Therefore, Jug adds support for result invalidation. When a results from a task are invalidated, all tasks which (directly or indirectly) depend on them are also invalidated. In the case where the parameters of a task have not changed, only the code implementing it, it is still necessary to manually invalidate tasks. This can be performed using the `jug invalidate` subcommand which will invalidate all tasks with the given name as well as other which directly or indirectly depend on them. For finer control, within the `jug shell` environment, individual tasks can be invalidated with the `invalidate` function.
An alternative, would be to take code which implements the task into account while computing the task hash. This would mark any results computed with this code as outdated if the code changed. While this would add another layer of protection, it would still be possible to make mistakes. If the function depended on other functions, especially if this was done in a dynamic way, it could be hard to discover all dependencies. Additionally, even minute refactoring of the code would lead to over eager recomputation. This could make the developer wary of making improvements in their code, resulting in overall worse code.
Therefore, as a design choice, Jug asks the user to explicitly invalidate their results, while supporting automatic dependency discovery. The recommendation is still that the user run the full pipeline from start to finish once they are satisfied with the state of the code and before publication, but the pipeline development stage can be more agile.

3 Example

This section presents an edited version of the code used in a previously published study of computer vision techniques for bioimage analysis [10].[2] The code was edited to remove superfluous details, however the overall logic is preserved as the original version was already based on Jug. As part of that paper, it was necessary to evaluate the classification accuracy of a machine learning model.
To evaluate classification, the dataset is broken up into 10 pieces and each of the pieces is held-out of the analysis and then used to evaluate analysis (the final result is the average of all ten values). This is known as cross-validation.
The image processing is done with mahotas [9], while the machine learning aspects are handled with sckit-learn [46].
The example starts with a simple Python function to parse the data directory structure and return a list of input files:

```
from jug import TaskGenerator, CachedFunction
```

[2]The original code to reproduce the full study is available online at `https://github.com/luispedro/Coelho2013_Bioinformatics`.

```
import mahotas as mh
from sklearn.cross_validation import KFold

def load():
    '''
    This function assumes that the images are stored in
    data/ with filenames matching the pattern
    label-protein-([0-9]).tiff
    '''
    from os import listdir
    images = []
    base = './data/'
    for path in listdir(base):
        if not 'protein' in path: continue
        label = path.split('-')[0]
        # We only store paths and will load data on demand
        # This saves memory.
        im = (base + path, label)
        images.append(im)
    return images
```

Now, we define as functions the core of the study. For space reasons, the code presented here computes only a simple set of features, namely Haralick features [29].

```
# TaskGenerator can be used with any function, including builtins:
jug_sum = TaskGenerator(sum)

@TaskGenerator
def features(im):
    im = mh.imread(im)
    return mh.features.haralick(im)

@TaskGenerator
def fold(features, labels, train, test):
    from sklearn import linear_model
    from sklearn import metrics
    clf = linear_model.LogisticRegression()
    clf.fit(features[train], labels[train])
    preds = clf.predict(features[test])
    return metrics.confusion_matrix(labels[test], preds)

@TaskGenerator
def output(cmatrix, oname):
    with open(oname, 'w') as output:
        output.write(cmatrix)
        output.write('\n')
```

In this case, we defined a function, `output` which will write the final results to a file. Finally, we call the above generators to process all the data. The resulting code is very readable for any programmer.

