signac Documentation

Carl Simon Adorf, Vyas Ramasubramani, Bradley Dice
The **signac** framework supports researchers in managing project-related data with a well-defined indexable storage layout for data and metadata. This streamlines post-processing and analysis and makes data collectively accessible.

This is the overall **framework documentation**. It provides a comprehensive overview on what you can do with packages that are part of the **signac** framework. If you are new to **signac**, the best place to start is to read the *Introduction* and the *Tutorial*. 
1.1 Introduction

The signac framework is designed to simplify the storage, generation and analysis of multidimensional data sets associated with large-scale, file-based computational studies. Any computational work that requires you to manage files and execute workflows may benefit from an integration with signac. Typical examples include hyperparameter optimization for machine learning applications and high-throughput screening of material properties with various simulation methods. The data model assumes that the work can be divided into so called projects, where each project is roughly confined by similarly structured data, e.g., a parameter study.

In signac, the elements of a project’s data space are called jobs. Every job is defined by a unique set of well-defined parameters that define the job’s context, and it also contains all the data associated with this metadata. This means that all data is uniquely addressable from the associated parameters. With signac, we define the processes generating and manipulating a specific data set as a sequence of operations on a job. Using this abstraction, signac can define workflows on an arbitrary signac data space.

1.2 Installation

All packages in the signac framework depend on the core signac package, which provides the data management functionality used by all other packages (See Packages (API) for more information). Most users should install the
signac and the signac-flow packages, which are tested for Python version 2.7.x and 3.4+ and do not have any hard dependencies, ensuring that no packages outside the signac framework are required for basic functionality. Please see the individual package documentation for instructions on how to install additional packages.

1.2.1 Install with conda

The recommended installation method for installing signac packages is via conda. The signac packages are distributed via the conda-forge channel. For a standard installation, execute:

```
$ conda install -c conda-forge signac signac-flow
```

Tip: Consider adding the conda-forge channel to your default channels with: $ conda config --add channels conda-forge.

1.2.2 Install with pip

For a standard installation with pip, execute:

```
$ pip install --user signac signac-flow
```

Note: If you want to install packages for all users on a machine, you can remove the --user option in the install command.

1.2.3 Installation from Source

Alternatively, you can clone any of the package’s source code repositories and install them manually. For example, to install the signac core package you can execute the following code:

```
git clone https://github.com/glotzerlab/signac.git
cd signac
python setup.py install --user
```

Note: If you want to install packages for all users on a machine, you can remove the --user option in the install command.

1.3 Quickstart

To get started, first install signac and then setup a new project with:

```
~ $ mkdir my_project
~ $ cd my_project/
~/my_project $ signac init MyProject
Initialized project 'MyProject'.
```
Once a project has been created, the next step is to initialize the *data space* with, e.g., a script called *init.py*:

```python
# init.py
import signac

project = signac.get_project()

for foo in range(3):
    project.open_job({'foo': foo}).init()
```

```
~/my_project $ python init.py
```

The key is using the Python *project* handle as the interface to initialize jobs (data points) in your data space. You can then implement a simple *data space operation* within a *project.py* script:

```python
# project.py
from flow import FlowProject

@FlowProject.operation
def hello_job(job):
    print("Hello from job {}, my foo is '{}'.".format(job, job.sp.foo))

if __name__ == '__main__':
    FlowProject().main()
```

Note the use of the `FlowProject.operation()` decorator to indicate that the `hello_job` function should be interpreted as an operation acting on the data space.

Operations can be executed for all of your jobs with:

```
~/my_project $ python project.py run
Execute operation 'hello_job(15e548a2d943845b33030e68801bd125)'
Hello from job 15e548a2d943845b33030e68801bd125, my foo is '1'.
Execute operation 'hello_job(2b985fa90138327bef586f9ad87fc310)'
Hello from job 2b985fa90138327bef586f9ad87fc310, my foo is '2'.
Execute operation 'hello_job(7f3e901b4266f28348b38721c099d612)'
Hello from job 7f3e901b4266f28348b38721c099d612, my foo is '0'.
```

See the *Tutorial* for a more detailed introduction to how to use *signac* to manage data and implement workflows.

### 1.4 Tutorial

**License**

The code shown in this tutorial is part of the *Examples*. It can be downloaded from the *signac-docs* repository and is released into the public domain.
This tutorial is designed to step new users through the basics of setting up a signac data space, defining and executing a simple workflow, and analyzing the data. For the complete code corresponding to this tutorial, see the Ideal Gas example.

### 1.4.1 Basics

#### Initializing the data space

In this tutorial, we will perform a simple study of the pressure-volume \((p-V)\) relationship of a noble gas. As a first approximation, we could model the gas as an ideal gas, so the ideal gas law applies:

\[
pV = Nk_B T
\]

Therefore, we can assume that the volume \(V\) can be directly calculated as a function of system size \(N\), Boltzmann’s constant \(k_B\), and temperature \(T\).

To test this relationship, we start by creating an empty project directory where we will place all the code and data associated with this computational study.

```bash
~ $ mkdir ideal_gas_project
~ $ cd ideal_gas_project/
~/ideal_gas_project $
```

We then proceed by initializing the data space within a Python script called `init.py`:

```python
# init.py
import signac

project = signac.init_project('ideal-gas-project')

for p in range(1, 10):
    sp = {'p': p, 'kT': 1.0, 'N': 1000}
    job = project.open_job(sp)
    job.init()
```

The `signac.init_project()` function initializes the signac project in the current working directory by creating a configuration file called `signac.rc`. The location of this file defines the project root directory. We can access the project interface from anywhere within and below the root directory by calling the `signac.get_project()` function, or from outside this directory by providing an explicit path, e.g., `signac.get_project('~/ideal_gas_project')`.

**Note:** The name of the project stored in the configuration file is independent of the directory name it resides in.

We can verify that the initialization worked by examining the implicit schema of the project we just created:

```bash
~/ideal_gas_project $ signac schema
{
    'N': 'int([1000], 1)',
    'kT': 'float([1.0], 1)',
    'p': 'int([1, 2, 3, ..., 9, 10], 10)',
}
```

The output of the `$ signac schema` command gives us a brief overview of all keys that were used as well as their value (range).
**signac Documentation**

**Note:** The `job.init()` function is idempotent, meaning that it is safe to call it multiple times even after a job has already been initialized. It is good practice make all steps that are part of the data space initialization routine idempotent.

---

**Exploring the data space**

The core function that **signac** offers is the ability to associate metadata — for example, a specific set of parameters such as temperature, pressure, and system size — with a distinct directory on the file system that contains all data related to said metadata. The `open_job()` method associates the metadata specified as its first argument with a distinct directory called a **job workspace**. These directories are located in the `workspace` sub-directory within the project directory and the directory name is the so called **job id**.

```bash
~/ideal_gas_project $ ls -1 workspace/
03585df0f87fada67bd0f540c102cce7
22a51374466c4e01ef0e67e65f73c52e
71855b321a04dd9ee27ce6c9cc0436f4
# ...
```

The **job id** is a highly compact, unambiguous representation of the full metadata, i.e., a distinct set of key-value pairs will always map to the same job id. However, it can also be somewhat cryptic, especially for users who would like to browse the data directly on the file system. Fortunately, you don’t need to worry about this internal representation of the data space while you are actively working with the data. Instead, you can create a linked view with the **signac view** command:

```bash
~/ideal_gas_project $ signac view
~/ideal_gas_project $ ls -d view/p/*
view/p/1 view/p/2 view/p/3 view/p/4 view/p/5 view/p/6 view/p/7 view/p/8 view/p/9
```

The linked view is the most compact representation of the data space in form of a nested directory structure. Most compact means in this case, that **signac** detected that the values for $kT$ and $N$ are constant across all jobs and are therefore safely omitted. It is designed to provide a human-readable representation of the metadata in the form of a nested directory structure. Each directory contains a job directory, which is a symbolic link to the actual workspace directory.

**Note:** Make sure to update the view paths by executing the `$ signac view` command (or equivalently with the `create_linked_view()` method) everytime you add or remove jobs from your data space.

---

**Interacting with the signac project**

You interact with the **signac** project on the command line using the **signac** command. You can also interact with the project within Python via the **signac.Project** class. You can obtain an instance of that class within the project root directory and all sub-directories with:

```python
>>> import signac
>>> project = signac.get_project()
```

Iterating through all jobs within the data space is then as easy as:
We can iterate through a select set of jobs with the `find_jobs()` method in combination with a query expression:

```python
>>> for job in project.find_jobs({"kT": 1.0, "p.$lt": 3.0}):  
    print(job, job.sp.p)
... 742c883cbee8e417b3bb236d40a99533 1
  ee550647e3f707b251eb094f43d434c 2
>>>
```

In this example we selected all jobs, where the value for $kT$ is equal to 1.0 – which would be all of them – and where the value for $p$ is less than 3.0. The equivalent selection on the command line would be achieved with `$ signac find kT 1.0 p.$lt 3.0`. See the detailed `Query API` documentation for more information on how to find and select specific jobs.

**Note:** The following expressions are all equivalent: `for job in project:`, `for job in project.find_jobs():`, and `for job in project.find_jobs(None):`

### Operating on the data space

Each job represents a data set associated with specific metadata. The point is to generate data which is a function of that metadata. Within the framework’s language, such a function is called a data space operation.

Coming back to our example, we could implement a very simple operation that calculates the volume $V$ as a function of our metadata like this:

```python
def volume(N, kT, p):
    return N * kT / p
```

Let’s store the volume within our data space in a file called `volume.txt`, by implementing this function in a Python script called `project.py`:

```python
# project.py
import signac

def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn('volume.txt'), 'w') as file:
        file.write(str(volume) + '

project = signac.get_project()
for job in project:
    compute_volume(job)
```

Executing this script will calculate and store the volume for each pressure-temperature combination in a file called `volume.txt` within each job’s workspace.
Note: The `job.fn('volume.txt')` expression is a short-cut for `os.path.join(job.workspace(), 'volume.txt')`.

### 1.4.2 Workflows

#### Implementing a simple workflow

In many cases, it is desirable to avoid the repeat execution of data space operations, especially if they are not idempotent or are significantly more expensive than our simple example. For this, we will incorporate the `compute_volume()` function into a workflow using the `FlowProject` class. We slightly modify our `project.py` script:

```python
# project.py
from flow import FlowProject

@FlowProject.operation
def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn('volume.txt'), 'w') as file:
        file.write(str(volume) + '

if __name__ == '__main__':
    FlowProject().main()
```

The `operation()` decorator identifies the `compute_volume` function as an operation function of our project. Furthermore, it is now directly executable from the command line via an interface provided by the `main()` method.

We can then execute all operations defined within the project with:

```bash
~/ideal_gas_project $ python project.py run
Execute operation 'compute_volume(03585df0f87fada67bd0f540c102cce7)'
Execute operation 'compute_volume(22a51374466c4e01ef0e67e65f73c52e)'
Execute operation 'compute_volume(71855b321a04dd9ee27ce6c9cc0436f4)'
# ...
```

However, if you execute this in your own terminal, you might have noticed a bunch of warning messages printed out at the end, that read similar to:

```bash
Operation 'compute_volume(03585df0f87fada67bd0f540c102cce7)' exceeds max. # of
→ allowed passes (1).
Operation 'compute_volume(22a51374466c4e01ef0e67e65f73c52e)' exceeds max. # of
→ allowed passes (1).
# and so on
```

That is because by default, the `run` command will continue to execute all defined operations until they are considered completed. An operation is considered completed when all its post conditions are met, and it is up to the user to define those post conditions. Since we have not defined any post conditions yet, `signac` would continue to execute the same operation indefinitely.

For this example, a good post condition would be the existence of the `volume.txt` file. To tell the `FlowProject` class when an operation is completed, we can modify the above example by adding a function that defines this condition:
# project.py
from flow import FlowProject

def volume_computed(job):
    return os.path.isfile("volume.txt")

@FlowProject.operation
@FlowProject.post(volume_computed)
def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn('volume.txt'), 'w') as file:
        file.write(str(volume) + '\n')

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

Tip: Simple conditions can be conveniently defined inline as lambda expressions: @FlowProject.post(lambda job: job.isfile("volume.txt")). We can check that we implemented the condition correctly by executing $ python project.py run again. This should now return without any message because all operations have already been completed.

Note: To simply, execute a specific operation from the command line ignoring all logic, use the exec command, e.g.: $ python project.py exec compute_volume. This command (as well as the run command) also accepts jobs as arguments, so you can specify that you only want to run operations for a specific set of jobs.

Extending the workflow

So far we learned how to define and implement data space operations and how to define simple post conditions to control the execution of said operations. In the next step, we will learn how to integrate multiple operations into a cohesive workflow.

First, let’s verify that the volume has actually been computed for all jobs. For this we transform the volume_computed() function into a label function by decorating it with the label() decorator:

# project.py
from flow import FlowProject

@FlowProject.label
def volume_computed(job):
    return job.isfile("volume.txt")

# ...

We can then view the project’s status with the status command:

~/ideal_gas_project $ python project.py status
Generate output...

(continues on next page)
Status project 'ideal-gas-project':
Total # of jobs: 10

<table>
<thead>
<tr>
<th>label</th>
<th>progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume_computed</td>
<td>#</td>
</tr>
</tbody>
</table>

That means that there is a `volume.txt` file in each and every job workspace directory.

Let’s assume that instead of storing the volume in a text file, we wanted to store it in a JSON file called `data.json`. Since we are pretending that computing the volume is an expensive operation, we will implement a second operation that copies the result stored in the `volume.txt` file into the `data.json` file instead of recomputing it:

```python
# project.py
from flow import FlowProject
import json

@FlowProject.operation
@FlowProject.pre(volume_computed)
@FlowProject.post.isfile("data.json")
def store_volume_in_json_file(job):
    with open(job.fn("volume.txt")) as textfile:
        with open(job.fn("data.json"), "w") as jsonfile:
            data = {"volume": float(textfile.read())}
            jsonfile.write(json.dumps(data) + ",
```

Here we reused the `volume_computed` condition function as a pre-condition and took advantage of the post.isfile short-cut function to define the post-condition for this operation function.

**Important:** An operation function is **eligible** for execution if all pre-conditions are met, at least one post-condition is not met and the operation is not currently submitted or running.

Next, instead of running this new function for all jobs, let’s test it for one job first.

```bash
~/ideal_gas_project $ python project.py run -n 1
Execute operation 'store_volume_in_json_file(742c883cbee8e417bbb236d40aea9543)'...
```

We can verify the output with:

```bash
~/ideal_gas_project $ cat workspace/742c883cbee8e417bbb236d40aea9543/data.json
{"volume": 1000.0}
```

Since that seems right, we can then store all other volumes in the respective `data.json` files by executing `$ python project run`.

