GCMCWorkflow Documentation

Richard J Gowers

Nov 20, 2018

Contents:

Welcome to the documentation for GCMCWorkflow, a package for automating Grand Canonical Monte Carlo sampling of porous materials.

CHAPTER 1

Installing GCMCWorkflow

1.1 Installing the GCMCWorkflow package

The easiest way to install the GCMCWorkflow Python package is using Anaconda from the conda-forge channel:

```
conda install -c conda-forge gcmcworkflow
```

or pip:

```
pip install gcmcworkflow
```

1.2 Setting up a Launchpad

GCMCWorkflow is powered by Fireworks and requires a functioning LaunchPad (database of jobs). For instructions on how to do this, please consult the official instructions Ultimately this should result in a *my_launchpad.yaml* file, which works with the various Fireworks commands such as *lpad get_fws*.

1.3 Setting up required drivers

GCMCWorkflow does not perform the individual GCMC simulations, but instead relies on a driver program beneath it. I.e. you use GCMCWorkflow, GCMCWorkflow uses Raspa.

GCMCWorkflow can currently use the following drivers:

- zeo++
- Raspa

CHAPTER 2

GCMCWorkflow tutorial

In this tutorial we will automate running three isotherms of Argon in IRMOF-1 using GCMCWorkflow. To follow this tutorial, you will need to have installed this package succesfully. To check this, try running `gcmcworkflow -h` from the terminal, you should see the help message for the command line tool we will be using.

2.1 Preparing input files

In order to run our GCMC simulations we must provide a single working input for Raspa. This file will be used as a template for all further simulations, with the temperature, pressure and simulation duration settings being customised as necessary. The forcefield setup and MC move descriptions will not be modified however, so these should be checked!

For this tutorial you can use your own Raspa input files, or alternatively by running the following command you can use the tutorial example files:

```
gcmcworkflow tutorial gen_inputfiles
```

2.2 Preparing the spec file

With the Raspa input ready, we now create the input file for GCMCWorkflow, know as the 'spec' file. This file will have all the information on sampling we want to perform, using the Raspa input we just prepared.

A template for this spec file can be generated using the `gcmcworkflow genspec` command. Thie file should follow yaml formatting rules, edit the spec file to read as follows, these settings will be explained below.

```
name: IRMOF1_Ar_tutorial
template: ./irmof/
workdir: ./
g_req: 3
max_iterations: 2
```

(continues on next page)

(continued from previous page)

```
use_grid: true
conditions:
 - pressures: [10k, 20k, 40k]
   temperatures: [78.0, 98.0]
 - pressures: [50k, 0.1M]
   temperatures: [118.0]
```

Explaining the spec file lines:

- **name** – this is a unique identifier for this Workflow. It must be unique on the LaunchPad and will be used to refer to this Workflow in the future
- **template** – path to the simulation input we want to use. This directory will be slurped into the Workflow.
- **workdir** – path to where we want to run the simulations. This path must be accessible on the workstation you want to execute the workflow, ie it might be a path on your HPC cluster.
- **g_req** – this is the desired simulation length, expressed as a multiple of the number of statistical decorrelations after equilibration to run for. Larger values will take longer, but be more statistically reliable answers, 3-5 seems to work ok.
- **max_iterations** – simulations will be restarted to gather enough data as defined above, this setting allows a maximum to the number of times a point is restarted.
- **use_grid** – Raspa allows for energy grids to be used to accelerate the GCMC sampling. This is often a good idea, except for very short simulations.
- **conditions** – starts a list of the system conditions we want to sample. Each entry must give temperatures and pressures. In this example we will run pressures of 10, 20, and 40 kPa for temperatures 78.0 and 98.0 K, then 50,000 and 100,000 kPa at a temperature of 118.0 K.

2.3 Submitting work to LaunchPad

Now we've defined our input and desired sampling we need to submit this information to our LaunchPad. Most of the time (unless you are locally running a MongoDB server) you will have a file called `my_launchpad.yaml` which holds the details to connect to the Fireworks LaunchPad.

Before submitting work to the LaunchPad it is a good idea to check the connection. This can be done using `lpad report`, which will return a quick description of the status of the LaunchPad. If this step does not work, consult the *Setting up a Launchpad* instructions.

To submit the workflow to the LaunchPad we call:

```
gcmcworkflow submit spec.yml -l my_launchpad.yaml
```

This reads the spec file we just created, translates this into a Fireworks Workflow, and submits this to the Fireworks LaunchPad. We can inspect the Workflow using Fireworks commands, such as the web gui (`lpad webgui`).