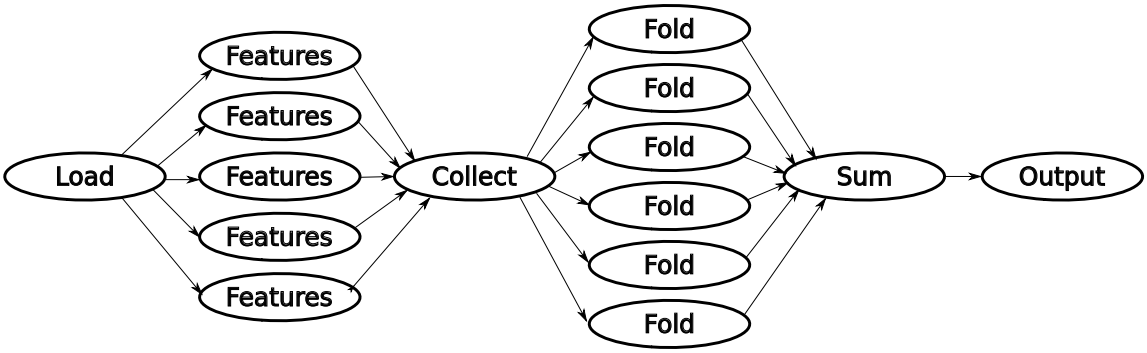

Figure 4: Dependency Structure for second example in the Text. The fan-out/fan-in structure is typical (it is an instance of the map-reduce framework).

```
# CachedFunction ensures that the load() function is run just once for
# efficiency:
images = CachedFunction(load)

allfeatures = []
labels = []
for im,label in images:
    labels.append(label)
    allfeatures.append(features(im))
partial_cmats = []
for train,test in KFold(len(features), n_folds=10):
    partial_cmats.append(
        fold(features, labels, train, test))
cmatrix = jug_sum(partial_cmats)
output(cmatrix, 'cmatrix.txt')
```

Figure 4 shows the dependency structure of this examples. The main feature is the fan-out/fan-in which is associated with map-reduce frameworks. A more compact representation can be generated by Jug itself, using the `jug graph` subcommand. Figure 5 shows an example of such auto-generated graphs.

4 Quality control

Jug follows best practices in the field [60, 63] and includes a full test suite (>100 tests) with continuous integration using the Travis service.

The user can run the test suite using the `test-jug` subcommand.

Jug is available under the MIT software license, which grants the user right to use, copy, and modify the code almost at will. It is developed using the git version control tool, with the project being hosted on the github platform. The Jug project is available at `https://www.github.com/luispedro/jug` and bug reports can be submitted using the github issues system at `https://github.com/luispedro/jug/issues`. An open mailing-list (`https://groups.google.com/forum/#!forum/jug-users`) provides discussion and support.

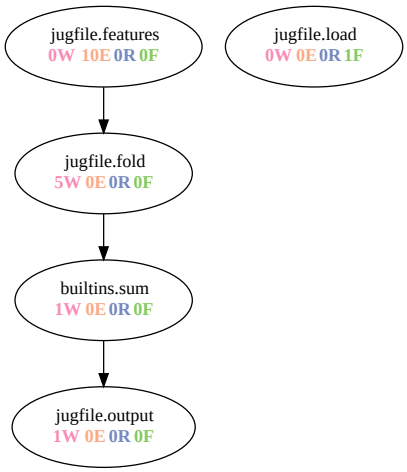

Figure 5: Dependency Structure for second example in the Text generated by the `jug graph` command. The numbers encode the number of tasks waiting (W), ready to be run (E), running (R), or finished (F).

Dependencies

Basic usage of Jug requires no dependencies beyond Python itself. Some specific subcommands or functionality, however, have additional requirements: the `jug shell` subcommand relies on IPython, `jug webstatus` on the bottle package; and writing out task completion metadata in Yaml format requires the pyyaml library.

List of contributors

Luis Pedro Coelho designed and implemented Jug.
The following individuals have contributed patches (alphabetically): Renato Alves, Alex Ford, Andreas Sorge, Breanndán Ó Nualláin, Christian Jauvin, Dominique Geffroy, Elliott Sales de Andrade, Hamilton Turner, and Ulrich Dobramysl; as well as the github users identified as abingham, cgcgcg, and dsign.

Software location:

Archive

Name: Zenodo
Persistent identifier: https://doi.org/10.5281/zenodo.847794
Licence: MIT
Publisher: Luis Pedro Coelho
Version published: 1.6.0
Date published: 24 Aug 2017

Code repository

Name: Github
Persistent identifier: https://github.com/luispedro/jug
Licence: MIT

Language

Jug is written in Python, with support for Python 2 (versions 2.6 and 2.7) and 3 (versions 3.3 and above). Jug is automatically tested on all these versions.

5 Reuse potential

Jug is a very generic framework for any computational pipeline. It has been used by the author in several projects [10, 11, 12, 57]. Others have used the framework in other contexts, such as physics [2], machine learning [30, 32, 52], metereology [60, 61], and it is used in the pyfssa package for algorithmic finite-size scaling analysis [55].

6 Discussion

6.1 Similar tools

Several pipeline tools have been used in scientific computing (for a recent review, see the work of Leipzig [35]).

The Make build system has a difficult syntax for any use beyond the most basic, but it is conceptually simple and widely available. Thus, it has been used as the basis of a set of conventions for reproducible research by Schwab et al. [53]. Fomel and Hennenfent [22] proposed a system built on top of Scons which shared superficially similar design. Scons, like Make, is a build system. It supports spawning parallel jobs using multiple threads. In the original use case, the tasks are delegated to other commands and the operating system can take care of parallelism. If, however, the tasks are computationally intensive in Python, contention for the global interpreter lock will limit the amount of real parallelism.[3] As Make syntax does not directly support complex operations, some researchers have developed alternative domain-specific syntaxes for for specifying a workflow graph [8, 44].