**Tip:** We could further simplify our workflow definition by replacing the `pre(volume_computed)` condition with `pre.after(compute_volume)`, which is a short-cut to reuse all of `compute_volume()`’s post-conditions as pre-conditions for the `store_volume_in_json_file()` operation.
The job document

Storing results in JSON format – as shown in the previous section – is good practice because the JSON format is an open, human-readable format, and parsers are readily available in a wide range of languages. Because of this, signac stores all metadata in JSON files and in addition comes with a built-in JSON-storage container for each job (see: The Job Document).

Let’s add another operation to our project.py script that stores the volume in the job document:

```python
# project.py
# ...
@FlowProject.operation
@FlowProject.pre.after(compute_volume)
@FlowProject.post(lambda job: 'volume' in job.document)
def store_volume_in_document(job):
    with open(job.fn("volume.txt")) as textfile:
        job.document.volume = float(textfile.read())
```

Besides needing fewer lines of code, storing data in the job document has one more distinct advantage: it is directly searchable. That means that we can find and select jobs based on its content.

Executing the $ python project.py run command after adding the above function to the project.py script will store all volume in the job documents. We can then inspect all searchable data with the $ signac find command in combination with the --show option:

```bash
~/ideal_gas_project $ signac find --show
03585df0f87fada67bd0f540c102c7ce7
{'N': 1000, 'kT': 1.0, 'p': 3}
{'volume': 333.3333333333333}
22a51374466c4e01ef6e67e6f73c52e
{'N': 1000, 'kT': 1.0, 'p': 5}
{'volume': 200.0}
71855b321a04dd9ee27ce6c9cc0436f4
{'N': 1000, 'kT': 1.0, 'p': 4}
{'volume': 250.0}
# ...
```

When executed with --show, the find command not only prints the job id, but also the metadata and the document for each job. In addition to selecting by metadata as shown earlier, we can also find and select jobs by their job document content, e.g.:

```bash
~/ideal_gas_project $ signac find --doc-filter volume.$lte 125 --show
Interpreted filter arguments as '{"volume.$lte": 125}'.
df1794892c1ec0909e5955079754fb0b
{'N': 1000, 'kT': 1.0, 'p': 10}
{'volume': 100.0}
dbe8094b72da6b3dd7c8f17abac6b7608
{'N': 1000, 'kT': 1.0, 'p': 9}
{'volume': 111.1111111111111}
97ac0114bb2269561556b16ae0f30d43
{'N': 1000, 'kT': 1.0, 'p': 8}
{'volume': 125.0}
```
Job.data and Job.stores

The job data attribute provides a dict-like interface to an HDF5-file, which is designed to store large numerical data, such as numpy arrays.

For example:

```python
with job.data:
    job.data.my_array = numpy.zeros(64, 32)
```

You can use the data attribute to store both built-in types, numpy arrays, and pandas dataframes. The job.data property is a short-cut for job.stores['signac_data'], you can access many different data stores by providing your own name, e.g., job.stores.my_data.

See Job Data Storage for an in-depth discussion.

1.4.3 Job scripts and cluster submission

Generating scripts

So far, we executed all operations directly on the command line with the run command. However we can also generate scripts for execution, which is especially relevant if you intend to submit the workflow to a scheduling system typically encountered in high-performance computing (HPC) environments.

Scripts are generated using the jinja2 templating system, but you don’t have to worry about that unless you want to change any of the default templates.

We can generate a script for the execution of the next eligible operations with the script command. We need to reset our workflow before we can test that:

```bash
~/ideal_gas_project $ rm -r workspace/
~/ideal_gas_project $ python init.py
```

Let’s start by generating a script for the execution of up to two eligible operations:

```bash
~/ideal_gas_project $ python project.py script -n 2
set -e
set -u
cd /Users/csadorf/ideal_gas_project
# Operation 'compute_volume' for job '03585df0f87fada67bd0f540c102cce7':
python project.py exec compute_volume 03585df0f87fada67bd0f540c102cce7
# Operation 'compute_volume' for job '22a51374466c4e01ef0e67e5f73c52e':
python project.py exec compute_volume 22a51374466c4e01ef0e67e5f73c52e
```

By default, the generated script will change into the project root directory and then execute the command for each next eligible operation for all selected jobs. We then have two ways to run this script. One option would be to pipe it into a file and then execute it:

```bash
~/ideal_gas_project $ python project.py script > run.sh
~/ideal_gas_project $ /bin/bash run.sh
```

Alternatively, we could pipe it directly into the command processor:

```bash
~/ideal_gas_project $ python project.py script | /bin/bash
```
Executing the `script` command again, we see that it would now execute both the `store_volume_in_document` and the `store_volume_in_json_file` operation, since both share the same pre-conditions:

```bash
~/ideal_gas_project $ python project.py script -n 2
set -e
set -u
cd /Users/csadorf/ideal_gas_project

# Operation 'store_volume_in_document' for job '03585df0f87fada67bd0f540c102cce7':
python project.py exec store_volume_in_document 03585df0f87fada67bd0f540c102cce7

# Operation 'store_volume_in_json_file' for job '03585df0f87fada67bd0f540c102cce7':
python project.py exec store_volume_in_json_file 03585df0f87fada67bd0f540c102cce7
```

If we wanted to customize the script generation, we could either extend the base template or simply replace the default template with our own. To replace the default template, we can put a template script called `script.sh` into a directory called `templates` within the project root directory. A simple template script might look like this:

```bash
cd {{ project.config.project_dir }}
{% for operation in operations %}
{{ operation.cmd }}
{% endfor %}
```

Storing the above template within a file called `templates/script.sh` will now change the output of the `script` command to:

```bash
~/ideal_gas_project $ python project.py script -n 2
cd /Users/csadorf/ideal_gas_project

python project.py exec store_volume_in_document 03585df0f87fada67bd0f540c102cce7
python project.py exec store_volume_in_json_file 03585df0f87fada67bd0f540c102cce7
```

Please see `$ python project.py script --template-help` to get more information on how to write and use custom templates.

### Submit operations to a scheduling system

In addition to executing operations directly on the command line and generating scripts, `signac` can also submit operations to a scheduler such as SLURM. This is essentially equivalent to generating a script as described in the previous section, but in this case the script will also contain the relevant scheduler directives such as the number of processors to request. In addition, `signac` will also keep track of submitted operations in addition to workflow progress, which almost completely automates the submission process as well as preventing the accidental repeated submission of operations.

To use this feature, make sure that you are on a system with any of the supported schedulers and then run the `$ python project.py submit` command.

As an example, we could submit the operation `compute_volume` to the cluster.

```bash
$ python project.py submit -o compute_volume -n 1 -w 1.5
```

This command submits to the cluster the operation `compute_volume` for the next available 5 jobs, and each job is submitted with a walltime of 1.5 hours. We can use the `--pretend` option to output the text of the submission document. Here is some sample output used on Stampede2, a SLURM-based queuing system:
We can submit 5 jobs simultaneously by changing `-n 1` to `-n 5`. After submitting, if we run `$ python project.py status -d`, a detailed report is produced that tracks the progress of each job.

```
$ python project.py status -d
Query scheduler...
Collect job status info: 100%|| 10/10 [00:00<00:00, 2500.48it/s]
# Overview:
  Total # of jobs: 10

# Detailed View:
<table>
<thead>
<tr>
<th>job_id</th>
<th>operation</th>
<th>labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>ee550647e3f707b251eeb094f43d434c</td>
<td>compute_volume</td>
<td>[Q]</td>
</tr>
<tr>
<td>df1794892c1sec0909e5955079754f00b</td>
<td>compute_volume</td>
<td>[Q]</td>
</tr>
<tr>
<td>71855b32a04dd9ee27ce6c9cc0436f4</td>
<td>compute_volume</td>
<td>[Q]</td>
</tr>
<tr>
<td>dbe8094b72da6b3dd7c8f17abdc6b7608</td>
<td>compute_volume</td>
<td>[Q]</td>
</tr>
<tr>
<td>a2fa2b86d0da1df3f5dbaa3a7798a59</td>
<td>compute_volume</td>
<td>[Q]</td>
</tr>
<tr>
<td>22a51374466c4e0ef0e67e65f73c52e</td>
<td>compute_volume</td>
<td>[U]</td>
</tr>
<tr>
<td>97ac0114bb2269561556b16ae0f030d43</td>
<td>compute_volume</td>
<td>[U]</td>
</tr>
<tr>
<td>03585df0f87fa6a67bd0f540c102ce7</td>
<td>compute_volume</td>
<td>[U]</td>
</tr>
<tr>
<td>e5613a5439caeb21ce40a2f0ebe7ed</td>
<td>compute_volume</td>
<td>[U]</td>
</tr>
<tr>
<td>742c883cb8e8e417bb236d40a9a9543</td>
<td>compute_volume</td>
<td>[U]</td>
</tr>
</tbody>
</table>

[U]:unknown [R]:registered [Q]:queued [A]:active [I]:inactive [!]:requires_attention
```

Jobs signified with `Q` are queued in the cluster; when calling `python project.py status -d` again, if `signac` queries the cluster to find those jobs have begun running, their status will be reported `A`.

See the `Cluster Submission` section for further details on how to use the `submit` option and the `Manage Environments` section for details on submitting to your particular cluster.
1.5 Topic Guides

A complete reference to all major components of the **signac** framework.

This section is primarily targeted at users who want to learn in-depth about the individual components. New users should go through the *Tutorial* first.

1.5.1 Managing Data

Projects

Introduction

For a full reference of the Project API, please see the Python API.

A **signac** project is a conceptual entity consisting of three components:

1. a **data space**,  
2. **scripts and routines** that operate on that space, and  
3. the project’s **documentation**.

This division corresponds largely to the definition of a computational project outlined by Wilson et al. The primary function of **signac** is to provide a single interface between component (2), the scripts encapsulating the project logic, and component (1), the underlying data generated and manipulated by these operations. By maintaining a clearly defined data space that can be easily indexed, **signac** can provide a consistent, homogeneous data access mechanism. In the process, **signac**’s maintenance of the data space also effectively functions as an implicit part of component (3), the project’s documentation.

Project Initialization

In order to use **signac** to manage a project’s data, the project must be **initialized** as a **signac** project. After a project has been initialized in **signac**, all shell and Python scripts executed within or below the project’s root directory have access to **signac**’s central facility, the **signac** project interface. The project interface provides simple and consistent access to the project’s underlying **data space**.

To initialize a project, simply execute $ **signac** init <project-name> on the command line inside the desired project directory (create a new project directory if needed). For example, to initialize a **signac** project named *MyProject* in a directory called *my_project*, execute:

```
$ mkdir my_project  
$ cd my_project  
$ signac init MyProject
```

You can alternatively initialize your project within Python with the **init_project()** function:

```
>>> project = signac.init_project('MyProject')
```

This will create a configuration file which contains the name of the project. The directory that contains this configuration file is the project’s root directory.

---

1 You can access a project interface from other locations by explicitly specifying the root directory.
The Data Space

The project data space is stored in the *workspace directory*. By default this is a sub-directory within the project’s root directory named `workspace`. Once a project has been initialized, any data inserted into the data space will be stored within this directory. This association is not permanent; a project can be reassociated with a new workspace at any time, and it may at times be beneficial to maintain multiple separate workspaces for a single project. You can access your signac `Project` and the associated *data space* from within your project’s root directory or any subdirectory from the command line:

```
$ signac project
MyProject
```

Or with the `get_project()` function:

```python
>>> import signac
>>> project = signac.get_project()
>>> print(project)
MyProject
```

Jobs

The central assumption of the *signac* data model is that the *data space* is divisible into individual data points, consisting of data and metadata, which are uniquely addressable in some manner. Specifically, the workspace is divided into sub-directories, where each directory corresponds to exactly one *Job*. Each job has a unique address, which is referred to as a *state point*. A job can consist of any type of data, ranging from a single value to multiple terabytes of simulation data; *signac*’s only requirement is that this data can be encoded in a file.

A job is essentially just a directory on the file system, which is part of a *project workspace*. That directory is called the *job workspace* and contains all data associated with that particular job.

You access a job by providing a *state point*, which is a unique key-value mapping describing your data. All data associated with your job should be a unique function of the *state point*, e.g., the parameters that go into your physics or machine learning model. For example, to store data associated with particular temperature or pressure of a simulation, you would first initialize a project, and then *open* a job like this:

```python
project = get_project('path/to/my_project')
job = project.open_job({'temperature': 20, 'pressure': 1.0})
job.init()
with open(job.fn('results.txt')) as file:
    ...
```

Tip: You only need to call the `Job.init()` function the first time that you are accessing a job. Furthermore, the `Job.init()` function returns itself, so you can abbreviate like this:
The job state point represents a unique address of your data within one project. There can never be two jobs that share the same state point within the same project. Any other kind of data and metadata that describe your job, but do not represent a unique address should be stored within the Job.doc, which has the exact same interface like the Job.sp, but does not represent a unique address of the job.

Tip: The Job interface and the various methods of storing data are described in detail in the Jobs section.

In addition to obtaining a job handle via the project open_job() function, you can also access it directly with the signac.get_job() function. For example, you can get a handle on a job by switching into the workspace directory and then calling signac.get_job():

```python
>>> import signac
>>> job = signac.get_job()
>>> print(job)
42b7b4f2921788ea14dac556e6f06d0
```

Finding jobs

In general, you can iterate over all initialized jobs using the following idiom:

```python
for job in project:
    pass
```

This notation is shorthand for the following snippet of code using the Project.find_jobs() method:

```python
for job in project.find_jobs():
    pass
```

However, the find_jobs() interface is much more powerful in that it allows filtering for subsets of jobs. For example, to iterate over all jobs that have a state point parameter b=0, execute:

```python
for job in project.find_jobs({'b': 0}):
    pass
```

For more information on how to search for specific jobs in Python and on the command line, please see the Query API chapter.

Grouping

Grouping operations can be performed on the complete project data space or the results of search queries, enabling aggregated analysis of multiple jobs and state points.

The return value of the find_jobs() method is a cursor that we can use to iterate over all jobs (or all jobs matching an optional filter if one is specified). This cursor is an instance of JobsCursor and allows us to group these jobs by state point parameters, the job document values, or even arbitrary functions.

Note: The groupby() method is very similar to Python's built-in itertools.groupby() function.
Basic Grouping by Key

Grouping can be quickly performed using a statepoint or job document key.

