signac: A Python framework for data and workflow management

Vyas Ramasubramani[‡*], Carl S. Adorf[‡], Paul M. Dodd[‡], Bradley D. Dice[¶], Sharon C. Glotzer[‡§¶‖]

https://youtu.be/CCKQH1M2uR4

Abstract—Computational research requires versatile data and workflow management tools that can easily adapt to the highly dynamic requirements of scientific investigations. Many existing tools require strict adherence to a particular usage pattern, so researchers often use less robust ad hoc solutions that they find easier to adopt. The resulting data fragmentation and methodological incompatibilities significantly impede research. Our talk showcases `signac`, an open-source Python framework that offers highly modular and scalable solutions for this problem. Named for the Pointillist painter Paul Signac, the framework's powerful workflow management tools enable users to construct and automate workflows that transition seamlessly from laptops to HPC clusters. Crucially, the underlying data model is completely independent of the workflow. The flexible, serverless, and schema-free `signac` database can be introduced into other workflows with essentially no overhead and no recourse to the `signac` workflow model. Additionally, the data model's simplicity makes it easy to parse the underlying data without using `signac` at all. This modularity and simplicity eliminates significant barriers for consistent data management across projects, facilitating improved provenance management and data sharing with minimal overhead.

Index Terms—data management, database, data sharing, provenance, computational workflow, hpc

Introduction

Streamlining data generation and analysis is a critical challenge for science in the age of big data and high performance computing (HPC). Modern computational resources can generate and consume enormous quantities of data, but process automation and data management tools have lagged behind. The highly file-based workflows characteristic of computational science are not amenable to traditional relational databases, and HPC applications require that data is available on-demand, enforcing strict performance requirements for any data storage mechanism. Building processes acting on this data requires transparent interaction with HPC clusters without sacrificing testability on personal computers, and these processes must be sufficiently malleable to adapt to changes in scientific inquiries.

* *Corresponding author: vramasub@umich.edu*
‡ *Department of Chemical Engineering, University of Michigan, Ann Arbor*
¶ *Department of Physics, University of Michigan, Ann Arbor*
§ *Department of Materials Science and Engineering, University of Michigan, Ann Arbor*
‖ *Biointerfaces Institute, University of Michigan, Ann Arbor*

To illustrate the obstacles that must be overcome, we consider a simple example in which we study the motion of an object through a fluid medium. If we initially model the motion only as a function of one parameter, an ad hoc solution for data storage would be to store the trajectories in paths named for the values of this parameter. If we then introduce some post-processing step, we could run it on each of these files. However, a problem arises if we realize that some additional parameter is also relevant. A simple solution might be to just rename the files to account for this parameter as well, but this approach would quickly become intractable if the parameter space increased further. A more flexible traditional solution involving the use of a relational MySQL [Cor16] database, for instance, might introduce undesirable setup costs and performance bottlenecks for file-based workflows on HPC. Even if we do employ such a solution, we also have to account for our workflow process: we need a way to run analysis and post-processing on just the new data points without performing unnecessary work on the old ones.

This paper showcases the `signac` framework, a data and workflow management tool that addresses these issues in a simple, powerful, and flexible manner (Fig. 1). The framework derives its name from the painter Paul Signac, one of the early pioneers of the Pointillist painting style. This style, in which paintings are composed of individual points of color rather than brushstrokes, provides an apt analogy for the underlying data model of the `signac` framework in which a data space is composed of individual data points that must be viewed together to make a complete picture. By storing JSON-encoded [Ecm17] metadata and the associated data together directly on the file system, `signac` provides database functionality such as searching and grouping data without the overhead of maintaining a server or interfacing with external systems, and it takes advantage of the high performance file systems common to HPC. Additionally, a `signac` database is entirely contained within a single root directory, making it compact and highly portable.