Ruffus is a Python-based solution which supports parallel execution of pipelines [25]. Using the Ruby programming language, Mishima et al. [42] proposed Pwrake, which supports parallel execution of bioinformatics workflows written (or accessible) in that language. Snakemake [34] also improves over Make by providing more complex rules and automatic interaction with a high performance compute cluster, while providing a domain specific language which can easily be extended with Python. Similarly, Sadedin et al. [51] proposed Bpipe, a tool based on a domain-specific language around shell scripting. One large difference of Jug compared to these tools is that Jug is tightly integrated with Python code. Thus, while it lacks support for directly spawning external processes, it makes it easier to call Python-based functions using a natural syntax (calling external processes is naturally still possible through standard Python code).

For large scale computation, there are several workflow engines which have been used in science [56, 58], such as Taverna [31], eHive [54], or Kepler [1, 39]. In general, these are generic frameworks which allow the user to specify a computational path, although domain specific solutions also exist, such as Galaxy [24] for bioinformatics.

Also relevant is the IncPy implementation of a Python interpreter with automatic persistent memoization [26, 27] (unfortunately, no longer maintained). The advantages of

[3] In the most commonly used Python interpreters, there is a lock that prevents more than one thread from simultaneously executing interpreted code, although they can execute non-interpreted code, as calling an external programme or certain external libraries.

that system apply to Jug as well, with a few differences. Their system implements automatic memoization, while, using Jug, the user needs to manually annotate functions to memoize using `TaskGenerator`. This extra control can be necessary to avoid a proliferation of intermediate results when these are very large (often a very large intermediate output is not necessary, and only a summary must be kept) at the cost of increased overhead for the programmer. Additionally, Jug supports running tasks in parallel, a functionality that is absent from IncPy.

Joblib (bundled with scikit-learn [46], but usable independently of it) and memo [43] use mechanisms similar to Jug for in-process and in-memory memoization and parallelization. Joblib additionally supports memoization to disk, which like Jug enables the reuse of partial results in computations. Many of the design choices are similar to Jug, but usage is different. While Joblib is designed to speed an analysis that is run using a traditional Python driver script, with Jug the user defines a computational graph in Python, but this graph is executed by the Jug machinery. This enables functionality such as `jug status` and `jug graph` making the status of the computation explicit.

Dask [13] provides a generic task execution framework, similar to Jug in addition to a distributed numeric library (which achieves high performance on a predefined limited set of operations). Like Jug, dask can coordinate computation on a dependency graph across compute nodes on a cluster. Joblib (mentioned above) can also use Dask as a computational backend. Dask uses a central scheduler dispatching jobs to workers, unlike Jug where each node is independently running the Jugfile script with only limited communication between the nodes through the result storage backend. This enables dynamic control of the workload as the scheduler can assign workers to tasks so as to minimize the expected I/O burden. However, this architecture requires extra setup on the part of the user to ensure that communication between worker nodes and the central scheduler is set up before computation can proceed (while Jug was designed to work well in a shared cluster where the number of available compute nodes may vary throughout the process). Jug also supports saving intermediate results between different runs of the process so that intermediate results are available even if the code for subsequent computations changes. This functionality is not present in Dask.

Peng and Eckel [47] describe cacher, an R framework which uses similar concepts to Jug to allow for results to be distributed. In principle, a very similar system could be built on top of Jug by sharing the backend between users (either in the same research unit or after publication). For simple reproducibility, it would be sufficient for the researcher to share their database upon publication.

To be able ensure complete reproducibility of a computational result, it is necessary to capture all dependencies in the environment. Sumatra [14] tracks execution of a process and captures all dependencies. Reprozip [50] uses a similar approach to build a single archive with all dependencies which can be used by other users to exactly reproduce the original computational environment. These can be used in combination with Jug to achieve perfect reproducibility.

6.2 Conclusions

Jug focuses on the development of the computation as much as its communication. In fact, when it comes to communication other tools might be better suited as they combine

written exposition with computer code. They can be used in combination with Jug as they do not often provide caching and distribution, needed for large projects. While the simplest Jug use employs a single single Jugfile, a Jugfile can import the results of another. For example, a first Jugfile might perform all heavy-duty computation and be run on a computing cluster. A second script could be embedded inside an executable paper to generate tables and plots [16]. This faster script could be run as part of building the paper.