If $a$ was a state point variable in a project’s parameter space, we can quickly enumerate the groups corresponding to each value of $a$ like this:

```python
for a, group in project.groupby('a):
    print(a, list(group))
```

Similarly, we can group by values in the job document as well. Here, we group all jobs in the project by a job document key $b$:

```python
for b, group in project.groupbydoc('b'):
    print(b, list(group))
```

Grouping by Multiple Keys

Grouping by multiple state point parameters or job document values is possible, by passing an iterable of fields that should be used for grouping. For example, we can group jobs by state point parameters $c$ and $d$:

```python
for (c, d), group in project.groupby(('c', 'd')):
    print(c, d, list(group))
```

Searching and Grouping

We can group a data subspace by combining a search with a group-by function. As an example, we can first select all jobs, where the state point key $e$ is equal to 1 and then group them by the state point parameter $f$:

```python
for f, group in project.find_jobs({"e": 1}).groupby('f'):
    print(f, list(group))
```

Custom Grouping Functions

We can group jobs by essentially arbitrary functions. For this, we define a function that expects one argument and then pass it into the `groupby()` method. Here is an example using an anonymous `lambda` function as the grouping function:

```python
for (d, count), group in project.groupby(
    lambda job: (job.sp['d'], job.document['count ->'])):
    print(d, count, list(group))
```

Moving, Copying and Removal

In some cases it may desirable to divide or merge a project data space. To move a job to a different project, use the `move()` method:

```python
other_project = get_project(root='/path/to/other_project')
for job in jobs_to_move:
    job.move(other_project)
```
Copy a job from a different project with the `clone()` method:

```python
project = get_project()
for job in jobs_to_copy:
    project.clone(job)
```

Trying to move or copy a job to a project which has already an initialized job with the same `state point`, will trigger a `DestinationExistsError`.

**Warning:** While **moving** is a cheap renaming operation, **copying** may be much more expensive since all of the job’s data will be copied from one workspace into the other.

To **clear** all data associated with a specific job, call the `clear()` method. Note that this function will do nothing if the job is uninitialized; the `reset()` method will also clear all data associated with a job, but it will also automatically initialize the job if it was not originally initialized. To **permanently delete** a job and its contents use the `remove()` method:

```python
job = project.open_job(statepoint)
job.remove()
assert job not in project
```

## Centralized Project Data

To support the centralization of project-level data, **signac** offers simple facilities for placing data at the project level instead of associating it with a specific job. For one, **signac** provides a **project document** and **project data** analogous to the **job document** and **job data**. The project document is stored in JSON format in the project root directory and can be used to store similar types of data to the job document.

```python
>>> project = signac.get_project()
>>> project.doc['hello'] = 'world'
>>> print(project.doc().get('hello'))
'world'
>>> print(project.doc.hello)
'world'
```

The project data is stored in HDF5 format in the project root directory and can be used to store similar types of data to the job data.

```python
>>> project = signac.get_project()
>>> with project.data:
...     project.data['hello'] = 'world'
>>> with project.data:
...     print(project.data.get('hello'))
'world'
>>> with project.data:
...     print(project.data.hello)
'world'
```

In addition, **signac** also provides the `signac.Project.fn()` method, which is analogous to the `Job.fn()` method described above:
>>> print(project.root_directory())
'/home/johndoe/my_project/'

>>> print(project.fn('foo.bar'))
'/home/johndoe/my_project/foo.bar'

Schema Detection

While `signac` does not require you to specify an *explicit* state point schema, it is always possible to deduce an *implicit* semi-structured schema from a project’s data space. This schema is comprised of the set of all keys present in all state points, as well as the range of values that these keys are associated with.

Assuming that we initialize our data space with two state point keys, `a` and `b`, where `a` is associated with some set of numbers and `b` contains a boolean value:

```python
for a in range(3):
    for b in (True, False):
        project.open_job({'a': a, 'b': b}).init()
```

Then we can use the `detect_schema()` method to get a basic summary of keys within the project’s data space and their respective range:

```python
>>> print(project.detect_schema())
{
    'a': 'int([0, 1, 2], 3)',
    'b': 'bool([False, True], 2)',
}
```

This functionality is also available directly from the command line:

```
$ signac schema
{
    'a': 'int([0, 1, 2], 3)',
    'b': 'bool([False, True], 2)',
}
```

Importing and Exporting Data

Data archival is important to preserving the integrity, utility, and shareability of a project. To this end, `signac` provides interfaces for importing workspaces from and exporting workspaces to directories, zip-files, and tarballs. The exported project archives are useful for publishing data, e.g., for researchers wishing to make an original data set available alongside a publication.

Exporting a Workspace

Exporting a project could be as simple as zipping the project files and workspace paths ($zip -r project_archive.zip /data/my_project/). The functionality provided by `signac export` is a bit more fine-grained and allows the use of a custom path structure or the export of a subset of the jobs based on state point or document filters or by job id.

For example, suppose we have a project stored locally in the path `/data/my_project` and want to export it to a directory `/data/my_project_archive`. The project’s jobs are assumed to have state point keys `a` and `b` with integer values. We would first change into the root directory of the project that we want to export and then call `signac export` with the target path:
signac Documentation

```shell
$ cd /data/my_project
$ signac export /data/my_project_archive
```

This would **copy** data from the source project to the export directory with the following directory structure:

```
/data/my_project_archive/a/0/b/0/
/data/my_project_archive/a/0/b/1/
/data/my_project_archive/a/0/b/2/
# etc.
```

The default path function is based on the implicit schema of all exported jobs, but we can also **optionally** specify a specific export path, for example like this:

```
$ signac export /data/my_project_archive "a_{a}/b_{b}" 
```

It is possible to directly export to a zip-file or tarball by simply providing the path to the archive-file as target (*e.g.* `$ signac export /data/my_project_archive.zip`). For more details on how to use `signac export`, type `$ signac export --help` or see the documentation for the `export_to()` method.

### Importing a Data Space

The import of data spaces into a **signac** workspace means to map all directories as part of an arbitrary directory structure to signac job state points. That is easiest when one imports a previously exported workspace, which will still contain all state point files.

For example, we could first export our workspace in `~/my_project` to `~/data/` with

```
~/my_project $ signac export ~/data/
```

and then import the exported data into a second project:

```
~/my_new_project $ signac import ~/data/
```

Since the imported data space was previously exported with **signac**, all state point metadata is automatically determined from the state point manifest files.

In the case that we want to import a data space that was not previously exported with **signac**, we need to provide a schema-function. In the simplest case, that is just a function based on the data space paths, *e.g.*

```
$ signac import /data/non_signac_archive "a_{a:int}/b_{b:int}" 
```

The command above will copy all data from the the `/data/non_signac_archive` directory and use the paths of sub-directories to identify the associated state points. For example, the path `a_0/b_1` will be interpreted as `{ 'a': 0, 'b': 1 }`. The type specification — here `int` for both `a` and `b` — is optional and means that these values are converted to `int`; the default type is `str`.

Importing from zip-files and tarballs works similarly, by specifying that path as the origin. For more details on how to use `signac import`, type `$ signac import --help` or see the documentation for `import_from()`.

### Linked Views

Data space organization by job id is both efficient and flexible, but the obfuscation introduced by the job id makes inspecting the workspace on the command line or via a file browser much harder. A **linked view** is a directory hierarchy with human-interpretable names that link to the actual job workspace directories. Unlike the default mode for **data export**, no data is copied for the generation of linked views.
To create views from the command line use the $ signac view command.

**Important:** When the project data space is changed by adding or removing jobs, simply update the view, by executing `create_linked_view()` or `$ signac view` for the same view directory again.

You can limit the linked view to a specific data subset by providing a set of job ids to the `create_linked_view()` method. This works similar for `$ signac view` on the command line, but here you can also specify a filter directly:

```
$ signac view -f a 0
```

will create a linked view for all jobs, where a=0.

### Synchronization

In some cases it may be necessary to store a project at more than one location, perhaps for backup purposes or for remote execution of data space operations. In this case there will be a regular need to synchronize these data spaces.

Synchronization of two projects can be accomplished by either using `rsync` to directly synchronize the respective workspace directories, or by using `signac sync`, a tool designed for more fine-grained synchronization of project data spaces. Users who are familiar with `rsync` will recognize that most of the core functionality and API of `rsync` is replicated in `signac sync`.

As an example, let’s assume that we have a project stored locally in the path `/data/my_project` and want to synchronize it with `/remote/my_project`. We would first change into the root directory of the project that we want to synchronize data into. Then we would call `signac sync` with the path of the project that we want to synchronize with:

```
$ cd /data/my_project
$ signac sync /remote/my_project
```

This would copy data from the remote project to the local project. For more details on how to use `signac sync`, type `$ signac sync --help`.

Projects can also be synchronized using the Python API:

```
project.sync('remote/my_project')
```

### Jobs

#### Overview

A job is a directory on the file system, which is part of a project workspace. That directory is called the job workspace and contains all data associated with that particular job. Every job has a unique address called the state point.

There are two ways to associated metadata with your job:

1. As part of the `Job.statepoint` (aliased by `Job.sp`).
2. As part of the `Job.document` (aliased by `Job.doc`).

Both containers have the exact same (dict-like) interface and capabilities, both are indexed (that means searchable), but only the former represents the unique address of the job. In other words, all data associated with a particular job should be a direct or indirect function of the state point.
Important: Every parameter that, when changed, would invalidate the job’s data, should be part of the state point; all others should not.

However, you only have to add those parameters that are actually changed (or anticipated to be changed) to the state point. It is perfectly acceptable to hard-code parameters up until the point where you actually change them, at which point you would add them to the state point retroactively.

You can, but do not have to use the Job interface to associate data with a job. Any file—with a name and format of your choosing— that is stored within the job’s workspace directory is considered data associated with the job.

However, if you do choose to interact with the data through the Job interface, there are four main ways of doing so:

1. You temporarily change the working directory to the job’s workspace directory by using the job as a context manager (with job:).
2. You use the Job.fn() function to construct file paths, where job.fn('data.txt') is equivalent to os.path.join(job.workspace(), 'data.txt').
4. You store (small or large) numerical array-like data in the Job.data container.

All three data containers Job.statepoint, Job.document, and Job.data provide a highly similar dict-like interface, which is described in more detail in the following sections.

The Job State Point

A state point is a simple mapping of key-value pairs containing metadata describing the job. The state point is then used to compute a hash value, called the job id, which serves as the unique id for the job. The signac framework keeps track of all data and metadata by associating each job with a workspace directory, which is just a subdirectory of the project workspace. This subdirectory is named by the job id, therefore guaranteeing a unique file system path for each job within the project’s workspace directory.

Note: Because signac assumes that the state point is a unique identifier, multiple jobs cannot share the same state point. A typical remedy for scenarios where, e.g., multiple replicas are required, is to append the replica number to the state point to generate a unique state point.

Both the state point and the job id are equivalent addresses for jobs in the data space. To access or modify a data point, obtain an instance of Job by passing the associated metadata as a mapping of key-value pairs (for example, as an instance of dict) into the open_job() method.

```python
# Define a state point:
>>> statepoint = {'a': 0}
# Get the associated job:
>>> job = project.open_job(statepoint)
>>> print(job.get_id())
9bfd29df07674bc4aa960cf661b5acd2
```

In general an instance of Job only gives you a handle to a python object. To create the underlying workspace directory and thus make the job part of the data space, you must initialize it. You can initialize a job explicitly, by calling the Job.init() method, or implicitly, by either accessing the job’s job document or by switching into the job’s workspace directory.
>>> job = project.open_job({'a': 2})
# Job does not exist yet
>>> job in project
False
>>> job.init()
# Job now exists
>>> job in project
True

Once a job has been initialized, it may also be opened by id as follows (initialization is required because prior to initialization the job id has not yet been calculated):

>>> job.init()
>>> job2 = project.open_job(id=job.get_id())
>>> job == job2
True

Whether a job is opened by state point or job id, an instance of Job can always be used to retrieve the associated state point, the job id, and the workspace directory with the Job.statepoint, Job.get_id(), and Job.workspace() methods, respectively:

>>> print(job.statepoint())
{'a': 0}
>>> print(job.get_id())
9bfd29df07674bc4aa960cf661b5acd2
>>> print(job.workspace())
'/home/johndoe/my_project/workspace/9bfd29df07674bc4aa960cf661b5acd2'

Evidently, the job’s workspace directory is a subdirectory of the project’s workspace and is named by the job’s id. We can use the Job.fn() function to prepend the workspace path to a file name; job.fn(filename) is equivalent to os.path.join(job.workspace(), filename). This function makes it easy to create or open files which are associated with the job:

>>> print(job.fn('newfile.txt'))
'/home/johndoe/my_project/workspace/9bfd29df07674bc4aa960cf661b5acd2/newfile.txt'

For convenience, the state point may also be accessed via the Job.statepoint or Job.sp attributes, e.g., the value for a can be printed using either print(job.sp.a) or print(job.statepoint.a). This also works for nested state points: print(job.sp.b.c)!

Modifying the State Point

As just mentioned, the state point of a job can be changed after initialization. A typical example where this may be necessary, is to add previously not needed state point keys. Modifying a state point entails modifying the job id which means that the state point file needs to be rewritten and the job’s workspace directory is renamed, both of which are computationally cheap operations. The user is nevertheless advised to take great care when modifying a job’s state point since errors may render the data space inconsistent.

There are three main options for modifying a job’s state point:

1. Directly via the job’s Job.statepoint and Job.sp attributes,
2. via the job’s Job.update_statepoint() method, and
3. via the job’s Job.reset_statepoint() method.
The `Job.update_statepoint()` method provides safeguards against accidental overwriting of existing state point values, while `Job.reset_statepoint()` will simply reset the whole state point without further questions. The `Job.statepoint` and `Job.sp` attributes provide the greatest flexibility, but similar to `Job.reset_statepoint()` they provide no additional protection.

**Important:** Regardless of method, signac will always raise a `DestinationExistsError` if a state point modification would result in the overwriting of an existing job.

The following examples demonstrate how to add, rename and delete state point keys using the `Job.sp` attribute:

To **add a new key** `b` to all existing state points that do not currently contain this key, execute:

```python
for job in project:
    job.sp.setdefault('b', 0)
```

Renaming a state point key from `b` to `c`:

```python
for job in project:
    assert 'c' not in job.sp
    job.sp.c = job.statepoint.pop('b')
```

To **remove** a state point key `c`:

```python
for job in project:
    if 'c' in job.sp:
        del job.sp['c']
```

You can modify nested state points in-place, but you will need to use dictionaries to add new nested keys, e.g.:

```python
>>> job.statepoint()
{'a': 0}
>>> job.sp.b.c = 0  # <-- will raise an AttributeError!!

# Instead:
>>> job.sp.b = {'c': 0}

# Now you can modify in-place:
>>> job.sp.b.c = 1
```

**Warning:** The statepoint object behaves like a dictionary in most cases, but because it persists changes to the filesystem, making a copy requires explicitly converting it to a dict. If you need a modifiable copy that will not modify the underlying JSON file, you can access a dict copy of the statepoint by calling it, e.g. `sp_dict = job.statepoint()` instead of `sp = job.statepoint`. For more information, see `JSONDict`.

### The Job Document

In addition to the state point, additional metadata can be associated with your job in the form of simple key-value pairs using the `job document`. This `job document` is automatically stored in the job’s workspace directory in JSON format. You can access it via the `Job.document` or the `Job.doc` attribute.

```python
>>> job = project.open_job(statepoint)
>>> job.doc['hello'] = 'world'
```
Just like the job state point, individual keys may be accessed either as attributes or through a functional interface, e.g.:

```python
>>> print(job.document().get('hello'))
world
>>> print(job.document.hello)
world
>>> print(job.doc.hello)
world
```

**Tip:** Use the Job.document.get() method to return None or another specified default value for missing values. This works exactly like with python’s built-in dictionaries.