With `signac`, data space modifications like the one discussed above are trivially achievable with just a few lines of Python code. `signac`'s workflow component makes it just as easy to modify the process of data generation by simply defining the steps as Python functions. The workflow component of the framework, `signac-flow`, will immediately enable the use of these functions on the existing data space through a single command, and it tracks which tasks are completed to avoid redundancy. The resulting data can be accessed without reference to the workflow, ensuring that it is immediately available to anyone irrespective of

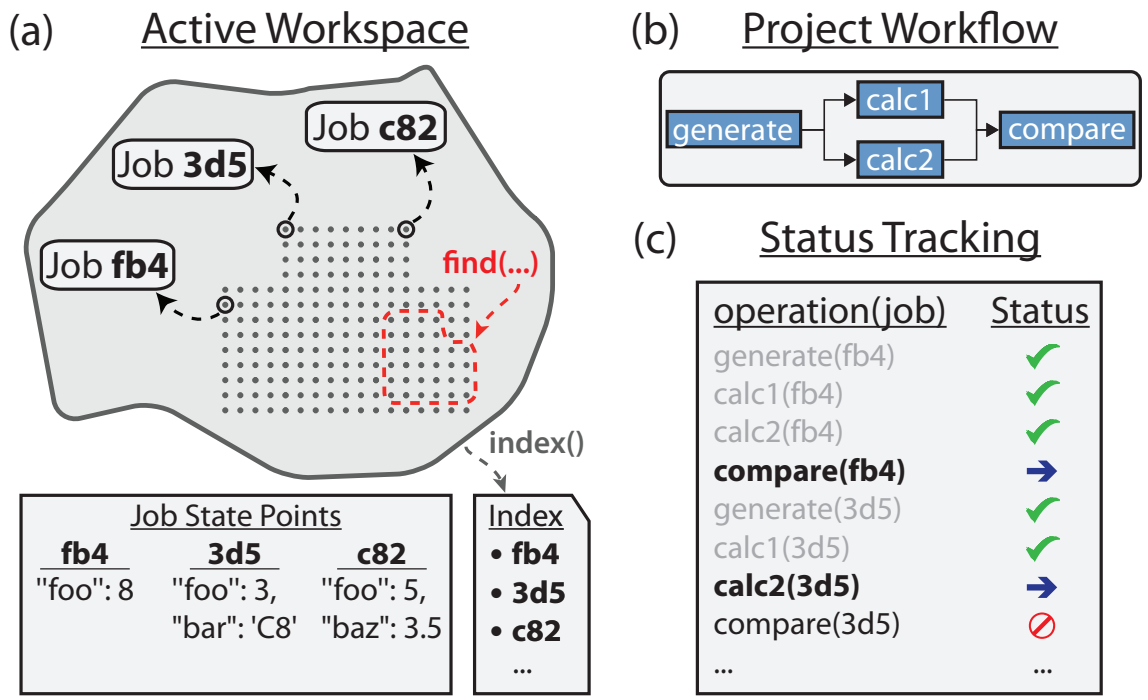

Fig. 1: *The data in a* `signac` *project (a) is contained in its workspace (dark grey outline), which in turn is composed of individual data points (grey points) that exist within some multidimensional parameter space (light grey background). Each data point, or job, is associated with a unique hash value (e.g., 3d5) computed from its state point, the unique key identifying the job. Using* `signac`*, the data can be easily searched, filtered, grouped, and indexed. To generate and act on this data space,* `signac` *can be used to define workflows (b), which are generically represented as a set of operations composing a directed graph. Using a series of pre- and post-conditions defined on these operations,* `signac` *tracks the progress of this workflow on a per-job basis (c) to determine whether a particular job is complete (greyed text, green check), eligible (bold text, blue arrow), or blocked (normal text, universal no).*

the tools they are using.

Overview and Examples

To demonstrate how `signac` works, we take a simple, concrete example of the scenario described above. Consider an experiment in which we want to find the optimal launch angle to maximize the distance traveled by a projectile through air. Figure 2 shows how we might organize the data associated with this study using `signac`. The central object in the `signac` data model is the *project*, which represents all the data associated with a particular instance of a `signac` data space. All of the project's data is contained within the *workspace* directory (see also Fig. 1). The workspace holds subdirectories corresponding to *jobs*, which are the individual data points in the data space. Each job is uniquely identified by its *state point*, which is an arbitrary key-value mapping. Although we see that these objects are stored in files and folders, we will show that these objects are structured in a way that provides layers of abstraction, making them far more useful than simple file system storage.