2.4 Inspecting the Workflow

By checking the webgui of the Workflow, we see a diagram of our Workflow like this:

At the start of the Workflow we can see that that there are two tasks that must be completed first, before we proceed to sampling different temperatures and pressures. These are copying the simulation template onto the worker filesystem

and creating the energy grid. Fireworks has allowed to express dependencies between these, so that the sampling simulations only run once the energy grid has been produced!

2.5 Running our Workflow

The Workflow can now be executed! This is done using the `rlaunch` commands from Fireworks. For example to run the tasks within the Workflow in parallel on 4 cores, you could use:

```
rlaunch multi 4
```

Remember that for this to work a connection to the LaunchPad is required, ie there should be a `launchpad.yaml` file in the directory you run from.

It is worth remembering that the Workflow can be ran on a different workstation than the one which submitted the Workflow. This allows you to define Workflow from your local workstation, then issue jobs on a HPC cluster to pull these jobs annd push results back, then finally download the results back to your local workstation.

2.6 Checking our results

CHAPTER 3

Notes on using in practice

Using GCMCWorkflow on a cluster

- packing jobs together in scripts
- checking progress of jobs
- accessing results once finished
- debugging failures

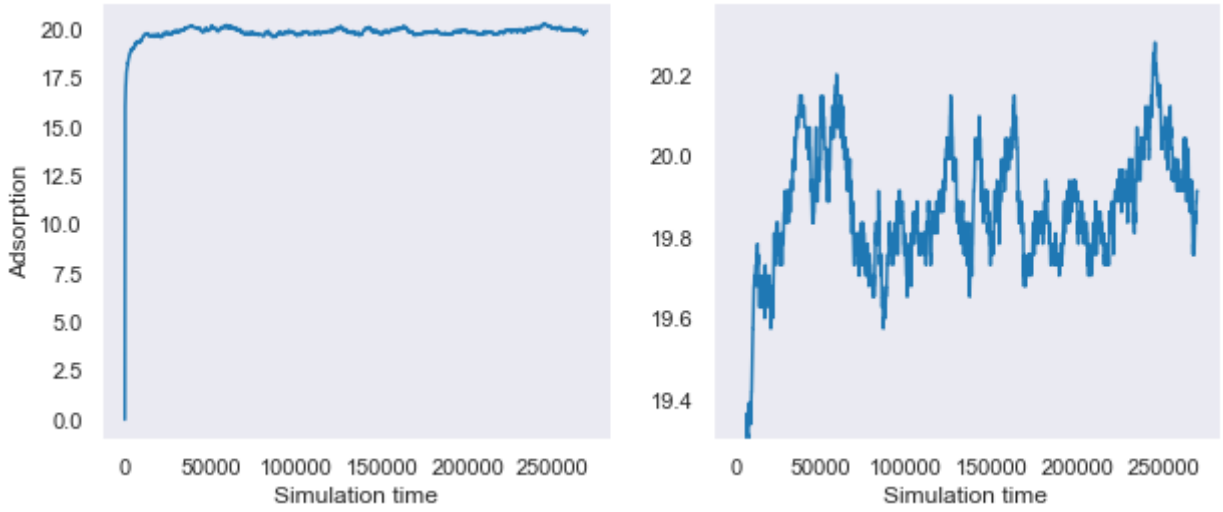
CHAPTER 4

Method descriptions

4.1 Equilibrium finding

Each individual sampling point that is simulated will initially start with an empty system. As the simulation progresses the system will gradually be filled with gas molecules until an equilibrium amount is reached. Data can only be gathered from the system once equilibrium has been reached.