Use cases for the job document include, but are not limited to:

1. **storage of lightweight data,**
2. Tracking of runtime information
3. **labeling** of jobs, e.g. to identify error states.

**Tip:** Large arrays of numerical data are often not conducive to store in the Job.document. Text-based formats like JSON can be slower and less precise for floating-point numbers. For this kind of data, consider using the Job.data container.

## Job Data Storage

### Basics

Large numerical or text data can be stored in the Job.data container, which is an instance of `signac.H5Store`. This container uses a file in HDF5 format to store array-like or dictionary-like information. Like the Job.document, this information can be accessed using key-value pairs. Unlike the Job.document, Job.data is not searchable.

An example of storing data:

```python
>>> import numpy as np
>>> job = project.open_job(statepoint)
>>> job.data['x'] = np.arange(0, 1, 0.01)
>>> job.data['hello'] = 'world'
```

Just like the job state point and document, individual keys may be accessed either as attributes or through a functional interface, e.g.:. The following examples are all equivalent:

```python
>>> print(job.data.get('hello'))
world
>>> print(job.data['hello'])
world
>>> print(job.data.hello)
world
```
**Tip:** Use the `Job.data.get()` method to return `None` or another specified default value for missing values. This works exactly like with Python’s built-in dictionaries.

### File handling

The underlying HDF5 file is openend and flushed after each read- and write-operation. You can keep the file explicitly open using a context manager. The file is only opened and flushed once in the following example:

```python
>>> with job.data:
...     job.data['hello'] = 'world'
...     print(job.data.x)
```

The default open-mode is append ("a"), but you can override the open-mode, by using the `signac.H5Store.open()` function explicitly. For example, to open the store in read-only mode, you would write:

```python
>>> with job.data.open(mode='r'):
...     print(job.data.x)
```

Explicitly opening the underlying file by either using the context manager or the open() function is required when reading and writing arrays, such as `numpy.arrays`. Please see the next section for details on accessing arrays.

**Warning:** It is strongly advised that operations on `Job.data` are not performed in parallel, to avoid data corruption.

### Accessing arrays

All values stored in `job.data` are returned as copies, except for arrays, which are accessed by reference and not automatically copied into memory. That is important to enable the storage of massive arrays that do not necessarily fit into memory.

However, you can always create an explicit memory copy using the copy-operator `[]`:

```python
>>> with job.data:
...     x = job.data.x[()]  
```

### Low-level API

The `H5Store` class that provides the interface for `Job.data` implements a dict-like interface to provide a homogeneous interface between `Job.statepoint`, `Job.document`, and `Job.data`. However, in some cases it may be desirable to use more advanced functions provided by the h5py library itself, which we consider low-level API in this context.

The low-level API is exposed as the `file` property, which is accessible whenever the store is open. For example, this is how we could use that to explicitly create an array:

```python
>>> with job.data:
...     dset = job.data.file.create_dataset("X", (64, 32), dtype='f4')
```
Note: The file must be open to access the file property!

Please see the h5py documentation for more information on how to interact with h5py.File objects.

Job Stores

As mentioned before, the Job.data property represents an instance of H5Store, specifically one that operates on a file called signac_data.h5 within the job workspace directory. However, there are some reasons why one would want to operate on multiple different HDF5 files instead of only one.

1. While the HDF5-format is generally mutable, it is fundamentally designed to be used as an immutable data container. It is therefore advantageous to write large arrays to a new file instead of modifying an existing file many times.

2. It easier to synchronize multiple files instead of just one.

3. Multiple operations executed in parallel can operate on different files circumventing file locking issues.

The Job.stores property provides a dict-like interface to access multiple different HDF5 files within the job workspace directory. In fact, the Job.data container is essentially just an alias for job.stores.signac_data.

For example, to store an array X within a file called my_data.h5, one could use the following approach:

```python
with job.stores.my_data as data:
    data['X'] = X
```

The Job.stores attribute is an instance of signac.H5StoreManager and implements a dict-like interface.

Query API

As briefly described in Finding jobs, the find_jobs() method provides much more powerful search functionality beyond simple selection of jobs with specific state point values. More generally, all find() functions within the framework accept filter arguments that will return a selection of jobs or documents. One of the key features of signac is the possibility to immediately search managed data spaces to select desired subsets as needed. Internally, all search operations are processed by an instance of Collection (see Collections). Therefore, they all follow the same syntax, so you can use the same type of filter arguments in find_jobs(), find_statepoints(), and so on.

Note: The signac framework query API is a subset of the MongoDB query API!

Basic Expressions

Filter arguments are a mapping of expressions, where a single expression consists of a key-value pair. All selected documents must match these expressions.

The simplest expression is an exact match. For example, in order to select all jobs whose state point key a has the value 42, you would use the following expression: {'a': 42} as follows:

```python
project.find_jobs({'a': 42})
```
Select All

If you want to select the complete data set, don’t provide any filter argument at all. The default argument of `None` or an empty expression `{}` will select all jobs or documents. As was previously demonstrated, iterating over all jobs in a project or all documents in a collection can be accomplished directly without using any `find` method at all:

```python
for job in project:
    pass
for doc in collection:
    pass
```

Simple Selection

To select documents by one or more specific key-value pairs, simply provide these directly as filter arguments. For example, assuming that we have a list of documents with values $N$, $kT$, and $p$, as such:

```
1: {'N': 1000, 'kT': 1.0, 'p': 1}
2: {'N': 1000, 'kT': 1.2, 'p': 2}
3: {'N': 1000, 'kT': 1.3, 'p': 3}
...```

We can select the 2nd document with `{ 'p': 2 }`, but also `{ 'N': 1000, 'p': 2 }` or any other matching combination.

Nested Keys

To match nested keys, avoid nesting the filter arguments, but instead use the `-`-operator. For example, if the documents shown in the example above were all nested like this:

```
1: {'statepoint': {'N': 1000, 'kT': 1.0, 'p': 1}}
2: {'statepoint': {'N': 1000, 'kT': 1.2, 'p': 2}}
3: {'statepoint': {'N': 1000, 'kT': 1.3, 'p': 3}}
...```

Then we would use `{ 'statepoint.p': 2 }` instead of `{ 'statepoint': { 'p': 2 } } as filter argument. This is not only easier to read, but also increases compatibility with MongoDB database systems.

Operator Expressions

In addition to simple exact value matching, signac also provides `operator-expressions` to execute more complicated search queries.

Arithmetic Expressions

If we wanted to match all documents where $p$ is greater than 2, we would use the following filter argument:

```
{ 'p': { '$gt': 2 } }
```

Note that we have replaced the value for $p$ with the expression `{ '$gt': 2 }` to select all all jobs with $p$ values greater than 2. Here is a complete list of all available `arithmetic operators`:  

```
• $eq$: equal to
• $ne$: not equal to
• $gt$: greater than
• $gte$: greater or equal than
• $lt$: less than
• $lte$: less or equal than

Near Operator

The $near operator is used to find jobs with state point parameters that are near a value, where floating point precision may make it difficult to match the exact value. The behavior of $near matches that of python’s math.isclose function. The “reference” value and tolerances are passed in as a list in the order [reference, [relative_tolerance, [absolute_tolerance]]], where the inner []s denote optional values. Note that default values are relative_tolerance = 1e-09 and absolute_tolerance = 0.

```
signac find theta.<\$near 0.6  # easier than typing 0.600000001
signac find '{'p.$near': [100, 0.05]}'  # p within 5% of 100
signac find '{'p.$near': [100, 0.05, 2]}'  # abs(p-100)/max(p, 100) < 0.05 or abs(p-100) < 2
```

Logical Operators

There are two supported logical operators: $and and $or. To querying with a logical expression, we construct a mapping with the logical-operator as the key and a list of expressions as the value. As usual, the $and operator matches documents where all the expressions are true, while the $or expression matches if any documents satisfy the provided expression. For example, we can match all documents where $p$ is greater than 2 or $kT=1.0$ we could use the following (split onto multiple lines for clarity):

```
{
  '$or': [
    {'p': {'$gt': 2}},       # either match this
    {'kT': 1.0}              # or this
  ]
}
```

Logical expressions may be nested, but cannot be the value of a key-value expression.

Exists Operator

If you want to check for the existence of a specific key but do not care about its actual value, use the $exists-operator. For example, the expression '{'p': {'$exists': True}} will return all documents that have a key p regardless of its value. Likewise, using False as argument would return all documents that have no key with the given name.

Array Operator

This operator may be used to determine whether specific keys have values, that are in ($in), or not in ($nin) a given array, e.g.:
{'p': {'$in': [1, 2, 3]}}

This would return all documents where the value for \( p \) is either 1, 2, or 3. The usage of \$\text{nin} \) is equivalent, and will return all documents where the value is \textit{not in} the given array.

**Regular Expression Operator**

This operator may be used to search for documents where the value of type \texttt{str} matches a given \textit{regular expression}. For example, to match all documents where the value for \textit{protocol} contains the string “assembly”, we could use:

\[
\{'\text{protocol}': \{'\$\text{regex}': 'assembly'\}\}
\]

This operator internally applies the \texttt{re.search()} function and will never match if the value is not of type \texttt{str}.

To negate a regular expression use a negative lookahead, \textit{e.g.}, to match all state points where the protocol does \texttt{not} contain the word “assembly”, you would use:

\[
\{'\text{protocol}': \{'\$\text{regex}': r'^!(?!.*assembly).*$'\}\}
\]

**Tip:** Use the Regex101 app to develop and test your regular expressions.

**Type Operator**

This operator may be used to search for documents where the value is of a specific type. For example, to match all documents, where the value of the key \textit{N} is of integer-type, we would use:

\[
\{'\text{N}': \{'\$\text{type}': \texttt{int}'\}\}
\]

Other supported types include \texttt{float}, \texttt{str}, \texttt{bool}, \texttt{list}, and \texttt{null}.

**Where Operator**

This operator allows us to apply a \textit{custom function} to each value and select based on its return value. For example, instead of using the \texttt{regex-operator}, as shown above, we could write the following expression:

\[
\{'\text{protocol}': \{'\$\text{where}': 'lambda x: "assembly" in x'\}\}
\]

**Simplified Syntax on the Command Line**

It is possible to use search expressions directly on the command line, for example in combination with the \$ signac find command. In this case filter arguments are expected to be provided as valid JSON expressions. However, for simple filters you can also use a \textit{simplified syntax}. For example, instead of \{'p': 2\}, you can simply type \texttt{p 2}.

A simplified expression consists of key-value pairs in alternation. The first argument will then be interpreted as the first key, the second argument as the first value, the third argument as the second key, and so on. If you provide an odd number of arguments, the last value will default to \{'$\text{exists}': \texttt{True}'\}. Querying via operator is supported using the \texttt{.-operator}. Finally, you can use /<regex>/ instead of \{'$\text{regex}': '\<regex>'\} for regular expressions.

The following list shows simplified expressions on the left and their equivalent standard expression on the right.
The Dashboard

This chapter describes how to create a dashboard to quickly visualize data stored in a `signac` data space. To install the `signac-dashboard` package, see Installation.

Getting Started

You can start a dashboard to visualize `signac` project data in the browser, by importing the `Dashboard` class and calling its `main()` method.

```python
from signac_dashboard import Dashboard
Dashboard().main()
```

Start a Dashboard

The code below will open a dashboard for an newly-initialized (empty) project, with no jobs and one module loaded. Write the file `dashboard.py` with these contents:

```python
from signac_dashboard import Dashboard
from signac_dashboard.modules import ImageViewer

if __name__ == '__main__':
    Dashboard(modules=[ImageViewer()]).main()
```

Then launch the dashboard with `python dashboard.py run`.

Included Modules

For a list of available modules and usage instructions, see Dashboard Modules.

Specifying a custom job title

By creating a class that inherits from `Dashboard` (which we’ll call `MyDashboard`), we can begin to customize some of the functions that make up the dashboard, like `job_title()`, which gives a human-readable title to each job.
Running dashboards on a remote host

To use dashboards hosted by a remote computer, open an SSH tunnel to the remote computer and forward the port where the dashboard is hosted. For example, connect to the remote computer with

```
ssh username@remote.server.org -L 8890:localhost:8888
```

to forward port 8888 on the host to port 8890 on your local computer.

Dissecting the Dashboard Structure

- **Jobs** are how *signac* manages data. Each job has a statepoint (which contains job metadata) and a document (for persistent storage of key-value pairs). Jobs can be displayed in list view or grid view. The list view provides quick descriptions and status information from many jobs, while the grid view is intended to show text and media content from one or more jobs.

- **Templates** provide the HTML structure of the dashboard’s pages, written in Jinja template syntax for rendering content on the server.

- **Modules** are server-side Python code that interface with your *signac* data to display content. Generally, a module will render content from a specific job into a card template.

- **Cards** are a type of template that is shown in grid view and contains content rendered by a module.

Searching jobs

The search bar accepts JSON-formatted queries in the same way as the *signac* find command-line tool. For example, using the query "{{"key": "value"}}" will return all jobs where the job statepoint key is set to value. To search jobs by their document key-value pairs, use `doc:` before the JSON-formatted query, like `doc:{{"key": "value"}}`.

1.5.2 Implementing Workflows

Operations and Conditions

This chapter introduces the two **fundamental concepts** for the implementation of workflows with the *signac-flow* package: Data Space Operations and Conditions.

Data Space Operations
Concept

It is highly recommended to divide individual modifications of your project’s data space into distinct functions. In this context, a **data space operation** is defined as a unary function with an instance of Job as its only argument.

We will demonstrate this concept with a simple example. Let’s initialize a project with a few jobs, by executing the following script within a ~/my_project directory:

```python
# init.py
import signac

project = signac.init_project('MyProject')
for i in range(10):
    project.open_job({'a': i}).init()
```

A very simple operation, which creates a file called hello.txt within a job’s workspace directory could then be implemented like this:

```python
# operations.py
def hello(job):
    print('hello', job)
    with job:
        with open('hello.txt', 'w') as file:
            file.write('world!
')
```

We could execute this operation for the complete data space, for example in the following manner:

```bash
>>> import signac
>>> from operations import hello
>>> project = signac.get_project()
>>> for job in project:
>>>     hello(job)
... hello 0d32543f785d3459f27b8746f2053824
hello 14fb5d016557165019abaac200785048
hello 2af7905ebe91ada597a8d4bb91a1c0fc
>>>```

**The flow.run()-interface**

However, we can do better. The flow package provides a command line interface for modules that contain operations. We can access this interface by calling the run() function:

**The run()-interface**

The run() function parses the module from where it was called and interprets all top-level unary functions as operations.

```python
# operations.py
def hello(job):
    print('hello', job)
```

(continues on next page)
with job:
    with open('hello.txt', 'w') as f:
        file.write('world!\n')

if __name__ == '__main__':
    import flow
    flow.run()

Since the hello() function is a public, top-level function within the module with only one argument, it is interpreted as a dataspace-operation. That means we can execute it directly from the command line:

    ~/my_project $ python operations.py hello
    hello 0d32543f785d3459f27b8746f2053824
    hello 14fb5d016557165019abaac200785048
    hello 2af7905ebf59798d4bb91a1c0fc

This is a brief demonstration on how to implement the operations.py module:

**Parallelized Execution**

The run() function automatically executes all operations in parallel on as a many processors as there are available. We can test that by adding a “cost-function” to our example operation:

    from time import sleep
    def hello(job):
        sleep(1)
        # ...