One could easily imagine interfacing existing scripts with this data model. The only requirement is some concept of a unique key for all data so that it can be inserted into the database. The unique key is what enables the creation of the 32 character hash, or *job id*, used to identify the job and its workspace folder (shown in Fig. 2). The uniqueness of this hash value is what enables `signac`'s efficient indexing and searching functionality. Additionally, this hash value is automatically updated to reflect any changes to individual jobs, making them highly mutable. For example, if we instead wanted to consider how changing initial velocity affects the distance traveled for a particular angle, we can add the velocity to the existing job state points by taking advantage of the fact that the project object is an iterable:

```
for job in project:
    job.sp.v = 1
```

In this case, we wanted to modify the entire workspace; more generally, however, we might want to modify only some subset of jobs. One way to accomplish this would be to apply a filter within the loop using conditionals based on the job state point, e.g. `if job.sp.theta < 5: job.sp.v = 1`. A more elegant solution, however, is to take advantage of `signac`'s query API, which allows the user to find only the jobs of interest using a dictionary as a filter. For example, in the above snippet we could replace `for job in project` with `for job in project.find_jobs()`, using an arbitrary dictionary as the argument to `find_jobs` to filter on the state point keys. The job finding functionality of `signac` is the entry point for its database functionality, which includes advanced indexing, selection, and grouping operations.

Having made the above change to our data space, we could now easily add new data points to test:

```
from numpy import linspace
for v in [1, 2, 3]:
    for theta in np.round(linspace(0, 1.57, 5), 2):
        sp = {"v": v, "theta": theta}
        project.open_job(sp).init()
```

```
In [1]: import signac
        project = signac.init_project("Projectiles")
        !ls
```

```
Notebook.ipynb  signac.rc
```

```
In [2]: job = project.open_job({"theta": 1.57})
        job.init()
        !find . -not -path '*/\.*'
```

```
.
./Notebook.ipynb
./signac.rc
./workspace
./workspace/4f8a64741a09749ac1320f4b61292e0c
./workspace/4f8a64741a09749ac1320f4b61292e0c/signac_statepoint.json
```

```
In [3]: print(job.get_id())
        print(job.statepoint())
```

```
4f8a64741a09749ac1320f4b61292e0c
{'theta': 1.57}
```

Fig. 2: *A very simple example using* `signac` *to create the basics of a data space. Initializing the project creates a* `signac.rc` *file, a configuration file identifying this folder as a* `signac` *project. The workspace directory is created when the first job is added to the project, and all job data is then stored in a subdirectory of the workspace. This subdirectory is named according to the job id, which is computed as the hash of the job state point. In this example, all work is conducted inside a Jupyter [PG07], [KRKP+16] notebook to indicate how easily this can be done. Note how fewer than ten lines of code are required to initialize a database and add data.*

Jobs that already exist in the data space will not be overwritten by the `init` operation, so there is no harm in performing a loop like this multiple times.

All of `signac`'s core functionality is not only available as a Python library, but also as a command line tool. This tool uses the Python `setuptools console_scripts` entry point, so it is automatically installed with `signac` and ships with built-in help information. This interface not only facilitates the integration of `signac` with non-Python code bases and workflows, it is also very useful for more ad hoc analyses of `signac` data spaces. For example, searching the database using the command line can be very useful for quick data inspection:

```
$ # Many simple queries are automatically
$ # translated into JSON
$ signac find theta 0.39
Interpreted filter arguments as '{"theta": 0.39}'.
d3012d490304c3c1171a273a50b653ad
1524633c646adce7579abdd9c0154d0f
22fa30ddf3cc90b1b79d19fa7385bc95

$ # Operators (e.g. less than) are available
$ # using a ".-operator" syntax
$ signac find v.\$lt 2
d61ac71a00bf73a38434c884c0aa82c9
00e5f0c36294f0eee4a30cabb7c6046c
585599fe9149eed3e2dced76ef246903
22fa30ddf3cc90b1b79d19fa7385bc95
9fa1900a378aa05b9fd3d89f11ef0e5b

$ # More complex queries can be constructed
$ # using JSON directly
$ signac find '{"theta": {"$in": [0, 0.78]}}'
2faf0f76bde3af984a91b5e42e0d6a0b
585599fe9149eed3e2dced76ef246903
03d50a048c0423bda80c9a56e939f05b
3201fd381819dde4329d1754233f7b76
d61ac71a00bf73a38434c884c0aa82c9
13d54ee5821a739d50fc824214ae9a60
```