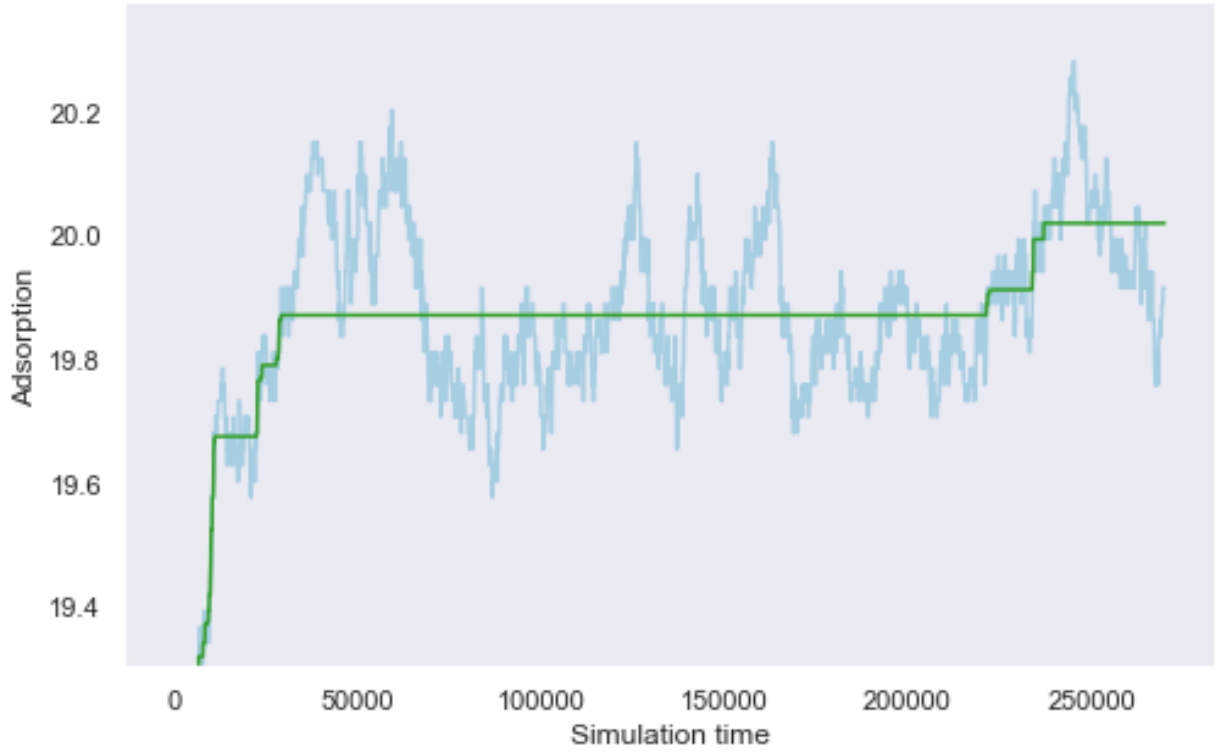
An example timeseries from a simulation is given below, with the right hand panel showing a close up of the data.



This page details the algorithm which detects if and when a simulation has reached equilibrium. This method attempts to find the earliest point in the timeseries where equilibrium has been reached. It will also be responsible for detecting when equilibrium has not been reached.