Executing this with $ python operations.py hello we can now see how many operations are executed in parallel:

**Conditions**

In the context of signac-flow, a workflow is defined by the **ordered** execution of operations. The execution order is determined by specific **conditions**.

That means in order to implement a workflow, we need to determine two things:

1. What is the **current state** of the data space?
2. What needs to happen **next**?

We answer the first question by evaluating unary condition functions for each job. Based on those conditions, we can then determine what should happen next.

Following the example from above, we define a greeted condition that determines whether the hello() operation was executed, e.g. the hello.txt file exists:

    def greeted(job):
        return job.isfile('hello.txt')

Executing this workflow in an ad-hoc manner could be accomplished like this:
This approach is fine for simple workflows, but would become very cumbersome for even slightly more complex workflows and is not very flexible. In the next chapter, we will demonstrate how to integrate operations and conditions into a well-defined workflow using the `FlowProject` class.

### The FlowProject

This chapter describes how to setup a complete workflow via the implementation of a `FlowProject`.

### Setup and Interface

To implement a more automated workflow, we can subclass a `FlowProject`:

```python
# project.py
from flow import FlowProject

class Project(FlowProject):
    pass

if __name__ == '__main__':
    Project().main()
```

**Tip:** You can generate boiler-plate templates like the one above with the `$ flow init` function. There are multiple different templates available via the `-t/--template` option.

Executing this script on the command line will give us access to this project's specific command line interface:

```bash
~/my_project $ python project.py
usage: project.py [-h] [-d] {status,next,run,script,submit,exec} ...
```

**Note:** You can have multiple implementations of `FlowProject` that all operate on the same `signac` project! This may be useful, for example, if you want to implement two very distinct workflows that operate on the same data space. Simply put those in different modules, e.g., `project_a.py` and `project_b.py`.

### Defining a workflow

We will reproduce the simple workflow introduced in the previous section by first copying both the `greeted()` condition function and the `hello()` operation function into the `project.py` module. We then use the `operation()` and the `post()` decorator functions to specify that the `hello()` operation function is part of our workflow and that it should only be executed if the `greeted()` condition is not met.

```python
# project.py
from flow import FlowProject
```

(continues on next page)
We can define both pre and post conditions, which allow us to define arbitrary workflows as an acyclic graph. A operation is only executed if all pre-conditions are met, and at least one post-condition is not met.

**Tip:** Cheap conditions should be placed before expensive conditions as they are evaluated lazily! That means for example, that given two pre-conditions, the following order of definition would be preferable:

```python
@Project.operation
@Project.pre(cheap_condition)
@Project.pre(expensive_condition)
def hello(job):
    pass
```

The same holds for post-conditions.

We can then execute this workflow with:

```
~/my_project $ python project.py run
Execute operation 'hello(15e548a2d943845b33030e68801bd125)' ...
hello 15e548a2d943845b33030e68801bd125
Execute operation 'hello(288f97857257baee75d9d84bf0e9dfa8)' ...
hello 288f97857257baee75d9d84bf0e9dfa8
Execute operation 'hello(2b985fa90138327bef586f9ad87fc310)' ...
hello 2b985fa90138327bef586f9ad87fc310
# ...
```

If we implemented and integrated the operation and condition functions correctly, calling the run command twice should produce no output the second time, since the greeted() condition is met for all jobs and the hello() operation should therefore not be executed.

**Tip:** The @with_job decorator can be used so the entire operation takes place in the job context. For example:
Is the same as:

```python
@Project.operation
@Project.post(greeted)
def hello(job):
    with job:
        with open('hello.txt', 'w') as file:
            file.write('world!
')
```

This saves a level of indentation and makes it clear the entire operation should take place in the `job` context. `@with_job` also works with the `@cmd` decorator but must be used first, e.g.:

```python
@Project.operation
@with_job
@cmd
def hello(job):
    return "echo 'hello {}'".format(job)
```

### The Project Status

The `FlowProject` class allows us to generate a status view of our project. The status view provides information about which conditions are met and what operations are pending execution.

A condition function which is supposed to be shown in the status view is called a label-function. We can convert any condition function into a label function by adding the `label()` decorator:

```python
# project.py
# ...
@Project.label
def greeted(job):
    return job.isfile('hello.txt')
# ...
```

We will reset the workflow for only a few jobs to get a more interesting status view:

```
~/my_project $ signac find a.\$lt 5 | xargs -I{} rm workspace/{}\hello.txt
```

We then generate a detailed status view with:

```
~/my_project.py status --detailed --stack --pretty
Collect job status info: 100%|| 10/10
# Overview:
Total # of jobs: 10

   label   ratio
------- --------------------------------------------------
greeted |###########################----------------| 50.00%

# Detailed View:
```
This view provides information about what labels are met for each job and what operations are eligible for execution. If we did things right, then only those jobs without the `greeted` label should have the `hello` operation pending. As shown before, all `eligible` operations can then be executed with:

```
~/my_project $ python project.py run
```

### Generating Execution Scripts

Instead of executing operations directly we can also create a script for execution. If we have any pending operations, a script might look like this:

```
~/my_project $ python project.py script

set -e
set -u

cd /Users/csadorf/my_project

# Operation 'hello' for job '14fb5d016557165019abaac200785048':
/Users/csadorf/miniconda3/bin/python project.py exec hello
→14fb5d016557165019abaac200785048
# Operation 'hello' for job '2af7905eb91ada597a8d4bb91a1c0fc':
/Users/csadorf/miniconda3/bin/python project.py exec hello
→2af7905eb91ada597a8d4bb91a1c0fc
# Operation 'hello' for job '42b7b4f2921788ea14dac5566e6f06d0':
/Users/csadorf/miniconda3/bin/python project.py exec hello
→42b7b4f2921788ea14dac5566e6f06d0
# Operation 'hello' for job '9bfd29df07674bc4aa960cf661b5acd2':
/Users/csadorf/miniconda3/bin/python project.py exec hello
→9bfd29df07674bc4aa960cf661b5acd2
# Operation 'hello' for job '9f8a8e5ba8c70c774d410a9107e2a32b':
/Users/csadorf/miniconda3/bin/python project.py exec hello
→9f8a8e5ba8c70c774d410a9107e2a32b
```

These scripts can be used for the execution of operations directly, or they could be submitted to a cluster environment for remote execution. For more information about how to submit operations for execution to a cluster environment,
see the `Cluster Submission` chapter.

This script is generated from a default Jinja2 template, which is shipped with the package. We can extend this default template or write our own to customize the script generation process.

Here is an example for such a template, that would essentially generate the same output:

```bash
cd {{ project.config.project_dir }}
{% for operation in operations %}
    operation.cmd
{% endfor %}
```

**Note:** Unlike the default template, this exemplary template would not allow for parallel execution.

Checkout the next section for a guide on how to submit operations to a cluster environment.

**Cluster Submission**

While it is always possible to manually submit scripts like the one shown in the previous section to a cluster, using the flow interface will allow us to keep track of submitted operations for example to prevent the resubmission of active operations.

In addition, signac-flow uses environment profiles to select which base template to use for the cluster job script generation. All base templates are in essence the same, but will be slightly adapted to the current cluster environment. That is because different cluster environments offer different resources and therefore require slightly different options for submission. You can check out the available options with the `python project.py submit --help` command.

**The submit interface**

In general, we submit operations through the primary interface of the FlowProject. We assume that we use the same `project.py` module as shown in the previous section.

Then we can submit operations from the command line with the following command:

```bash
~/.my_project $ python project.py submit
```

This will submit all eligible job-operations to the cluster scheduler and block that specific job-operation from resubmission.

In some cases you can provide additional arguments to the scheduler, such as which partition to submit to, which will then be used by the template script. In addition you can always forward any arguments directly to the scheduler as positional arguments. For example, if we wanted to specify an account name with a torque scheduler, we could use the following command:

```bash
~/.my_project $ python project.py submit -- -l A:my_account_name
```

Everything after the two dashes `--` will not be interpreted by the submit interface, but directly forwarded to the scheduler as is.

Unless you have one of the supported schedulers installed, you will not be able to submit any operations in your local environment. However, signac-flow comes with a simple scheduler for testing purposes. You can execute it with `$ simple-scheduler run` and then follow the instructions on screen.
Submitting specific Operations

The submission process consists of the following steps:

1. Gathering of all job-operations eligible for submission.
2. Generation of scripts to execute those job-operations.
3. Submission of those scripts to the scheduler.

The first step is largely determined by your project workflow. You can see which operation might be submitted by looking at the output of `$ python project.py status --detailed`. You may further reduce the operations to be submitted by selecting specific jobs (e.g. with the `-j`, `-f`, or `-d` options), specific operations (`-o`), or generally reduce the total number of operations to be submitted (`-n`). For example the following command would submit up to 5 hello operations, where `a` is less than 5.

```
~/my_project $ python project.py submit -o hello -n 5 -f a.$lt 5
```

The submission scripts are generated using the same templating system like the `script` command.

Tip: Use the `--pretend` or `--test` option to pre-view the generated submission scripts on screen instead of submitting them.

Parallelization and Bundling

By default all eligible job-operations will be submitted as separate cluster jobs. This is usually the best model for clusters that provide shared compute partitions. However, sometimes it is beneficial to execute multiple operations within one cluster job, especially if the compute cluster can only make reservation for full nodes.

You can place multiple job-operations within one cluster submission with the `--bundle` option. For example, the following command will bundle up to 5 job-operations to be executed in parallel into a single cluster submission:

```
~/my_project $ python project.py submit --bundle=5 --parallel
```

Without any argument the `--bundle` option will bundle all eligible job-operations into a single cluster job.

Tip: Recognizing that `--bundle=1` is the default option might help you to better understand the bundling concept.

For more information on managing different environments, see the next section.

Manage Environments

The `signac-flow` package uses environment profiles to adjust the submission process to local environments. That is because different environments provide different resources and options for the submission of operations to those resources. Although the basic options will always be the same, there might be some subtle differences depending on where you want to submit your operations.

Tip: If you are running on a high-performance super computer, add the following line to your `project.py` module to import packaged profiles: `import flow.environments Please see Supported Environments for more information.`
**How to Use Environments**

Environments are defined by subclassing from the `ComputeEnvironment` class. The `ComputeEnvironment` class is a *meta-class* that ensures that all subclasses are automatically globally registered when they are defined. This enables us to use environments simply by defining them or importing them from a different module. The `flow.get_environment()` function will go through all defined `ComputeEnvironment` classes and return the one where the `is_present()` class method returns `True`.

**Packaged Environments**

The package comes with a few *default environments* which are *always available* and designed for specific schedulers. That includes the `DefaultTorqueEnvironment` and the `DefaultSlurmEnvironment`. This means that if you are within an environment with a *torque* or *slurm scheduler* you should be immediately able to submit to the cluster.

In addition, `signac-flow` comes with some environments tailored to specific compute clusters that are defined in the `flow.environments` module. These environments are not automatically available. Instead, you need to *explicitly import* the `flow.environments` module.

For a full list of all packaged environments, please see Supported Environments.

**Defining New Environments**

In order to implement a new environment, create a new class that inherits from `flow.ComputeEnvironment`. You will need to define a detection algorithm for your environment, by default we use a regular expression that matches the return value of `socket.getfqdn()`.

Those are the steps usually required to define a new environment:

2. Determine a regular expression that would match the output of `socket.getfqdn()`.
3. Create a template and specify the template name as `template` class variable.

This is an example for a typical environment class definition:

```python
class MyUniversityCluster(flow.DefaultTorqueEnvironment):
    hostname_pattern = r'\.*\mycluster\.university\\.edu$'  # Matches names like login.
    template = 'mycluster.myuniversity.sh'
```

Then, add the `mycluster.myuniversity.sh` template script to the `templates/` directory within your project root directory.

**Important:** The new environment will be automatically registered and used as long as it is either defined within the same module as your `FlowProject` class or its module is imported into the same module.

As an example on how to write a submission script template, this would be a viable template to define the header for a SLURM scheduler:
Contributing Environments to the Package

Users are **highly encouraged** to contribute environment profiles that they developed for their local environments. In order to contribute an environment, either simply email them to the package maintainers (see the README for contact information) or create a pull request.

Templates

The `signac-flow` package uses jinja2 templates to generate scripts for execution and submission to cluster scheduling systems. Templates for simple bash execution and submission to popular schedulers and compute clusters are shipped with the package. To customize the script generation, a user can replace the default template or customize any of the provided ones.

Replacing the default template

The default `script.sh` template simply extends from another script according to the `base_script` variable, which is dynamically set by `signac-flow`:

```bash
{% extends "base_script.sh" %}
{% block header %}
#!/bin/bash
#SBATCH --job-name="{{ id }}"
#SBATCH --partition="{{ partition }}"
#SBATCH -t {{ walltime|format_timedelta }}
{% block tasks %}
#SBATCH --ntasks="{{ np_global }}"
{% endblock %}
{% endblock %}
```

This `base_script` variable provides a way to inject specific templates for, *e.g.*, different environments. However, by default any templates placed within your project root directory in a file called `templates/script.sh` will be used instead of the defaults provided by `signac-flow`. This makes it easy to completely replace the scripts provided by signac-flow: to use your own custom script, simply place a new `script.sh` in a `templates` directory within your project root directory. This is an example for a basic template that would be sufficient for the simple serial execution of multiple operations:

```bash
cd {{ project.config.project_dir }}
{% for operation in operations %}
  {{ operation.cmd }}
{% endfor %}
```

Customize provided templates

Instead of simply replacing the template as shown above, we can also customize the provided templates. One major advantage is that we can still use the template scripts for cluster submission.
Assuming that we wanted to write a time stamp to some log file before executing operations, we could provide a custom template such as this one:

```{%
extends base_script 
%
block body 
%
date >> execution.log
%
super() 
%
endblock 
%}
```

The first line again indicates that this template extends an existing template based on the value of `base_script`; how this variable is set is explained in more detail in the next section. The second and last line indicate that the enclosed lines are to be placed in the `body` block of the base template. The third line is the actual command that we want to add and the fourth line ensures that the code provided by the base template within the `body` block is still added.