The query syntax is based on the MongoDB [Mon16] syntax and enables, for instance, logical and arithmetic operators. In fact, `signac` natively supports export of its databases to MongoDB. Although we can add support for integration with any database management system, we started with MongoDB for two reasons: first, because researchers are likely to prefer the comparatively less rigid approach of NoSQL databases to table-based relational databases; and second, because translation from a `signac` database to another JSON-based database is relatively straightforward. Due to the ease of export and shared query syntax, switching between `signac` and MongoDB is quite easy.

At any point, we can also get an overview of what the implicit data space schema looks like:

```
$ signac schema
{
 'theta': 'float([0.0, ..., 1.57], 5)',
 'v': 'int([1, 2, 3], 3)',
}
```

Keys with constant values across the entire data space can be optionally omitted from the schema. Additionally, schema can be filtered, nested keys can be compressed to specified depths, and the number of entries shown in each range can be limited as desired.

Workflows

The `signac` database is intended to be usable as a drop-in solution for data management issues. The `signac` framework, however, is designed to simplify the entire process of data generation, which includes clearly defining the processes that generate

and operate on the data cleanly and concisely. To manage workflows, the `signac-flow` component of the framework provides the `FlowProject` class (not to be confused with the `signac Project` class that interfaces with the data in a `signac` project). The `FlowProject` encodes operations acting on `signac` data spaces as well as the sequence information required to string these operations together into a complete workflow. In Fig. 3, we demonstrate how `signac-flow` can be used to automate our projectile investigation.

In this script, we register a simple function `calculate` as an operation with the `FlowProject.operation` decorator. We store our output in the *job document*, a lightweight JSON storage mechanism that `signac` provides, and we check the document to determine when the operation has been completed using the `FlowProject.post` decorator. Any function of a job can be used as a pre- or post-condition. In this case, we simply look for the `tmax` key in the job document using the `complete` function. Note the `FlowProject.label` decorator for this function; we will discuss this in further detail below.

Although this particular example is quite simple, in principle any workflow that can be represented by a directed graph may be encoded and executed using `signac-flow`. In the context of `signac-flow`, individual operations are the nodes of a graph, and the pre- or post-conditions associated with each operation determine the vertices. To simplify running such workflows, by default the `project.py run` interface demonstrated in Fig. 3 will automatically run the entire workflow for every job in the workspace. When conditions are defined in the manner shown above, `signac-flow` will ensure that only incomplete tasks are run, i.e., in this example, once `tmax` has been calculated for a particular job, the `calculate` operation will not run again for that job. Rather than running everything at once, it is also possible to exercise more fine-grained control over which operations to run using `signac-flow`:

```
$ # Runs all outstanding operations for all jobs
$ python project.py run
$ # `exec` ignores the workflow and just runs a
$ # specific job-operation
$ python project.py exec ${OP} ${JOB_ID}
$ # Run up to two operations for a specific job
$ python project.py run -j ${JOB_ID} -n 2
```