Jug improves the pipeline development experience and makes the programming researcher more productive and less error-prone. As many scientists now spend a significant fraction of their time performing computational work [28, 49], increasing their productivity in this task can have significant effects. Jug is based on Python, a programming language which is widely used for scientific computation. Thus, it can be used in a known environment without a high learning curve.

Jug does not address the issue of how to build a reproducible analysis environment. However, it can be used in combination with other tools which ensure a reproducible environment, such as container based tools [7, 37] or package managers which emphacise reproducibility such as Nix [17] (Jug is available as a nix package in the main nixpkgs repository).

Jug was also not designed to compete with large scale frameworks such as Hadoop or Spark, which scale better to very large projects. However, those frameworks have much higher costs in terms of development time (code must be written especially for the framework) and overhead. They also require that the user learn a new framework. For scientists who want to quickly adapt pre-existing code to run on a cluster or even a multi-core machine, Jug provides a better trade-off than those higher-powered alternatives in a familiar programming language.

Acknowledgments

The author thanks Sven Augustin for helpful comments on a previous version of this manuscript.

References

[1] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. “Kepler: an extensible system for design and execution of scientific workflows”. In: *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on* (Apr. 2004). URL: https://scholar.google.com/scholar?cluster=17284613261601846997 (cit. on p.).

[2] Sven Augustin and Carsten Müller. “Interference effects in Bethe-Heitler pair creation in a bichromatic laser field”. In: *Physical Review A* 88.2 (2013), p. 022109. ISSN: 2469-9934. DOI: 10.1103/physreva.88.022109. URL: http://dx.doi.org/10.1103/physreva.88.022109 (cit. on p.).

[3] Ben Baumer, Mine Cetinkaya-Rundel, Andrew Bray, Linda Loi, and Nicholas J Horton. “R Markdown: Integrating A Reproducible Analysis Tool into Introductory Statistics’”. In: *Technological Innovations in Statistics Education* 8 (2014) (cit. on p.).

[4] David M Beazley. "Automated scientific software scripting with SWIG". In: *Future Generation Computer Systems* 19 (Mar. 2003). URL: https://scholar.google.com/scholar?cluster=14166776132178739884 (cit. on p.).

[5] David M Beazley. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++." In: *Tcl/Tk Workshop* (1996). URL: https://scholar.google.com/scholar?cluster=2768773569829356266 (cit. on p.).

[6] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2 (Nov. 2011), pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/mcse.2010.118. URL: http://dx.doi.org/10.1109/mcse.2010.118 (cit. on p.).

[7] Carl Boettiger. "An introduction to Docker for reproducible research". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79. ISSN: 0163-5980. DOI: 10.1145/2723872.2723882. URL: http://dx.doi.org/10.1145/2723872.2723882 (cit. on p.).

[8] Pablo Cingolani, Rob Sladek, and Mathieu Blanchette. "BigDataScript: a scripting language for data pipelines". In: *Bioinformatics* 31.1 (2015), pp. 10–16. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btu595. URL: http://dx.doi.org/10.1093/bioinformatics/btu595 (cit. on p.).

[9] Luis Pedro Coelho. "Mahotas: Open source software for scriptable computer vision". In: *Journal of Open Research Software* 1.1 (2013), e3. ISSN: 2049-9647. DOI: 10.5334/jors.ac. URL: http://dx.doi.org/10.5334/jors.ac (cit. on p.).

[10] Luis Pedro Coelho, Joshua D Kangas, Armaghan W Naik, Elvira Osuna-Highley, Estelle Glory-Afshar, Margaret Fuhrman, Ramanuja Simha, Peter B Berget, Jonathan W Jarvik, and Robert F Murphy. "Determining the subcellular location of new proteins from microscope images using local features." In: *Bioinformatics (Oxford, England)* 29.18 (2013), pp. 2343–9. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btt392. URL: http://dx.doi.org/10.1093/bioinformatics/btt392 (cit. on p.).

[11] Luis Pedro Coelho, Catarina Pato, Ana Friães, Ariane Neumann, Maren von Köckritz-Blickwede, Mário Ramirez, and João André Carriço. "Automatic determination of NET (neutrophil extracellular traps) coverage in fluorescent microscopy images." In: *Bioinformatics (Oxford, England)* 31.14 (2015), pp. 2364–70. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btv156. URL: http://dx.doi.org/10.1093/bioinformatics/btv156 (cit. on p.).