### The base template

The **signac-flow** package will select a different base script template depending on whether you are simply generating a script using the `script` command or whether you are submitting to a scheduling system with `submit`. In the latter case, the base script template is selected based on whether you are on any of the officially supported environments, and if not, whether one of the known scheduling system (torque or slurm) is available. This is a short illustration of that heuristic:

```
# The `script` command always uses the same base script template:
project.py script --> base_script='base_script.sh'

# On system with SLURM scheduler:
project.py submit --> base_script='slurm.sh' (extends 'base_script.sh')

# On XSEDE Comet
project.py submit --> base_script='comet.sh' (extends 'slurm.sh')
```

Regardless of which **base script template** you are actually extending from, all templates shipped with **flow** follow the same basic structure:

- **resources** Calculation of the total resources required for the execution of this (submission) script.
- **header** Directives for the scheduling system such as the cluster job name and required resources. This block is empty for shell script templates.
- **project_header** Commands that should be executed once before the execution of operations, such as switching into the project root directory or setting up the software environment.
- **body** All commands required for the actual execution of operations.
- **footer** Any commands that should be executed at the very end of the script.

### Execution Directives

### Available directives

Any **FlowProject** **operation** can be amended with so called **execution directives**. For example, to specify that we want to parallelize a particular operation on 4 processing units, we would provide the `np=4` directive:

```python
from flow import FlowProject, directives
from multiprocessing import Pool
```

(continues on next page)
All directives are essentially conventions, the \texttt{np} directive in particular means that this particular operation requires 4 processors for execution. The following directives are respected by all base templates shipped with \texttt{signac-flow}:

- **executable** Specify which Python executable should be used to execute this operation. Defaults to the one used to generate the script (\texttt{sys.executable}).

- **np** The total number of processing units required for this operation.

- **nranks** The number of MPI ranks required for this operation. The command will be prefixed with environment specific MPI command, e.g.: \texttt{mpiexec -n 4}. The value for \texttt{np} will default to \texttt{nranks} unless specified separately.

- **omp_num_threads** The number of OpenMP threads. The value for \texttt{np} will default to: “\texttt{nranks x omp_num_threads}” unless otherwise specified.

- **ngpu** The number of GPUs required for this operation.

### Execution Modes

Using these directives and their combinations allows us to realize the following essential execution modes:

- **serial**: \texttt{@flow.directives()}
  
  This operation is a simple serial process, no directive needed.

- **parallelized**: \texttt{@flow.directives(np=4)}
  
  This operation requires 4 processing units.

- **MPI parallelized**: \texttt{@flow.directives(nranks=4)}
  
  This operation will be executed on 4 MPI ranks.

- **MPI/OpenMP Hybrid**: \texttt{@flow.directives(nranks=4, omp_num_threads=2)}
  
  This operation will be executed on 4 MPI ranks with 2 OpenMP threads per rank.

- **GPU**: \texttt{@flow.directives(ngpu=1)}
  
  The operation requires one GPU for execution.

### 1.5.3 Additional Topics

#### Indexing

**Concept**

Data spaces managed with \texttt{signac} on the file system are immediately searchable because \texttt{signac} creates an index of all relevant files \textit{on the fly} whenever a search operation is executed. This data index contains all information about the project’s files, their location and associated metadata such as the \texttt{signac id} and the \texttt{state point}.

A file index has \textit{one entry per file} and each document has the following fields:
• **id**: a unique value which serves as a primary key  
• **root**: The root path of the file  
• **filename**: The filename of the file  
• **md5**: A MD5-hash value of the file content  
• **file_id**: A number identifying the file content\(^1\)  
• **format**: A format definition (optional)

The **signac** project interface is specifically designed to assist with processes related to data curation. However, especially when working with a data set comprised of multiple projects or sources that are not managed with **signac**, it might be easier to work with a data index directly.

For example, this is how we would access files related to a specific data subset using the project interface:

```python
for job in project.find_jobs({"a": 42}):
    with open(job.fn('hello.txt')) as file:
        print(file.read())
```

And this is how we would do the same, but operating directly with an index:

```python
index = signac.Collection(project.index(".*\.txt"))

for doc in index.find({
    "statepoint.a": 42,
    "filename": {"$regex": "hello.txt"}}):
    with signac.fetch(doc) as file:
        print(file.read())
```

Here, we first generate the index with the `Project.index()` function and stored the result in a `Collection` container. Then, we search the index collection for a specific state point and use `fetch()` to open the associated file. The `fetch()` function works very similar to Python’s built-in `open()` function to open files, but in addition will be able to fetch a file from multiple different sources if necessary.

The next few sections are a more detailed outline of how such a workflow can be realized.

### Indexing a signac Project

As shown in the previous section, a **signac** project index can be generated directly with the `Project.index()` function in Python. Alternatively, we can generate the index on the command line with `$ signac project --index`.

A signac project index is like a regular file index, but contains the following additional fields:

• **signac_id**: The state point id the document is associated with.

• **statepoint**: The state point mapping associated with the file.

Each signac project index will have **at least one** entry for each initialized job. This special index document is associated with the job's `document` file and contains not only the `signac_id` and the `statepoint`, but also the data stored in the job document. This means the following code snippet would be valid:

```python
for job in project:
    job.document['foo'] = 'bar'
```

---

\(^1\) Identical to the **md5** value in the current implementation.
for doc in project.index():
    assert doc['foo'] == 'bar'

By default, no additional files are indexed; the user is expected to *explicitly* specify which files should be part of the index as described in the next section.

Indexing files

Indexing specific files as part of a project index requires using regular expressions. For instance, in the initial example we used the expression ".*\.txt" to specify that all files with a filename ending with "txt" should be part of the index.

We can extract metadata directly from the filename by using regular expressions with named groups. For example, if we have a filename pattern: a_0.txt, a_1.txt and so on, where the number following a_ is to be extracted as the a field, we can use the following expression:

```python
for doc in project.index('.*a_(?P<a>\d+):':
    print(doc['a'])
```

To further simplify the selection of different files from the index, we may provide multiple patterns with an optional format definition. Let’s imagine we would like to classify the text files with the a field from the previous example as well as PDF-files that adhere to the following pattern: init.pdf or final.pdf. This is how we could generate this index:

```python
formats = {
    '.*a_(?P<a>\d+).txt': 'TextFile',
    '.*(?P<class>init|final).pdf': 'PDFFile'}

for doc in project.index(formats):
    print(doc)
```

**Tip:** To generate regular expressions for the filename patterns in your data space, copy & paste a few representative filenames into the excellent regex101 online app. That will allow you to work out your expressions while getting direct graphical feedback.

If we want to file an arbitrary directory structure that is not managed by signac, we can use the `signac.index_files()` function, that expects the root path as the first argument, and indexes all files by default.

```python
for doc in signac.index_files('/data'):
    pass
```

Fetching Data

Index documents can be used to directly fetch associated data. The `signac.fetch()` function is essentially equivalent to python’s built-in `open()` function, but instead of a file path it uses an index document² to locate and open the file.

---

² or a file id
The `fetch()` function will attempt to retrieve data from more than one source if data was *mirrored*. Overall, this enables us to operate on indexed project data in a way which is more agnostic to its actual source.

### Deep Indexing

We may want to add additional metadata to the index that is neither based on neither the state point, the job document, or the filename, but instead is directly extracted from the data. Such a pattern is typically referred to as *deep indexing* and can be easily implemented with `signac`.

As an example, imagine that we wanted add the number of lines within a file as an additional metadata field in our data index. For this, we use Python’s built-in `map()` function, which allows us to apply a function to all index entries:

```
def add_num_lines(doc):
    if 'filename' in doc:
        with signac.fetch(doc) as file:
            doc['num_lines'] = len(list(file))
    return doc

index = map(add_num_lines, project.index())
```

The `index` variable now contains an index, where each index entry has an additional `num_lines` field.

**Tip:** We are free to apply multiple *deep indexing* functions in succession; the functions are only executed when the `index` iterable is actually evaluated.

### Searching an Index

An index generated with the `Project.index()` method or any other index function is just an iterable over the index documents. To be able to *search* the index, we need to either implement routines to select specific documents or use containers that implement such routines, such as the `Collection` class that `signac` uses internally for all search operations.

For example, if we are looking for all files that correspond to a state point variable `a=42`, we could implement the following for-loop:

```
index = project.index()
docs = []
for doc in index:
    if doc['statepoint']['a'] == 42:
        docs.append(doc)
```

This is the same logic implemented more concisely as a list comprehension:

```
docs = [doc for doc in index if doc['statepoint']['a'] == 42]
```

Using loops is a very viable approach as long as the index is not too large and the search queries are relatively simple. Alternatively, we can manage the index using a `Collection` container, which then allows us to search the index with
the query expressions that we are used to elsewhere using `signac`. For example, to execute the same search operation from above, we could use the `find()` method:

```python
index = Collection(signac.index())
docs = index.find({'statepoint.a': 42})
```

**Tip**

You can search a collection on the command line by calling it’s `main()` method.

Unless they are very small, searching collections is usually much more efficient than the pure python approach, especially when searching multiple times within the same session. Furthermore, since a collection may be saved to and loaded from a file, we only have to generate an index once, saving us the effort of regenerating it each time we use it:

```python
with Collection.open('index.txt') as index:
    if update_index:
        index.update(signac.index())
    docs = index.find({'statepoint.a': 42})
```

Since `signac`’s decentralized approach is not designed to automatically keep track of changes, it is up to the user to determine when a particular index needs to be updated. To automatically identify and remove stale documents\(^3\), use the `signac.export()` function:

```python
with Collection.open('index.txt') as index:
    signac.export(signac.index(), index, update=True)
```

**Tip:** The `Collection` class has the same interface as a `pymongo.collection.Collection` class. That means you can use these two types of collections interchangeably.

---

**Master Indexes**

**Generating a Master Index**

A master index is a compilation of multiple indexes that simplifies operating on a larger data space. To make a signac project part of a master index, we simply create a file called `signac_access.py` in its root directory. The existance of this file tells `signac` that the projects in those directories should be indexed as part of a master index.

Imagine that we have two projects in two different directories `~/project_a` and `~/project_b` within our home directory. We create the `signac_access.py` file in each respective project directory like this:

```bash
$ touch ~/project_a/signac_access.py
$ touch ~/project_b/signac_access.py
```

Executing the `index()` function for the home directory

```python
for doc in signac.index('~'):
    print(doc)
```

\(^3\) A stale document is associated with a file or state point that has been removed.
will now yield a joint index for both projects in ~/project_a and ~/project_b.
For more information on how to have more control over the index creation, see the signac access module section.

**Tip:** By typing $ signac index you can directly generate a signac master index on the command line and then pipe it into a file:

```
$ signac index > index.txt
```

---

### The signac_access.py Module

We can use the signac_access.py module to control the index generation across projects. An empty module is equivalent to a module which contains the following directives:

```python
import signac

def get_indexes(root):
    yield signac.get_project(root).index()
```

This means that any index yielded from a get_indexes() function defined within the access module will be compiled into the master index.

By putting this code explicitly into the module, we have full control over the index generation. For example, to specify that all files with filenames ending with .txt should be added to the index, we would put the following code into the module:

```python
import signac

def get_indexes(root):
    yield signac.get_project(root).index(formats='.*\.txt')
```

You can generate a basic access module for a signac project using the create_access_module() method.

**Tip:** The signac_access.py module is perfectly suited to implement deep indexing patterns.

---

### Database Integration

#### Database access

After configuring one or more database hosts you can access a database with the signac.get_database() function.

#### Mirroring of Data

Using the signac.fetch() function it is possible retrieve files that are associated with index documents. Those files will preferably be opened directly via a local system path. However, in some cases it may be desirable to mirror files at a different location, e.g., in a database or a different path, to increase the accessibility of files.
Use the mirrors argument in the `signac.export()` function to automatically mirror all files associated with exported index documents. `signac` provides handlers for a local file system and the MongoDB GridFS database file system.

```python
from signac import fs, export, get_database

db = get_database('mirror')
localfs = fs.LocalFS('/path/to/mirror')
gridfs = fs.GridFS(db)
export(crawler.crawl(), db.index, mirrors=[localfs, gridfs])
```

To access the data, provide the mirrors argument to the `signac.fetch()` function:

```python
for doc in index:
    with signac.fetch(doc, mirrors=[localfs, gridfs]) as file:
        do_something_with_file(file)
```

**Note:** File systems are used to fetch data in the order provided, starting with the native data path.

### Using Tags to Control Access

It may be desirable to only index select projects for a specific master index, e.g., to distinguish between public and private indexes. For this purpose, it is possible to specify tags that are **required** by a crawler or index. This means that an index **requiring** tags will be ignored during a master index compilation, unless at least one of the tags is also **provided**.

For example, you can define **required** tags for indexes returned from the `get_indexes()` function, by attaching them to the function like this:

```python
def get_indexes(root):
    yield signac.get_project(root).index()
get_indexes.tags = {'public', 'foo'}
```

Similarly, you can require tags for specific crawlers:

```python
class MyCrawler(SignacProjectCrawler):
    tags = {'public', 'foo'}
```

Unless you **provide at least one** of these tags (`public` or `foo`), the examples above would be ignored during the master index compilation. This means only the second one of the following two lines would **not ignore** the examples above:

```python
index = signac.index()  # examples above are ignored
index = signac.index(tags={'public'})  # includes examples above
```

Similarly on the command line:

```bash
$ signac index          # examples above are ignored
$ signac index --tags public  # includes examples above
```

In summary, there must be an overlap between the **requested** and the **provided** tags.
How to publish an index

Here we demonstrate how to compile a master index with data mirroring, which is designed to be publicly accessible. The index will be stored in a document collection called `index` as part of a database called `public_db`. All data files will be mirrored within the same database. That means everybody with access to the `public_db` database will have access to the index as well as to the associated files.

```python
import signac

db = signac.get_database('public_db')

# We define two mirrors
file_mirrors = [
    # The GridFS database file system is stored in the
    # same database, that we use to publish the index.
    # This means that anyone with access to the index,
    # will be able to access the associated files as well.
    signac.fs.GridFS(db),

    # The second mirror is on the local file system.
    # It can be downloaded and made available locally,
    # for example to reduce the amount of required
    # network traffic.
    signac.fs.LocalFS('/path/to/mirror')
]

# Only crawlers which have been explicitly cleared for
# publication with the 'public' tag will be compiled and exported.
index = signac.index('/path/to/projects', tags={'public'})

# The export() function pushes the index documents to the database
# collection and copies all associated files to the file mirrors.
signac.export(index, db.index, file_mirrors, update=True)
```

Collections

An instance of `Collection` is a container for multiple documents, where a document is an associative array of key-value pairs. Examples are the job state point, or the job document.

The `Collection` class is used internally to manage and search data space indexes which are generated on-the-fly. But you can also use such a container explicitly for managing document data.

Creating collections

To create an empty collection, simply call the default constructor:

```python
from signac import Collection

collection = Collection()
```

You can then add documents with the `signac.Collection.insert_one()` method. Alternatively you can pass an iterable of documents as the first argument, such as the return value of the `signac.Project.index()` method:
By default, the collection is stored purely in memory. But you can use the `signac.Collection` container also to manage collections **directly on disk**. For this, simply open a file like this:

```python
with Collection.open('my_collection.txt') as collection:
    pass
```

A collection file by default is opened in **append plus** mode, that means it is opened for both reading and writing. The `open()` function accepts all standard file open modes, such as `r` for `read-only`, etc.