A critical feature of the `signac` framework is its scalability to HPC. The file-based data model is designed to leverage the high performance file systems common on such systems, and workflows designed locally are immediately executable on HPC clusters. In particular, any operation that can be successfully executed in the manner shown in Fig. 3 can also be immediately submitted to cluster schedulers. The `signac-flow` package achieves this by creating cluster job scripts that perform the above operations:

```
$ # Print the script for one 12-hour job
$ # Additional scheduler directives are customizable
$ python project.py submit -n 1 -w 12 --pretend
Query scheduler...
Submitting cluster job 'Projectiles/d61...':
 - Operation: calculate(d61...)
#PBS -N Projectiles/d61...
#PBS -l walltime=12:00:00
#PBS -l nodes=1
#PBS -V

set -e
set -u
cd /path/to/project

# Operation 'calculate' for job 'd61...':
python project.py exec calculate d61
```

The workflow tracking functionality of `signac-flow` also extends to compute clusters. Users can always check the status of particular jobs to see how far they have progressed in the workflow, and when working on a system with a scheduler, `signac-flow` will automatically provide information about the status of jobs submitted to the scheduler. Depending on the desired verbosity, this status information can be output in a variety of formats. A relatively detailed version of the output is shown here:

```
$ # Submit 3 random jobs for 12 hours
$ python project.py submit -n 3 -w 12
$ # Status output has options to control detail
$ python project.py status -de
# Overview:
Total # of jobs: 15

label       ratio
-------     -------
complete    |#----------------------------| 6.67%

# Detailed View:

## Labels:
job_id                                labels
--------------------------------  --------
00e5f0c36294f0eee4a30cabb7c6046c  complete
d61ac71a00bf73a38434c884c0aa82c9
...

## Operations:
job_id   operation    eligible    cluster_status
------  -----------  ----------  ----------------
d61ac7  calculate    Y           Q
41dea8  calculate    Y           A
585599  calculate    Y           Q
2fc415  calculate    Y           I
...
```

In the overview section, we see that 6.67%, or $\frac{1}{15}$ jobs have completed, reflecting the job run locally in Fig. 3. The rows in this section are populated by any function decorated with the `FlowProject.label` decorator, with each row showing the percentage of jobs that evaluate to `True` for that function. While any callable, such as a lambda expression, could be used as a pre- or post-condition, using a function decorated in this manner makes it easy to track total progress through the workflow. The labels section below the overview provides the same information on a per-job basis, in this case showing which jobs have completed and which have not.

Finally, the operations section indicates the progress of jobs on a per-operation basis. In this particular view, the `eligible` column is redundant because we have omitted completed operations for brevity; however, if we requested a complete listing, the job marked as complete in the labels section would be listed here with an `N` in the eligible column. In this instance, there are fourteen jobs remaining that are eligible for the `calculate` operation, of which three have been submitted to the cluster (and are therefore marked as active). Of these three, one has actually begun running (and is marked as `[A]`), while the other two indicate that they are queued (marked as `[Q]`). The final job shown is inactive on the cluster (`[I]`) as it has not yet been submitted.

The quick overview of this section highlights the core features of the `signac` framework. Although the example demonstrated here is quite simple, the data model scales easily to thousands of

```
In [4]: %%writefile project.py
        import flow
        from flow.project import FlowProject

        @FlowProject.label
        def complete(job):
            return 'tmax' in job.document

        @FlowProject.operation
        @FlowProject.post(complete)
        def calculate(job):
            import numpy as np
            g = 9.81
            roots = np.roots([-g/2, np.sin(job.sp.theta), 0])
            tmax = roots[roots != 0][0]
            job.doc.tmax = tmax

        if __name__ == "__main__":
            FlowProject().main()

        Writing project.py

In [5]: !python project.py run

        Execute operation 'calculate(4f8a64741a09749ac1320f4b61292e0c)'...

In [6]: !find workspace
        job.document()

        workspace
        workspace/4f8a64741a09749ac1320f4b61292e0c
        workspace/4f8a64741a09749ac1320f4b61292e0c/signac_job_document.json
        workspace/4f8a64741a09749ac1320f4b61292e0c/signac_statepoint.json

Out[6]: {'tmax': 0.2038735337271834}
```

Fig. 3: *The `signac-flow` module enables the easy automation of workflows operating on `signac` workspaces. Here we demonstrate such a workflow operating on the data space defined in Fig. 2. In this case, the workspace consists only of one job; the real power of the FlowProject arises from its ability to automatically handle an arbitrary sequence of operations on a large number of jobs. Note that in this figure we are still assuming `v=1` for simplicity.*

data points and far more complex and nonlinear workflows. More involved demonstrations can be seen in the documentation[1], on the `signac` website[2], or in the original paper published in the Journal of Computational Materials Science [ADRG18].