[12] Luis Pedro Coelho, Tao Peng, and Robert F. Murphy. "Quantifying the distribution of probes between subcellular locations using unsupervised pattern unmixing". In: *Bioinformatics* 26.12 (Oct. 2010), pp. i7–i12. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btq220. URL: http://dx.doi.org/10.1093/bioinformatics/btq220 (cit. on p.).

[13] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: http://dask.pydata.org (cit. on p.).

[14] Andrew Davison. "Automated Capture of Experiment Context for Easier Reproducibility in Computational Research". In: *Computing in Science & Engineering* 14.4 (Dec. 2012), pp. 48–56. ISSN: 1521-9615. DOI: 10.1109/mcse.2012.41. URL: http://dx.doi.org/10.1109/mcse.2012.41 (cit. on p.).

[15] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (Aug. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492 (cit. on p.).

[16] Matthieu Delescluse, Romain Franconville, Sébastien Joucla, Tiffany Lieury, and Christophe Pouzat. "Making neurophysiological data analysis reproducible: Why and how?" In: *Journal of Physiology-Paris* (Nov. 2011). ISSN: 0928-4257. DOI: 10.1016/j.jphysparis.2011.09.011. URL: http://www.sciencedirect.com/science/article/pii/S0928425711000374 (cit. on p.).

[17] Adrien Devresse, Fabien Delalondre, and Felix Schürmann. "Nix Based Fully Automated Workflows and Ecosystem to Guarantee Scientific Result Reproducibility Across Software Environments and Systems". In: *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. SE-HPCCSE '15. Austin, Texas: ACM, 2015, pp. 25–31. ISBN: 978-1-4503-4012-0. DOI: 10.1145/2830168.2830172. URL: http://doi.acm.org/10.1145/2830168.2830172 (cit. on p.).

[18] David L Donoho, Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. "Reproducible Research in Computational Harmonic Analysis". In: *Computing in Science & Engineering* 11.1 (Sept. 2009), pp. 8–18. ISSN: 1521-9615. DOI: 10.1109/mcse.2009.15. URL: http://dx.doi.org/10.1109/mcse.2009.15 (cit. on p.).

[19] Joel T Dudley and Atul J Butte. "Reproducible in silico research in the era of cloud computing". In: *Nature biotechnology* 28 (Oct. 2010). URL: https://scholar.google.com/scholar?cluster=14329535853377349322 (cit. on p.).

[20] Georg Feulner. *Reproducibility: Principles, Problems, Practices, and Prospects: Principles, Problems, Practices, and Prospects*. 2016, pp. 269–285. DOI: 10.1002/9781118865064.ch12. URL: http://dx.doi.org/10.1002/9781118865064.ch12 (cit. on p.).

[21] Sergey Fomel. "Reproducible Research as a Community Effort: Lessons from the Madagascar Project". In: *Computing in Science & Engineering* 17.1 (2015), pp. 20–26. ISSN: 1521-9615. DOI: 10.1109/mcse.2014.94. URL: http://dx.doi.org/10.1109/mcse.2014.94 (cit. on p.).

[22] Sergey Fomel and Gilles Hennenfent. "Reproducible Computational Experiments using Scons". In: (July 2007). DOI: 10.1109/icassp.2007.367305. URL: http://dx.doi.org/10.1109/icassp.2007.367305 (cit. on p.).

[23] Carole Goble. "Better Software, Better Research". In: *IEEE Internet Computing* 18.5 (2014), pp. 4–8. ISSN: 1089-7801. DOI: 10.1109/mic.2014.88. URL: http://dx.doi.org/10.1109/mic.2014.88 (cit. on p.).

[24] Jeremy Goecks, Anton Nekrutenko, James Taylor, and The Galaxy Team. "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences". In: *Genome Biology* 11.8 (Oct. 2010), pp. 1–13. ISSN: 1474-760X. DOI: 10.1186/gb-2010-11-8-r86. URL: http://dx.doi.org/10.1186/gb-2010-11-8-r86 (cit. on p.).

[25] Leo Goodstadt. "Ruffus: A Lightweight Python Library for Computational Pipelines". In: *Bioinformatics* (Oct. 2010). DOI: 10.1093/bioinformatics/btq524. URL: http://bioinformatics.oxfordjournals.org/content/early/2010/09/16/bioinformatics.btq524.abstract (cit. on p.).