Large collections can also be stored in a compressed format using gzip for efficiency. To use a compressed collection, simply pass in a compression level from 1-9 as a `compresslevel` argument to the `signac.Collection` constructor:

```python
from signac import Collection

collection = Collection(compresslevel=9)
```

### Searching collections

To search a collection, use the `signac.Collection.find()` method. As an example, to search all documents where the value `a` is equal to 42, execute:

```python
for doc in collection.find({'a': 42}):
    pass
```

The `signac.Collection.find()` method uses the framework-wide `query` API.

### Command Line Interface

To manage and search a collection file directly from the command line, create a python script with the following content:

```python
from signac import Collection

with Collection.open('my_collection.txt') as c:
    c.main()
```

Storing the code above in a file called `find.py` and then executing it will allow you to search for all or specific documents within the collection, directly from the command line `$ python find.py`.

For more information on how to use the command line interface, execute: `$ python find.py --help`.

### Configuration

#### Overview

The `signac` framework is configured with configuration files, which are named either `.signacrc` or `signac.rc`. These configuration files are searched for at multiple locations in the following order:

1. in the current working directory,
2. in each directory above the current working directory until a project configuration file is found,
3. and the user’s home directory.

The configuration file follows the standard “ini-style”. Global configuration options, should be stored in the home directory, while project-specific options should be stored locally in a project configuration file.

This is an example for a global configuration file in the user’s home directory:

```bash
# ~/.signarc
[hosts]
[[localhost]]
url = mongodb://localhost
```

You can either edit these configuration files manually, or execute `signac config` on the command line. Please see `signac config --help` for more information.

**Project configuration**

A project configuration file is defined by containing the keyword `project`. Once `signac` found a project configuration file it will stop to search for more configuration files above the current working directory.

For example, to initialize a project named `MyProject`, navigate to the project’s root directory and either execute `$ signac init MyProject` on the command line, use the `signac.init_project()` function or create the project configuration file manually. This is an example for a project configuration file:

```bash
# signac.rc
project = MyProject
workspace_dir = $HOME/myproject/workspace
```

- **project** The name is required for the identification of the project’s root directory.
- **workspace_dir** The path to your project’s workspace, which defaults to `$project_root_dir/workspace`. Can be configured relative to the project’s root directory or as absolute path and may contain environment variables.

**Host configuration**

The current version of `signac` supports MongoDB databases as a backend. To use `signac` in combination with a MongoDB database, make sure to install `pymongo`.

**Configuring a new host**

To configure a new MongoDB database host, create a new entry in the `[hosts]` section of the configuration file. We can do so manually or by using the `signac config host` command.

Assuming that we a have a MongoDB database reachable via `example.com`, which requires a username and a password for login, execute:

```bash
$ signac config host example mongodb://example.com -u johndoe -p
Configuring new host 'example'.
Password: 
Configured host 'example':
[hosts]
[[example]]
  url = mongodb://example.com
```

(continues on next page)
The name of the configured host (here: example) can be freely chosen. You can omit the -p/\--password argument, in which case the password will not be stored and you will prompted to enter it for each session.

We can now connect to this host with:

```python
>>> import signac
>>> db = signac.get_database('mydatabase', hostname='example')
```

The hostname argument defaults to the first configured host and can always be omitted if there is only one configured host.

**Note:** To prevent unauthorized users from obtaining your login credentials, *signac* will update the configuration file permissions such that it is only readable by yourself.

### Changing the password

To change the password for a configured host, execute

```bash
$ signac host example --update-pw -p
```

**Warning:** By default, any password set in this way will be encrypted. This means that the actual password is different from the one that you entered. However, while it is practically impossible to guess what you entered, a stored password hash will give any intruder access to the database. This means you need to treat the hash like a password!

### Copying a configuration

In general, in order to copy a configuration from one machine to another, you can simply copy the `.signacrc` file as is. If you only want to copy a single host configuration, you can either manually copy the associated section or use the `signac config host` command for export:

```bash
$ signac config host example > example_config.rc
```

Then copy the `example_config.rc` file to the new machine and rename or append it to an existing `.signacrc` file. For security reasons, any stored password is not directly copied in this way. To copy the password, follow:

```bash
# Copy the password from the old machine:
johndoe@oldmachine $ signac config host example --show-pw
XXXX
# Enter it on the new machine:
johndoe@newmachine $ signac config host example -p
```
Manual host configuration

You can configure one or multiple hosts in the [hosts] section, where each subsection header specifies the host’s name.

url The url specifies the MongoDB host url, e.g. mongodb://localhost.

authentication_method (default=none) Specify the authentication method with the database, possible choices are: none or SCRAM-SHA-1.

username A username is required if you authenticate via SCRAM-SHA-1.

password The password to authenticate via SCRAM-SHA-1.

db_auth (default=admin) The database to authenticate with.

password_config In case that you update, but not store your password, the configuration file will contain only meta hashing data, such as the salt. This allows to authenticate by entering the password for each session, which is generally more secure than storing the actual password hash.

Warning: signac will automatically change the file permissions of the configuration file to user read-write only in case that it contains authentication credentials. In case that this fails, you can set the permissions manually, e.g., on UNIX-like operating systems with: chmod 600 ~/.signacrc.

1.6 Packages (API)

The signac framework is currently comprised of three packages. You only have to install those that provide the functionality you need, however both signac-flow and signac-dashboard require the signac core package.

The links below lead to the package-specific documentation, including a complete API documentation and changelogs.

signac (core)

The core signac package implements a simple, serverless, distributed database directly on the file system. It allows you manage files on the file system and associate them with JSON-encoded metadata.

This metadata is immediately searchable, which allows you to find and select data for specific data sub spaces.

signac-flow

The signac-flow package allows us to implement workflows that operate on a data space managed with signac. These workflows range from simple, linear workflows, to large workflows with complex dependencies between operations. These workflows can be executed directly on the command line or submitted to a cluster scheduling system, which is relevant for users who work in high-performance computing (HPC) environments.

signac-dashboard

The signac-dashboard allows users to browse their signac-managed data spaces through a web-based GUI. The dashboard can be used for visualization and analysis and is very helpful when sharing data with collaborators.
1.7 Examples

License

All the code shown here can be downloaded from the signac-docs repository, and is released into the public domain.

This is a collection of example projects which are designed to illustrate how to implement certain applications and solutions with signac. Unlike the tutorial, the examples consist mainly of complete, immediately executable source code with less explanation.

1.7.1 Ideal Gas

This example is based on the Tutorial and assumes that we want to model a system using the ideal gas law:

\[
pV = Nk_B T
\]

The data space is initialized for a specific system size \( N \), thermal energy \( kT \), and pressure \( p \) in a script called init.py:

```python
# init.py
import signac
project = signac.init_project('ideal-gas-project')
for p in range(1, 11):
    sp = {'p': p, 'kT': 1.0, 'N': 1000}
    job = project.open_job(sp)
    job.init()
```

The workflow consists of a compute_volume operation that computes the volume based on the given parameters and stores it within a file called V.txt within each job’s workspace directory. The two additional operations copy the result into a JSON file called data.json and into the job document under the volume key respectively. All operations are defined in project.py:

```python
# project.py
from flow import FlowProject

@FlowProject.label
def volume_computed(job):
    return job.isfile("volume.txt")

@FlowProject.operation
@FlowProject.post(volume_computed)
def compute_volume(job):
    volume = job.sp.N * job.sp.kT / job.sp.p
    with open(job.fn("volume.txt"), "w") as file:
        file.write(str(volume) + 

@FlowProject.operation
@FlowProject.pre.after(compute_volume)
```

(continues on next page)
@FlowProject.post.isfile("data.json")
def store_volume_in_json_file(job):
    import json
    with open(job.fn("volume.txt")) as textfile:
        with open(job.fn("data.json"), "w") as jsonfile:
            data = {"volume": float(textfile.read())}
            jsonfile.write(json.dumps(data) + "\n")

@FlowProject.operation
@FlowProject.pre.after(compute_volume)
@FlowProject.post(lambda job: 'volume' in job.document)
def store_volume_in_document(job):
    with open(job.fn("volume.txt")) as textfile:
        job.document.volume = float(textfile.read())

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

The complete workflow can be executed on the command line with $ python project.py run.

### 1.7.2 MD with HOOMD-blue

This example demonstrates how to setup and analyze the simulation of a Lennard-Jones fluid with molecular dynamics (MD) using HOOMD-blue. The project data space is initialized in a src/init.py script with explicit random seed:

```python
#!/usr/bin/env python

"""Initialize the project's data space.
Iterates over all defined state points and initializes the associated job workspace directories."""
import logging
import argparse
from hashlib import sha1
import signac
import numpy as np

def main(args, random_seed):
    project = signac.init_project('Ideal-Gas-Example-Project')
    for replication_index in range(args.num_replicas):
        for p in np.linspace(0.5, 5.0, 10):
            statepoint = dict(
                # system size
                N=512,

                # Lennard-Jones potential parameters
                sigma=1.0,
                epsilon=1.0,
                r_cut=2.5,

                # random seed
```
seed=random_seed*(replication_index + 1),

    # thermal energy
    kT=1.0,
    # pressure
    P=P,
    # thermostat coupling constant
    tau=1.0,
    # barostat coupling constant
    tauP=1.0

project.open_job(statepoint).init()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Initialize the data space.")
    parser.add_argument(
        'random',
        type=str,
        help="A string to generate a random seed.")
    parser.add_argument(
        '-n', '--num-replicas',
        type=int,
        default=1,
        help="Initialize multiple replications.")
    args = parser.parse_args()

    try:
        random_seed = int(args.random)
    except ValueError:
        random_seed = int(sha1(args.random.encode()).hexdigest(), 16) % (10 ** 8)

    logging.basicConfig(level=logging.INFO)
    main(args, random_seed)

Using this script, one replica set (for a given random seed, e.g., 42) can then be initialized with:

```
$ python src/init.py 42
```

The simulation and analysis workflow is broken into three operations:

1. **init**: Initialize the simulation configuration.
2. **estimate**: Use the ideal gas law to estimate the expected volume.
3. **sample**: Carry out the simulation with HOOMD-blue.

Those three operations and corresponding condition functions are defined and implemented within a `src/project.py` module:

```
"""This module contains the operation functions for this project.

The workflow defined in this file can be executed from the command line with

    $ python src/project.py run [job_id [job_id ...]]
```

(continues on next page)
See also: $ python src/project.py --help

```
from flow import FlowProject

class Project(FlowProject):
    pass

@Project.operation
@Project.post.isfile('init.gsd')
def initialize(job):
    """Initialize the simulation configuration."
    import hoomd
    from math import ceil
    if hoomd.context.exec_conf is None:
        hoomd.context.initialize('')
    with job:
        with hoomd.context.SimulationContext():
            n = int(ceil(pow(job.sp.N, 1.0/3)))
            assert n**3 == job.sp.N
            hoomd.init.create_lattice(unitcell=hoomd.lattice.sc(a=1.0), n=n)
            hoomd.dump.gsd('init.gsd', period=None, group=hoomd.group.all())

@Project.operation
@Project.post(lamda job: 'volume_estimate' in job.document)
def estimate(job):
    """Ideal-gas estimate operation."
    # Calculate volume using ideal gas law
    V = job.sp.N * job.sp.kT / job.sp.p
    job.document.volume_estimate = V

def current_step(job):
    import gsd.hoomd
    if job.isfile('dump.gsd'):
        with gsd.hoomd.open(job.fn('dump.gsd')) as traj:
            return traj[-1].configuration.step
    return -1

@Project.label
def sampled(job):
    return current_step(job) >= 5000

@Project.label
def progress(job):
    return '{:.4f}'.format(int(round(current_step(job) / 5000) * 4))

@Project.operation
@Project.pre.after(initialize)
@Project.post(sampled)
def sample(job):
    """Sample operation."""
import logging
import hoomd
from hoomd import md
if hoomd.context.exec_conf is None:
    hoomd.context.initialize('')
with job:
    with hoomd.context.SimulationContext():
        hoomd.init.read_gsd('init.gsd', restart='restart.gsd')
        group = hoomd.group.all()
        gsd_restart = hoomd.dump.gsd('restart.gsd', truncate=True, period=100, phase=0, group=group)
        lj = md.pair.lj(r_cut=job.sp.r_cut, nlist=md.nlist.cell())
        lj.pair_coeff.set('A', 'A', epsilon=job.sp.epsilon, sigma=job.sp.sigma)
        md.integrate.mode_standard(dt=0.005)
        md.integrate.npt(group=group, kT=job.sp.kT, tau=job.sp.tau, P=job.sp.p, tauP=job.sp.tauP)
        hoomd.analyze.log('dump.log', ['volume'], period=100)
        hoomd.dump.gsd('dump.gsd', period=100, group=hoomd.group.all())
        try:
            hoomd.run_upto(5001)
        except hoomd.WalltimeLimitReached:
            logging.warning("Reached walltime limit.")
        finally:
            gsd_restart.write_restart()
            job.document["sample_step"] = hoomd.get_step()
if __name__ == '__main__':
    Project().main()

There are two additional label functions, which show whether the simulation has finished (sampled) and one that shows the rough progress in quarters (progress).
Execute the initialization and simulation with:

```bash
$ python src/project.py run
```

### 1.7.3 Integration with Sacred

Integrating a sacred experiment with signac-flow is very simple. Assuming the following sacred experiment defined in a experiment.py module:

```python
from sacred import Experiment
ex = Experiment()

@ex.main
def hello(foo):
    print('hello', foo)

if __name__ == '__main__':
    ex.run_commandline()
```
Then we can integrate that experiment on a per job basis like this:

```python
def run_experiment(job):
    ex.add_config(**job.sp())
    ex.observers[:]= [FileStorageObserver.create(job.fn('my_runs'))]
    ex.run()
```

```python
if __name__=='__main__':
    SacredProject().main()
```

## 1.8 Recipes

This is a collection of recipes on how to solve typical problems using signac.

### 1.8.1 How to migrate (change) the data space schema.

**Adding/rename/delete keys**

Oftentimes, one discovers at a later stage that important keys are missing from the metadata schema. For example, in the tutorial we are modeling a gas using the ideal gas law, but we might discover later that important effects are not captured using this overly simplistic model and decide to replace it with the van der Waals equation:

\[
\left(p + \frac{N^2 a}{V^2}\right)(V - Nb) = Nk_B T
\]

Since the ideal gas law can be considered a special case of the equation above with \(a = b = 0\), we could migrate all jobs with:

```python
for job in project:
    ... job.sp.setdefault('a', 0)
    ... job.sp.setdefault('b', 0)
    ...
```

The `setdefault()` function sets the value for `a` and `b` to 0 in case that they are not already present.

- To **delete** a key use `del job.sp['key_to_be_removed']`.
- To **rename** a key, use `job.sp.new_name = job.sp.pop('old_name')`.

**Note:** The `job.sp` and `job.doc` attributes provide all basic functions of a regular Python dict.
### Initializing Jobs with Replica Indices

If you want to initialize your workspace with multiple instances of the same statepoint, you may want to include a `replica_index` or `random_seed` parameter in the statepoint.