Design and Implementation

Having provided an overview of `signac`'s functionality, we will now delve into the specifics of its implementation. The central element of the framework is the `signac` data management package, which provides the means for organizing data directly on the filesystem. The primary requirement for using this database is that every job (data point) in the data space must be uniquely indexable by some set of key-value pairs, namely the job state point. The hash of this state point defines the job id, which in turn is used to define the directory where data associated with this job is stored. To ensure that the state point associated with the job id can be recovered, a JSON-encoded copy of the state point is stored within this directory.

This storage mechanism enables $O(1)$ access to the data associated with a particular state point through its hash as well as $O(N)$ indexing of the data space. This indexing is performed by traversing the data space and parsing the state point files directly; other files may also be parsed along the way if desired. In general, `signac` automatically caches generated indexes within a single session where possible, but for better performance after start-up

1. http://signac.readthedocs.io
2. http://signac.io

the indexes can also be stored persistently. These indexes then allow efficient selection and searching of the data space, and MongoDB-style queries can be used for complex selections.

This distributed mode of operation is well-suited to the high performance filesystems common to high performance computing. The explicit horizontal partitioning and distributed storage of data on a per-job basis is well suited to HPC operations, which are typically executed for multiple jobs in parallel. Since data is accessed distributively, there is no inherent bottleneck posed by funneling all data read and write operations through one or more server applications. Further sharding across multiple filesystems, for instance, could be accomplished by devising a scheme to divide a project's data into multiple workspaces that would then be indexed independently.

From the Python implementation standpoint, the central component to the `signac` framework is the `Project` class, which provides the interface to `signac`'s data model and features. In addition to the core index-related functionality previously mentioned, the `signac` `Project` also encapsulates numerous additional features, including, for example, the generation of human-readable views of the hash-obfuscated workspace; the ability to move, copy, or clone a full project; the ability to synchronize data across projects; and the detection of implicit schema. We qualify these schema as implicit because they are only defined by the state points of jobs within the workspace, *i.e* there is nothing like a table schema to enforce a particular structure for the state points of individual jobs. Searching through or iterating over a `Project` instance generates `Job` objects, which provide Python interfaces to the jobs within the project and their associated data. In addition to providing a Pythonic access point to the job state point and the job document, a `Job` object can always be mapped to its location on the filesystem, making it ideal for associating file-based data with the appropriate data point.

The central object in the `signac-flow` package is the `FlowProject` class, which encapsulates a set of operations acting on a `signac` data space. There is a tight relationship between the `FlowProject` and the underlying data space, because operations are in general assumed to act on a per-job basis. Using the sequence of conditions associated with each operation, a `FlowProject` also tracks workflow progress on per-job basis to determine which operations to run next for a given job. Different HPC environments and cluster schedulers are represented by separate Python classes that provide the means for querying schedulers for cluster job statuses, writing out the job scripts, and constructing the submission commands. Job scripts are created using templates written in `jinja2` [Ron], making them easily customizable for the requirements of specific compute clusters or users. This means that workflows designed on one cluster can be easily ported to another, and that users can easily contribute new environment configurations that can be used by others. Currently, we support Slurm and TORQUE schedulers, along with more specialized support for the following supercomputers (listed along with their funding organizations): XSEDE Comet, XSEDE Stampede, XSEDE Bridges, INCITE Titan, INCITE Eos, and the University of Michigan Flux clusters.