[26] Philip J. Guo and Dawson Engler. "Towards Practical Incremental Recomputation for Scientists: An Implementation for the Python Language". In: *Proceedings of the 2Nd Conference on Theory and Practice of Provenance*. TAPP'10. San Jose, California: USENIX Association, 2010, pp. 6–6. URL: http://dl.acm.org/citation.cfm?id=1855795.1855801 (cit. on p.).

[27] Philip J. Guo and Dawson Engler. *Using automatic persistent memoization to facilitate data analysis scripting*. Nov. 2011, pp. 287–297. DOI: 10.1145/2001420.2001455. URL: http://doi.acm.org/10.1145/2001420.2001455 (cit. on p.).

[28] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. "How do scientists develop and use scientific software?" In: (Sept. 2009), pp. 1–8. DOI: 10.1109/secse.2009.5069155. URL: http://dx.doi.org/10.1109/secse.2009.5069155 (cit. on p.).

[29] Robert M Haralick, K Shanmugam, and Its'Hak Dinstein. "Textural Features for Image Classification". In: *IEEE Transactions on Systems, Man, and Cybernetics* 3.6 (1973), pp. 610–621. ISSN: 0018-9472. DOI: 10.1109/tsmc.1973.4309314. URL: http://dx.doi.org/10.1109/tsmc.1973.4309314 (cit. on p.).

[30] James Hensman, Alexander Matthews, Maurizio Filippone, and Zoubin Ghahramani. "MCMC for Variationally Sparse Gaussian Processes". In: (2015) (cit. on p.).

[31] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. "Taverna: a tool for building and running workflows of services". In: *Nucleic Acids Research* 34.suppl 2 (June 2006), W729–W732. ISSN: 0305-1048. DOI: 10.1093/nar/gkl320. URL: http://dx.doi.org/10.1093/nar/gkl320 (cit. on p.).

[32] Andrew Ilyas. "MicroFilters: Harnessing twitter for disaster management". In: (2014), pp. 417–424. DOI: 10.1109/ghtc.2014.6970316. URL: http://dx.doi.org/10.1109/ghtc.2014.6970316 (cit. on p.).

[33] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*. 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87. URL: https://doi.org/10.3233/978-1-61499-649-1-87 (cit. on p.).

[34] Johannes Köster and Sven Rahmann. "Snakemake–a scalable bioinformatics workflow engine." In: *Bioinformatics (Oxford, England)* 28.19 (Dec. 2012), pp. 2520–2. URL: http://www.ncbi.nlm.nih.gov/pubmed/22908215 (cit. on p.).

[35] Jeremy Leipzig. "A review of bioinformatic pipeline frameworks". In: *Briefings in Bioinformatics* (2016), bbw020. ISSN: 1467-5463. DOI: 10.1093/bib/bbw020. URL: http://dx.doi.org/10.1093/bib/bbw020 (cit. on p.).

[36] Friedrich Leisch. *Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis*. Feb. 2002, pp. 575–580. DOI: 10.1007/978-3-642-57489-4_89. URL: http://dx.doi.org/10.1007/978-3-642-57489-4_89 (cit. on p.).

[37] Felipe da Veiga Leprevost, Björn A. Grüning, Saulo Alves Aflitos, Hannes L. Röst, Julian Uszkoreit, Harald Barsnes, Marc Vaudel, Pablo Moreno, Laurent Gatto, Jonas Weber, Mingze Bai, Rafael C Jimenez, Timo Sachsenberg, Julianus Pfeuffer, Roberto Vera Alvarez, Johannes Griss, Alexey I. Nesvizhskii, and Yasset Perez-Riverol. "BioContainers: An open-source and community-driven framework for software standardization". In: *Bioinformatics* 33.16 (), btx192–. ISSN: 1460-2059. DOI: 10.1093/bioinformatics/btx192. URL: http://dx.doi.org/10.1093/bioinformatics/btx192 (cit. on p.).

[38] Randall J LeVeque, Ian M Mitchell, and Victoria Stodden. "Reproducible research for scientific computing: Tools and strategies for changing the culture". In: *Computing in Science & Engineering* 14.4 (Dec. 2012), pp. 13–17. ISSN: 1521-9615. DOI: 10.1109/mcse.2012.38. URL: http://dx.doi.org/10.1109/mcse.2012.38 (cit. on p.).

[39] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. "Scientific workflow management and the Kepler system". In: *Concurrency and Computation: Practice and Experience* 18.10 (June 2006), pp. 1039–1065. ISSN: 1532-0634. DOI: 10.1002/cpe.994. URL: http://dx.doi.org/10.1002/cpe.994 (cit. on p.).