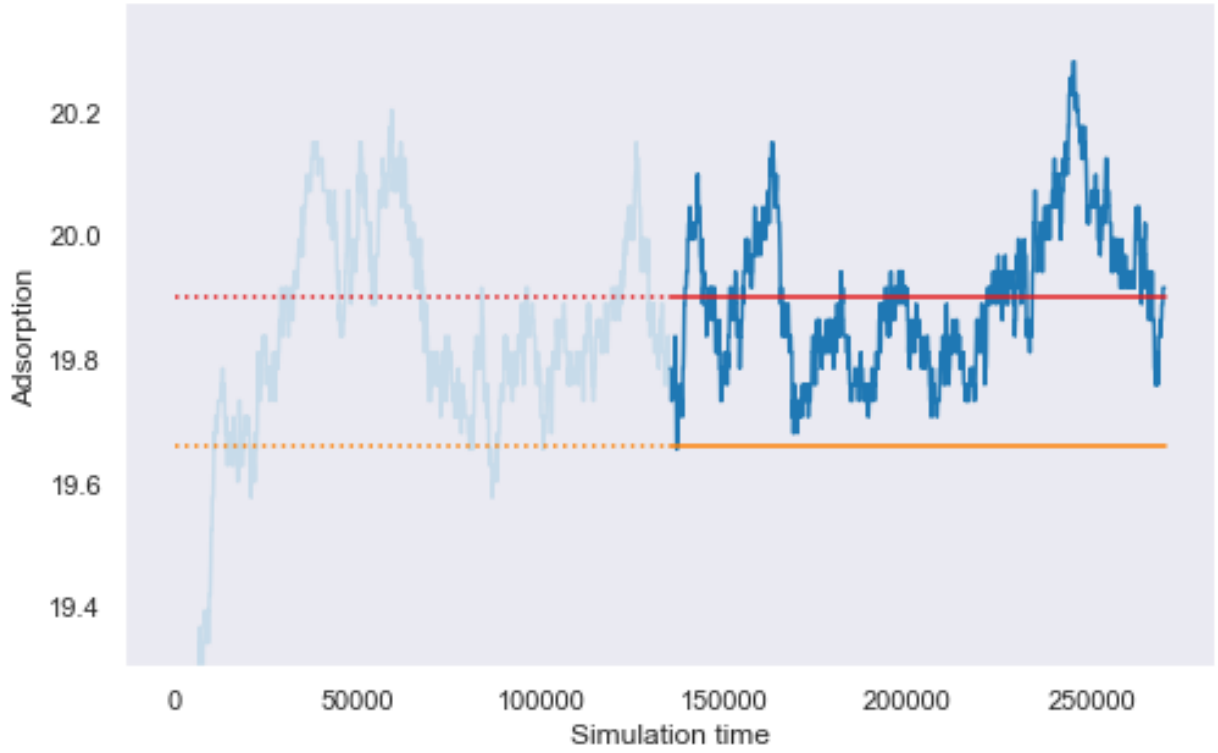
4.1.1 Monotonic fit

From inspecting the timeseries, it is immediately obvious there is some "noise" in the signal. This isn't actually noise, but instead the natural oscillations around the mean value in the system. To help smooth the data, a monotonic regression is fitted to the entire timeseries. A monotonic regression can be thought of as a rolling line of best fit which can only increase or remain constant, but never decrease. As previously mentioned, the system initially starts at 0 and will rise to an equilibrium value, therefore the monotonic fit provides a good model of the smoothed behaviour of the timeseries.



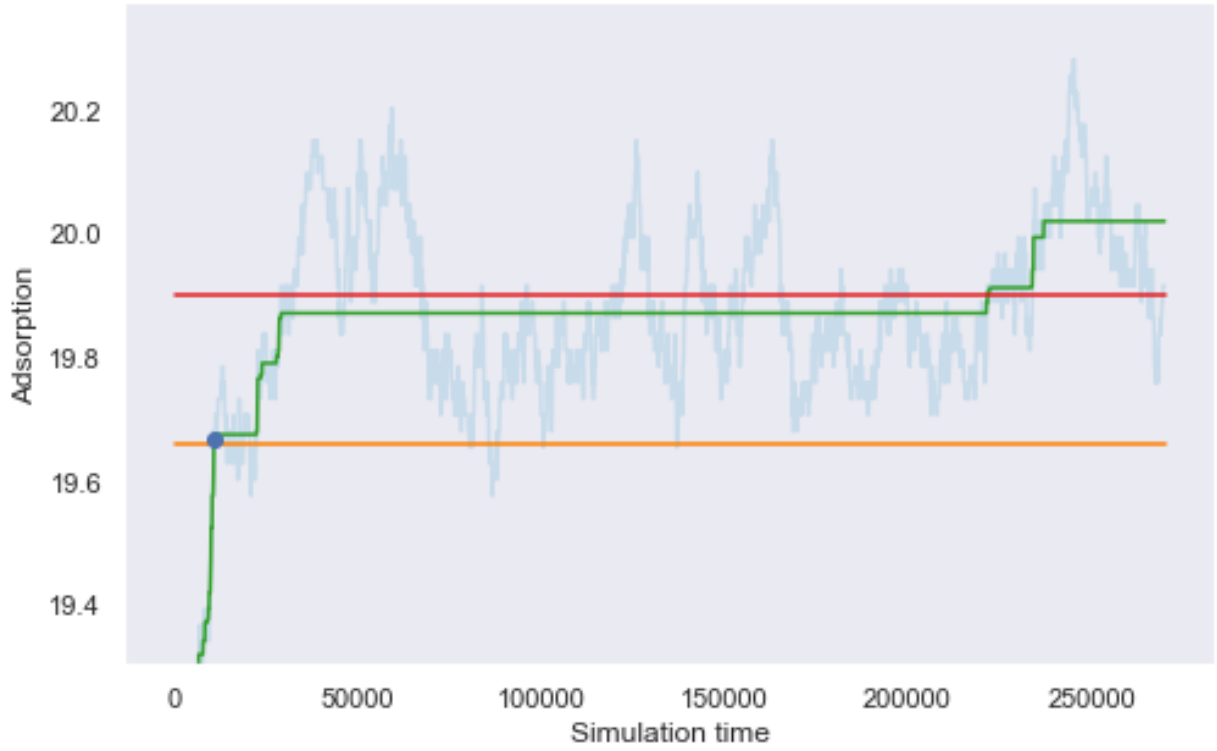
4.1.2 Estimating the mean and standard deviation

Once the signal has reached equilibrium, it will oscillate around a mean value. We therefore want to identify a value which represents the lower bound of what is possible. To do this we assume that the final half of the signal can be used to represent the mean and standard deviation of the timeseries. We then choose a value of 2 standard deviations below the mean to represent our lower bound. We will of course later check the assumptions made here. It is important to note that these values will not be correct estimates of the data, but they are useful for identifying the equilibration point.