The `signac` framework prioritizes modularity and interoperability over monolithic functionality, making it readily extensible. One of the tools built on top of the core infrastructure is `signac-dashboard` [Bra18], a web interface for visualizing `signac` data spaces that is currently under active development. All tools in the framework, including `signac-flow`, share the `signac` database as a core dependency. Aside from that, however, core `signac` and `signac-flow` avoid any hard dependencies and are implemented as pure Python packages compatible with Python 2.7 and 3.3+. In conjunction with the framework's full-featured command line interface, these features of the framework ensure that it can be easily incorporated into any existing file-based workflows, even those using primarily non-Python tools.

Comparisons

In recent years, many Python tools have emerged to address issues with data provenance and workflow management in computational science. While some are very similar to the `signac` framework in their goals, a major distinction between `signac` and other comparable tools is that the `signac` data management component is independent of `signac-flow`, making it much easier to interact with the data outside the context of the workflow. As a result, while these packages solve problems similar to those addressed by `signac`, they take different and generally less modular approaches to doing so. Other packages have focused on the distinct but related need for complete provenance management for reproducibility. These tools are orthogonal to `signac` and may be used in conjunction with it.

Workflow and Provenance Management

Two of the best-known, most comparable Python workflow managers are Fireworks [JOC+15] and AiiDA [PCS+16]. Fireworks and AiiDA are full-featured workflow managers that, like `signac-flow`, interface with high performance compute clusters to execute complex, potentially nonlinear workflows. These tools in fact currently offer more powerful features than `signac-flow` for monitoring the progress of jobs, features that are supported by the use of databases on the back end. However, maintaining a server for workflow management can be cumbersome, and it introduces additional unnecessary complexities.

A more significant limitation of these other tools is that their data representations are closely tied to the workflow execution, making it much more challenging to access the data outside the context of the workflow. Concretely, these software typically store data in a specific location based on a particular instance of an operation's execution, so the data can only be found by looking for that specific instance of the operation. Conversely, in `signac` the data is identified by its own metadata, namely its state point, so once it has been generated its access is no longer linked to a specific instance of a `signac-flow` operation (assuming that `signac-flow` is being used at all).

Of course, knowing exactly where and how data was generated and transformed, *i.e.*, the data provenance, is also valuable information. Two tools that are specialized for this task are Sacred [GKC+17] and Sumatra [Dav12]. Superficially, the `signac` framework appears especially similar to Sacred. Both use decorators to convert functions into executable operations, and configurations can be injected into these functions (in `signac`'s case, using the job object). Internally, Sacred and `signac-flow` both depend on the registration of particular functions with some internal API: in `signac-flow`, functions are stored as operations within the `FlowProject`, whereas Sacred tracks functions through the `Experiment` class. However, the focus of Sacred is not to store data or execute workflows, but instead to track when an operation was executed, the configuration that was used, and what output was generated. Therefore, in principle `signac` and

Sacred are complementary pieces of software that could be used in concert to achieve different benefits.

We have found that integrating Sacred with `signac` is in fact quite simple. Once functions are registered with either a Sacred `Experiment` or a `signac-flow` `FlowProject`, the operations can be run either through Python or on the command line. While both tools typically advocate using their command line interfaces, the two can be integrated by using one from the command line while having it internally call the other through the corresponding Python interface. When used in concert with `signac`, the primary purpose of the Sacred command line interface, the ability to directly interact with the configuration, is instead being managed by the underlying `signac` database; in principle, the goal of this integration would be to have all configuration information tracked using `signac`. Conversely, `signac-flow`'s command line interface offers not only the ability to specify which parts of the workflow to run, but also to query status information or submit operations to a scheduler with a particular set of script options. As a result, to optimally utilize both tools, we advocate using the `signac-flow` command line functionality and encoding a Sacred Experiment within a `signac-flow` operation.

The Sumatra provenance tracking tool is an alternative to Sacred. Although it is written in Python, it is primarily designed for use as a command line utility, making it more suitable than Sacred for non Python application. However, it does provide a Python API that offers greater flexibility than the command line tool, and this is the recommended mode for integration with `signac-flow` operations.