[40] Florian Markowetz. "Five selfish reasons to work reproducibly". In: *Genome Biology* 16.1 (2015), p. 274. ISSN: 1465-6906. DOI: 10.1186/s13059-015-0850-7. URL: http://dx.doi.org/10.1186/s13059-015-0850-7 (cit. on p.).

[41] Ben Marwick. "Computational Reproducibility in Archaeological Research: Basic Principles and a Case Study of Their Implementation". In: *Journal of Archaeological Method and Theory* (2016), pp. 1–27. ISSN: 1072-5369. DOI: 10.1007/s10816-015-9272-9. URL: http://dx.doi.org/10.1007/s10816-015-9272-9 (cit. on p.).

[42] Hiroyuki Mishima, Kensaku Sasaki, Masahiro Tanaka, Osamu Tatebe, and Kohichiro Yoshiura. "Agile parallel bioinformatics workflow management using Pwrake". In: *BMC Research Notes* 4.1 (Nov. 2011), p. 331. ISSN: 1756-0500. DOI: 10.1186/1756-0500-4-331. URL: http://www.biomedcentral.com/1756-0500/4/331 (cit. on p.).

[43] Alexander Moreno and Tucker Balch. "Improving financial computation speed with full and subproblem memoization". In: *Concurrency and Computation: Practice and Experience* 28.3 (2016), pp. 905–915. ISSN: 1532-0634. DOI: 10.1002/cpe.3693. URL: http://dx.doi.org/10.1002/cpe.3693 (cit. on p.).

[44] Francesco Napolitano, Renato Mariani-Costantini, and Roberto Tagliaferri. "Bioinformatic pipelines in Python with Leaf". In: *BMC Bioinformatics* 14.1 (2013), pp. 1–14. ISSN: 1471-2105. DOI: 10.1186/1471-2105-14-201. URL: http://dx.doi.org/10.1186/1471-2105-14-201 (cit. on p.).

[45] Eilen Nordlie, Marc-Oliver Gewaltig, and Hans Ekkehard Plesser. "Towards Reproducible Descriptions of Neuronal Network Models". In: *PLoS Comput Biol* 5.8 (Sept. 2009), e1000456. DOI: 10.1371/journal.pcbi.1000456. URL: http://dx.doi.org/10.1371%5C%2Fjournal.pcbi.1000456 (cit. on p.).

[46] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. "Scikit-learn: Machine Learning in Python". In: (Dec. 2012) (cit. on p.).

[47] Roger D Peng and Sandrah P Eckel. "Distributed Reproducible Research Using Cached Computations". In: *Computing in Science & Engineering* 11.1 (Sept. 2009), pp. 28–34. ISSN: 1521-9615. DOI: 10.1109/mcse.2009.6. URL: http://dx.doi.org/10.1109/mcse.2009.6 (cit. on p.).

[48] Fernando Perez and Brian E. Granger. "IPython: A System for Interactive Scientific Computing". In: *Computing in Science & Engineering* 9.3 (July 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/mcse.2007.53. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4160251 (cit. on p.).

[49] Prakash Prabhu, Yun Zhang, Soumyadeep Ghosh, David I August, Jialu Huang, Stephen Beard, Hanjun Kim, Taewook Oh, Thomas B Jablin, Nick P Johnson, Matthew Zoufaly, Arun Raman, Feng Liu, and David Walker. *A survey of the practice of computational science*. Nov. 2011, p. 19. DOI: 10.1145/2063348.2063374. URL: http://dx.doi.org/10.1145/2063348.2063374 (cit. on p.).

[50] Rémi Rampin, Fernando Chirigati, Dennis Shasha, Juliana Freire, and Vicky Steeves. "ReproZip: The Reproducibility Packer". In: *The Journal of Open Source Software* 1.8 (Dec. 2016). DOI: 10.21105/joss.00107. URL: https://doi.org/10.21105/joss.00107 (cit. on p.).

[51] Simon P. Sadedin, Bernard Pope, and Alicia Oshlack. "Bpipe: a tool for running and managing bioinformatics pipelines". In: *Bioinformatics* 28.11 (Dec. 2012), pp. 1525–1526. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts167. URL: http://dx.doi.org/10.1093/bioinformatics/bts167 (cit. on p.).

[52] Alan D Saul, James Hensman, Aki Vehtari, and Neil D Lawrence. "Chained Gaussian Processes". In: *BMC Bioinformatics* 14.1 (2016), pp. 1431–1440. ISSN: 1471-2105. DOI: 10.1186/1471-2105-14-252. URL: http://dx.doi.org/10.1186/1471-2105-14-252 (cit. on p.).