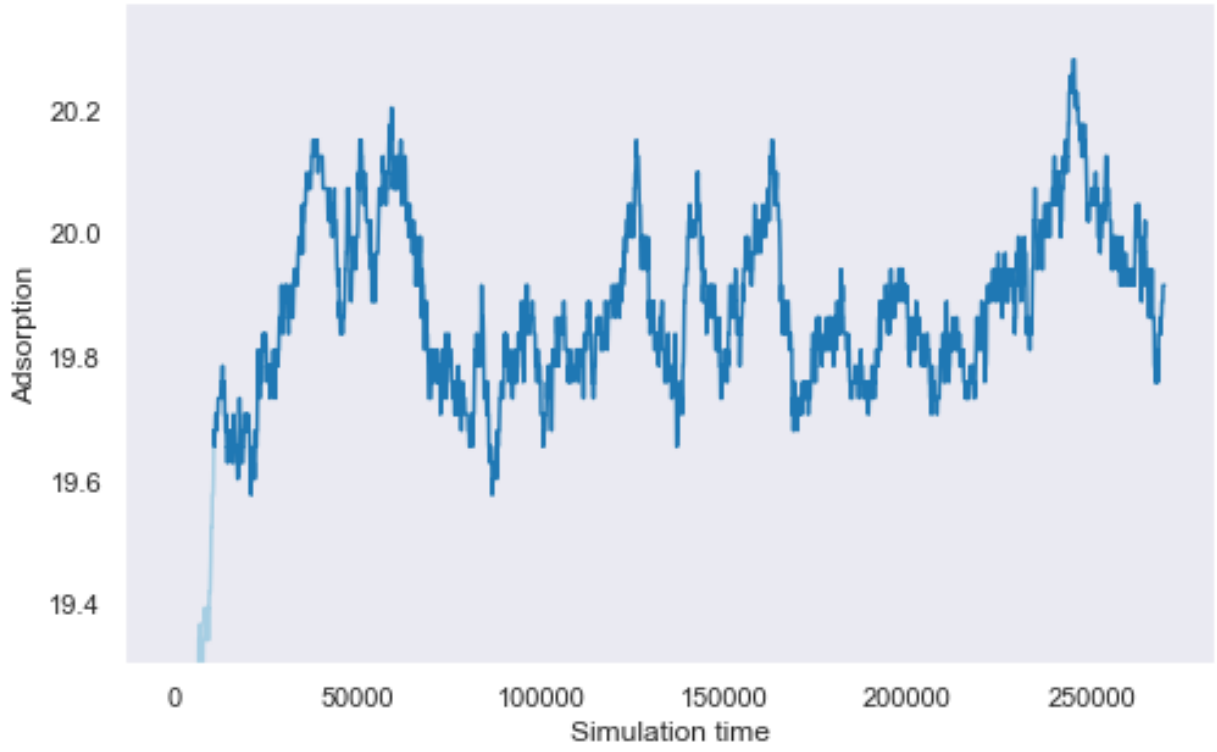


4.1.3 Finding the equilibration point

We can then combine these two pieces of information together to find where our model of the smoothed timeseries first crosses our estimate of the lower bound of the data. This interception is then proposed as the equilibration point, with all values in the timeseries after this point to be used as data.