Data Management

We have found fewer alternatives to direct usage of the `signac` data model; as mentioned previously, most currently existing software packages tightly couple their data representation with the workflow model. The closest comparison that we have found is datreant [DSL+16], which provides the means for interacting with files on the file system along with some features for finding, filtering, and grouping. There are two primary distinctions between datreant and `signac`: `signac` requires a unique key for each data point, and `signac` offers a tightly integrated workflow management tool. The datreant data model is even simpler than `signac`'s, which provides additional flexibility at the cost of `signac`'s database functionality. This difference is indicative of datreant's focus on more general file management problems than the issues `signac` is designed to solve. The generality of the datreant data model makes integrating it into existing workflows just as easy as integrating `signac`, and the MDSynthesis package [Dot15] is one example of a workflow tool built around a datreant-managed data space. However, MDSynthesis is highly domain-specific and it cannot be used for other types of computational studies. Therefore, while the combination of MDSynthesis and datreant is a comparable tool to the `signac` framework in the field of molecular simulation, it does not generalize to other use-cases.

Conclusions

The `signac` framework provides all the tools required for thorough data and workflow management in scientific computing. Motivated by the need for managing the dynamic, heterogeneous data spaces characteristic in computational sciences, the tools are tailored for the use-cases most commonly faced in this field. The framework has strived to achieve high ease of use and interoperability by emphasizing simple interfaces, minimizing external requirements, and employing open data formats like JSON. By doing so, the framework aims to minimize the initial barriers for new users, making it easy for researchers to begin using `signac` with little effort. The framework frees computational scientists from repeatedly solving common data and workflow problems throughout their research, and at a higher level, reduces the burden of data sharing and provenance tracking, both of which are critical to accelerating the production of reproducible and reusable scientific results.

Acknowledgments

We would like to thank all contributors to the development of the framework's components, J.A. Anderson, M.E. Irrgang and P. Damasceno for fruitful discussion, feedback and support, and B. Swerdlow for his contributions and feedback and coming up with the name. We would also like to thank all early adopters that provided feedback and thus helped in guiding and improving the development process. Development and deployment supported by MICCoM, as part of the Computational Materials Sciences Program funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences, Materials Sciences and Engineering Division, under Subcontract No. 6F-30844. Project conceptualization and implementation supported by the National Science Foundation, Award # DMR 1409620.

References

[ADRG18] Carl S. Adorf, Paul M. Dodd, Vyas Ramasubramani, and Sharon C. Glotzer. Simple data and workflow management with the signac framework. *Computational Materials Science*, 146:220 – 229, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0927025618300429.

[Bra18] Bradley D. Dice. signac-dashboard, 2018. URL: https://bitbucket.org/glotzer/signac-dashboard/src/master/.

[Cor16] Oracle Corporation. Mysql, 2016. URL: https://www.mysql.com.

[Dav12] Andrew P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Comput. Sci. Eng.*, 14:48–56, 2012.

[Dot15] David L. Dotson. MDSynthesis: a Python package enabling data-driven molecular dynamics research, July 2015.

[DSL+16] David L. Dotson, Sean L. Seyler, Max Linke, Richard J. Gowers, and Oliver Beckstein. datreant: persistent, pythonic trees for heterogeneous data. In S Benthall and S Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 51–56, Austin, TX, 2016.

[Ecm17] Ecma. The JSON Data Interchange Syntax, December 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[GKC+17] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The sacred infrastructure for computational research. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49–56, Austin, TX, 2017.

[JOC+15] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, Daniel Gunter, and Kristin A. Persson. Fireworks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning*

and Power in Academic Publishing: Players, Agents and Agendas, pages 87 – 90. IOS Press, 2016.

[Mon16] MongoDB, Inc. MongoDB, 2016. URL: https://www.mongodb.com/.

[PCS+16] Giovanni Pizzi, Andrea Cepellotti, Riccardo Sabatini, Nicola Marzari, and Boris Kozinsky. AiiDA: automated interactive infrastructure and database for computational science. *Comput. Mater. Sci.*, 111:218–230, 2016.

[PG07] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. URL: http://ipython.org.

[Ron] Armin Ronacher. jinja2. Accessed on 2017/09/29. URL: http://jinja.pocoo.org/.