[53] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. "Making scientific computations reproducible". In: *Computing in Science & Engineering* 2.6 (2000), pp. 61–67. ISSN: 1521-9615. DOI: 10.1109/5992.881708. URL: http://dx.doi.org/10.1109/5992.881708 (cit. on p.).

[54] Jessica Severin, Kathryn Beal, Albert Vilella, Stephen Fitzgerald, Michael Schuster, Leo Gordon, Abel Ureta-Vidal, Paul Flicek, and Javier Herrero. "eHive: An Artificial Intelligence workflow system for genomic analysis". In: *BMC Bioinformatics* 11.1 (Oct. 2010), p. 240. ISSN: 1471-2105. DOI: 10.1186/1471-2105-11-240. URL: http://www.biomedcentral.com/1471-2105/11/240 (cit. on p.).

[55] Andreas Sorge. *pyfssa 0.7.6*. Dec. 2015. DOI: 10.5281/zenodo.35293. URL: https://doi.org/10.5281/zenodo.35293 (cit. on p.).

[56] Ola Spjuth, Erik Bongcam-Rudloff, Guillermo Carrasco Hernández, Lukas Forer, Mario Giovacchini, Roman Valls Guimera, Aleksi Kallio, Eija Korpelainen, Maciej M Kańduła, Milko Krachunov, David P Kreil, Ognyan Kulev, Paweł P. Łabaj, Samuel Lampa, Luca Pireddu, Sebastian Schönherr, Alexey Siretskiy, and Dimitar Vassilev. "Experiences with workflows for automating data-intensive bioinformatics". In: *Biology Direct* 10.1 (2015). ISSN: 1745-6150. DOI: 10.1186/s13062-015-0071-8. URL: http://dx.doi.org/10.1186/s13062-015-0071-8 (cit. on p.).

[57] S Sunagawa et al. "Structure and function of the global ocean microbiome". In: *Science* 348.6237 (2015), p. 1261359. ISSN: 0036-8075. DOI: 10.1126/science.1261359. URL: http://dx.doi.org/10.1126/science.1261359 (cit. on p.).

[58] Ian J Taylor, Ewa Deelman, Dennis B Gannon, and Matthew Shields. "Workflows for e-Science: scientific workflows for grids". In: (2014). URL: https://scholar.google.com/scholar?cluster=7040555504385453 83 (cit. on p.).

[59] P. Vandewalle, J. Kovacevic, and M. Vetterli. "Reproducible research in signal processing". In: *Signal Processing Magazine, IEEE* 26.3 (Sept. 2009), pp. 37–47. ISSN: 1053-5888. DOI: 10.1109/msp.2009.932122. URL: http://dx.doi.org/10.1109/msp.2009.932122 (cit. on p.).

[60] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. *Quality and productivity outcomes relating to continuous integration in GitHub*. 2015, pp. 805–816. DOI: 10.1145/2786805.2786850. URL: http://dx.doi.org/10.1145/2786805.2786850 (cit. on p.).

[61] H. Vuollekoski, M. Vogt, V. A. Sinclair, J. Duplissy, H. Järvinen, E.-M. Kyrö, R. Makkonen, T. Petäjä, N. L. Prisle, P. Räisänen, M. Sipilä, J. Ylhäisi, and M. Kulmala. "Estimates of global dew collection potential on artificial surfaces". In: *Hydrology and Earth System Sciences* 19.1 (2015), pp. 601–613. ISSN: 1027-5606. DOI: 10.5194/hess-19-601-2015. URL: http://dx.doi.org/10.5194/hess-19-601-2015 (cit. on p.).

[62] Stefan Van Der Walt, S Chris Colbert, and Gaël Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (Nov. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/mcse.2011.37. URL: http://dx.doi.org/10.1109/mcse.2011.37 (cit. on p.).

[63] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. "Best Practices for Scientific Computing". In: *PLoS Biology* 12.1 (2014), e1001745. ISSN: 1544-9173. DOI: 10.1371/journal.pbio.1001745. URL: http://dx.doi.org/10.1371/journal.pbio.1001745 (cit. on p.).

[64] Yihui Xie. "Dynamic Documents with R and knitr". In: 29 (2015). URL: https://scholar.google.com/scholar?cluster=17231182275289080006 (cit. on p.).

[65] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: cluster computing with working sets." In: *HotCloud* 10 (Oct. 2010). URL: https://scholar.google.com/scholar?cluster=14934743972440878 (cit. on p.).