```python
num_reps = 3
for i in range(num_reps):
    for p in range(1, 11):
        sp = {'p': p, 'kT': 1.0, 'N': 1000, "replica_index": i}
        job = project.open_job(sp)
        job.init()
```

### Apply document-wide changes

The safest approach to apply multiple document-wide changes is to replace the document in one operation. Here is an example on how we could recursively replace all dot (.)-characters with the underscore-character in all keys:

```python
import signac
from collections.abc import Mapping

def migrate(doc):
    if isinstance(doc, Mapping):
        return {k.replace('.', '_'): migrate(v) for k, v in doc.items()}
    else:
        return doc

for job in signac.get_project():
    job.sp = migrate(job.sp)
    job.doc = migrate(job.doc)
```

This approach makes it also easy to compare the pre- and post-migration states before actually applying them.

#### 1.8.2 How to define parameter-dependent operations

Operations defined as a function as part of a `signac-flow` workflow can only have one required argument: the job. That is to ensure reproducibility of these operations. An operation should be a true function of the job’s data without any hidden parameters.

Here we show how to define operations that are a function of one or more additional parameters without violating the above mentioned principle. Assuming that we have an operation called `foo`, which depends on parameter `bar`, here is how we could implement multiple operations that depend on that additional parameter without code duplication:

```python
class Project(FlowProject):
    pass

def add_foo_workflow(bar):
    job.doc.setdefault('foo', dict())

def foo_ran(job):
    return bar in job.doc.foo
```

---

1. The use of dots in keys is deprecated. Dots will be exclusively used to denote nested keywords in the future.
# Make sure to make the operation-name a function of the parameter(s)!

```python
@Project.operation('foo-{}'.format(bar))
@Project.post(foo_ran)
def foo(job):
    job.doc.foo[bar] = 'hello world!'  
```

```python
for bar in (4, 8, 15, 16, 23, 42):
    add_foo_workflow(bar=bar)
```

**Important:** The operation and condition functions must be defined in a way such that they are truly independent from each other! For instance, the example above would fail if all `foo`-operations had the same name or if they were all writing output to the same location.

### 1.8.3 How to integrate signac-flow with MATLAB or other software without Python interface

The easiest way to integrate software that has no native Python interface is to implement `signac-flow` operations in combination with the `flow.cmd` decorator. Assuming that we have a MATLAB script called `prog.m` within the project root directory:

```matlab
function []=prog(arg1, arg2)
    display(arg1);
    display(arg2);
    exitcode = 0;
```

Then, we could implement a simple operation that passes it some metadata parameters like this:

```python
@FlowProject.operation
@flow.cmd
def compute_volume(job):
    return "matlab -r 'prog {job.sp.foo} {job.sp.bar}' > {job.ws}/output.txt"
```

Executing this operation will store the output of the MATLAB script within the job’s workspace within a file called `output.txt`.

### 1.8.4 How to implement MPI-parallelized operations

There are basically two strategies to implement `FlowProject` operations that are MPI-parallelized, one for external programs and one for Python scripts.

**Tip:** Fully functional scripts can be found in the signac-docs repository under `examples/MPI`.  

1.8. Recipes 65
**MPI-operations with mpi4py or similar**

Assuming that your operation is using mpi4py or similar, you do not have to change your code:

```python
@FlowProject.operation
def hello_mpi(job):
    from mpi4py import MPI
    print("Hello from rank", MPI.COMM_WORLD.Get_rank())
```

You could run this operation directly with: `mpiexec -n 2 python project.py run -o hello_mpi`.

**Note:** This strategy might fail in cases where you cannot ensure that the MPI communicator is initialized within the operation function.

**Danger:** Read and write operations to the job-/ and project-document are not protected against race-conditions and should only be executed on one rank at a time. This can be ensured for example like this:

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
if comm.Get_rank() == 0:
    job.doc.foo = 'abc'
comm.barrier()
```

**MPI-operations with flow.cmd**

Alternatively, you can implement an MPI-parallelized operation with the flow.cmd decorator, optionally in combination with the flow.directives decorator. This strategy lets you define the number of ranks directly within the code and is also the only possible strategy when integrating external programs without a Python interface.

Assuming that we have an MPI-parallelized program named `my_program`, which expects an input file as its first argument and which we want to run on two ranks, we could implement the operation like this:

```python
@FlowProject.operation
@flow.cmd
@flow.directives(np=2)
def hello_mpi(job):
    return "mpiexec -n 2 mpi_program {job.ws}/input_file.txt"
```

The flow.cmd decorator instructs signac-flow to interpret the operation as a command rather than a Python function. The flow.directives decorator provides additional instructions on how to execute this operation and is not strictly necessary for the example above to work. However, some script templates, including those designed for HPC cluster submissions, will use the value provided by the np key to compute the required compute ranks for a specific submission.

**Tip:** You do not have to hard-code the number of ranks, it may be a function of the job, e.g.: `flow.directives(np=lambda job: job.sp.system_size // 1000)`.
MPI-operations with custom script templates

Finally, instead of modifying the operation implementation, you could use a custom script template, such as this one:

```{% extends base_script %}
{% block body %}
{% for operation in operations %}
mpiexec -n {{ operation.directives.np }} operation.cmd
{% endfor %}
{% endblock %}
```

Storing the above template in a file called `templates/script.sh` within your project root directory will prepend every operation command with `mpiexec` and so on.

### 1.8.5 How to run in containerized environments

Using `signac-flow` in combination with container systems such as `docker` or `singularity` is easily achieved by modifying the `executable` directive. For example, assuming that we wanted to use a singularity container named `software.simg`, which is placed within the project root directory, we use the following directive to specify that a given operation is to be executed within then container:

```python
@Project.operation
@flow.directives(executable='singularity exec software.simg python')
def containerized_operation(job):
    pass
```

If you are using the `run` command for execution, simply execute the whole script in the container:

```
$ singularity exec software.simg python project.py run
```

**Attention:** Many cluster environments will not allow you to submit jobs to the scheduler using the container image. This means that the actual submission, (e.g. `python project.py submit` or similar) will need to be executed with a local Python executable.

To avoid issues with dependencies that are only available in the container image, move imports into the operation function. Condition functions will be executed during the submission process to determine what to submit, so dependencies for those must be installed into the local environment as well.

**Tip:** You can define a decorator that can be reused like this:

```python
def on_container(func):
    return flow.directives(executable='singularity exec software.simg python')(func)

@on_container
@Project.operation
def containerized_operation(job):
    pass
```
1.9 FAQ

This is a collection of frequently asked questions (and their answers) that might help new users avoid common mistakes or provide useful hints to more experienced users.

1.9.1 How do I design a good schema?

There is really no good answer on how to generally design a good schema because it is heavily dependent on the domain and the specific application. Nonetheless, there are some basic rules worth following:

1. Be descriptive. Although we are using short variable names in the tutorial, in general metadata keys should be as long as necessary for a third party to understand their meaning without needing to ask someone.

2. Any parameter which is likely to be varied at some point during the study should be part of the metadata right from the start to avoid needing to modify the schema later.

3. Take advantage of grouping keys! The job metadata mapping may be nested, just like any other Python dict.

4. Even if you don’t use “official” schemas, consider to work out standardized schemas among your peers or with your collaborators.

5. Use the state point to define the identity of each job, use the document to store additional metadata.

1.9.2 What is the difference between the job state point and the job document?

The state point defines the identity of each job in form of the job id. Conceptually, all data related to a job should be a function of the state point. That means that any metadata that could be changed without invalidating the data, should in principle be placed in the job document.

**Important:** The state point defines the identity of each job, the job document is data.

1.9.3 How do I avoid replicating metadata in filenames?

Many users, especially those new to signac, fall into the trap of storing metadata in filenames within a job’s workspace even though that metadata is already encoded in the job itself.

Using the Tutorial project as an example, we might have stored the volume corresponding to the job at pressure 4 in a file called volume_pressure_4.txt. However, this is completely unnecessary since that information can already be accessed through the job via job.sp.p. Furthermore, creating files this way causes additional complications, such as the need to modify filenames whenever we operate on the data space. For example, extracting the volume from a particular job originally consisted of doing this:

```python
volume = float(open(job.fn('volume.txt')).read())
```

Now, we instead need to adjust the filename for each job:

```python
volume = float(open(job.fn('volume_pressure_{}.txt'.format(job.sp.p))).read())
```

In general, it is desirable to keep the filenames across the workspace as uniform as possible.
1.9.4 How do I reference data/jobs in scripts?

You can reference other jobs in a script using the path to the project root directory in combination with a query-expression. While it is perfectly fine to copy & paste job ids during interactive work or for small tests, hard-coded job ids within code are almost always a bad sign. One of the main advantages of using signac for data management is that the schema is flexible and may be migrated at any time without too much hassle. That also means that existing ids will change and scripts that used them in a hard-coded fashion will fail.

Whenever you find yourself hard-coding ids into your code, consider replacing it with a function that uses the `find_jobs()` function instead.

1.10 Community

1.10.1 Chat Support

To best way to get support for signac is to join the signac-gitter channel. The developers and other users are usually able to help within a few minutes. Alternatively, you can send an email to signac-support@umich.edu.

Please use the issue tracker of the individual packages to file bug reports or request new features!

1.10.2 Contributions

Contributions to signac are very welcome! We highly appreciate contributions in the form of user feedback and bug reports on the gitter channel, the issue trackers of individual packages, or via email. Developers are invited to contribute to the framework by pull request to the appropriate package repository. The source code for all packages is hosted on github. We recommend discussing new features in form of a proposal on the issue tracker for the appropriate project prior to development.

All code contributed via pull request needs to adhere to the following guidelines:

- Use the OneFlow model of development: - Both new features and bug fixes should be developed in branches based on master. - Hotfixes (critical bugs that need to be released fast) should be developed in a branch based on the latest tagged release.
- Write code that is compatible with all supported versions of Python (listed in the package setup.py file).
- Avoid introducing dependencies – especially those that might be harder to install in high-performance computing environments.
- All code needs to adhere to the PEP8 style guide, with the exception that a line may have up to 100 characters.
- Create unit tests and integration tests that cover the common cases and the corner cases of the code.
- Preserve backwards-compatibility whenever possible, and make clear if something must change.
- Document any portions of the code that might be less clear to others, especially to new developers.
- Write API documentation as part of the doc-strings of the package, and put usage information, guides, and concept overviews in the framework documentation, the page you are currently on (source).

Tip: During continuous integration, the code is checked automatically with Flake8. Run the following commands to set up a pre-commit hook that will ensure your code is compliant before committing:

```
flake8 --install-hook git
git config --bool flake8.strict true
```
**Note:** Please see the individual package documentation for detailed guidelines on how to contribute to a specific package.

### 1.11 License

**Important:** Please see the individual package documentation for detailed licensing information.

In general, all packages that are part of the **signac** framework are licensed under the **BSD-3-Clause License**:

```plaintext
BSD 3-Clause License for the software signac.

Copyright (c) 2016-2019, The Regents of the University of Michigan
All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may
   be used to endorse or promote products derived from this software without
   specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

Most source code shown as part of the documentation is released into the public domain:

```plaintext
This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or
 distribute this software, either in source code form or as a compiled
 binary, for any purpose, commercial or non-commercial, and by any
 means.
```

(continues on next page)
In jurisdictions that recognize copyright laws, the author or authors of this software dedicate any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to <https://unlicense.org>

The signac, signac-flow, and signac-dashboard logos & badges are licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license:

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 -- Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)−(2) are not Copyright and Similar Rights.
d. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

i. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

   a. reproduce and Share the Licensed Material, in whole or in part; and

   b. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public
License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

   a. Offer from the Licensor -- Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

   b. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

   1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

   2. Patent and trademark rights are not licensed under this Public License.

   3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory
licensing scheme. In all other cases the Licensor expressly 
reserves any right to collect such royalties.

Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the 
following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified 
form), You must:

   a. retain the following if it is supplied by the Licensor 
   with the Licensed Material:

      i. identification of the creator(s) of the Licensed 
         Material and any others designated to receive 
         attribution, in any reasonable manner requested by 
         the Licensor (including by pseudonym if 
         designated);

      ii. a copyright notice;

      iii. a notice that refers to this Public License;

      iv. a notice that refers to the disclaimer of 
         warranties;

      v. a URI or hyperlink to the Licensed Material to the 
         extent reasonably practicable;

   b. indicate if You modified the Licensed Material and 
   retain an indication of any previous modifications; and

   c. indicate the Licensed Material is licensed under this 
   Public License, and include the text of, or the URI or 
   hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any 
reasonable manner based on the medium, means, and context in 
which You Share the Licensed Material. For example, it may be 
reasonable to satisfy the conditions by providing a URI or 
hyperlink to a resource that includes the required 
information.

3. If requested by the Licensor, You must remove any of the 
information required by Section 3(a)(1)(A) to the extent 
reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's 
License You apply must not prevent recipients of the Adapted 
Material from complying with this Public License.

Section 4 -- Sui Generis Database Rights.
Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 -- Disclaimer of Warranties and Limitation of Liability.

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 -- Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.
b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 -- Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 -- Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.
Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." The text of the Creative Commons public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.

1.12 How to cite signac

Consider to place this badge on your slides or README files! See Badge for instructions.

Please acknowledge the use of this software within the body of your publication for example by copying or adapting the following formulation:

*The computational workflow in general and data management in particular for this publication was primarily supported by the signac data management framework*[1,2].


References for previous release versions can be found here. A preprint of the paper published in the Journal of Computational Materials Science is available on the arXiv.

To cite these references, you can use the following BibTeX entries:

```
@article{signac_commat,
  author = {Carl S. Adorf and
            Paul M. Dodd and
            Vyas Ramasubramani and
            Sharon C. Glotzer},
  title = {Simple data and workflow management with the signac framework},
  journal = {Comput. Mater. Sci.},
  volume = {146},
}
```

(continues on next page)
1.12.1 Badge

You can add the following badge to your slides or README files to acknowledge the use of signac as part of your project:

Downloads

- PDF (high resolution)
- PNG (high resolution)
- SVG (low resolution)

URL


Markdown

![signac](http://docs.signac.io/en/latest/_images/signac.io-blue.svg)

reStructuredText

.. image:: http://docs.signac.io/en/latest/_images/signac.io-blue.svg
   :target: https://signac.io

Badges generated with shields.io.
1.12.2 Logos

The `signac`, `signac-flow`, and `signac-dashboard` logos & badges are licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. This means you can use logos as long as you attribute the logos to the relevant `signac` project.
CHAPTER 2

Indices and tables

• genindex
• modindex
• search
Index

B
body, 45

E
executable, 46

F
footer, 45

G
GPU:, 46

H
header, 45

M
MPI parallelized:, 46
MPI/OpenMP Hybrid:, 46

N
ngpu, 46
np, 46
nranks, 46

O
omp_num_threads, 46

P
parallelized:, 46
project_header, 45

R
resources, 45

S
serial:, 46