4.1.4 Checking our assumptions

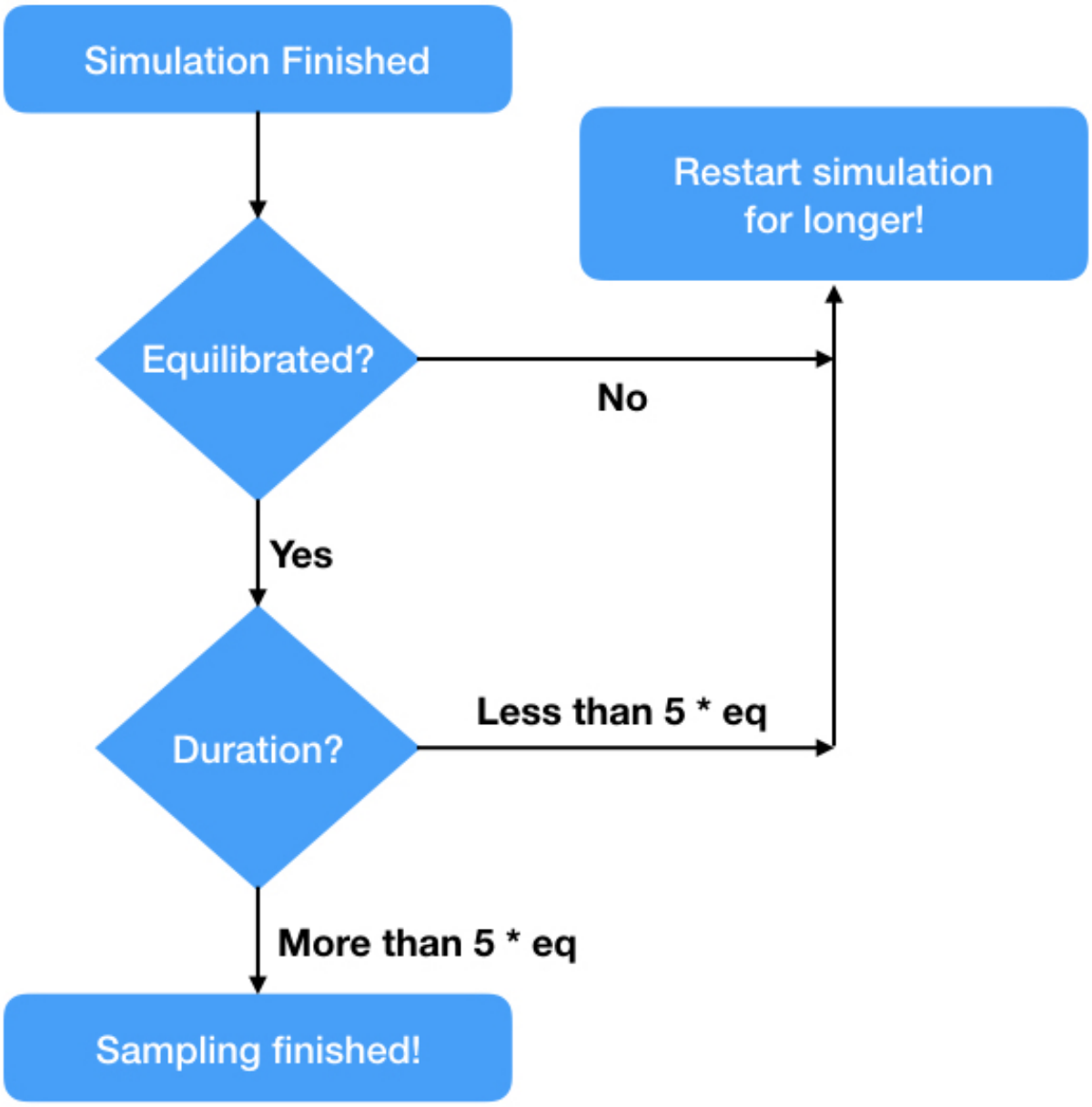


At this point we must check our hypothesis that this is the equilibration point. To do this, an augmented Dickey-Fuller test is used to check if the timeseries after the proposed equilibration point is stationary (ie flat). If the timeseries is found to have statistically significant drift, then the hypothesis is rejected and further simulation is required to gather data.

4.2 Automatic Runlength

How automatic runlength works

- once eq is found, we can measure how much data we have produced
- use heuristic of g ~~ n_eq [from Mol. Sim paper]
- sampling is defined in terms of number of g to collect
- if enough g hasn't been collected, a restart is issued



4.3 Crash Recovery

When running simulations on a HPC cluster, there is often a maximum wall time for your job, typically 1 or 2 days. However for some GCMC simulations, especially at higher pressures, this is not long enough to collect enough data. GCMCWorkflow is able to work around this problem by allowing the restart of terminated simulations.

After a RunSimulation task has been performed, the PostProcess task checks if the simulation exited correctly. If it finds that the simulation didn't exit correctly, then it ascertains how long it should have ran for, and how long the simulation actually ran for. It then parses what data was generated (so this is not wasted), then creates a new RunSimulation stage in the Workflow which will complete the remaining number of steps.

If a job was killed due to walltime constraints, the job will naturally not have been able to communicate that it was killed. These jobs will therefore appear as if they are still running, even after the job on the HPC cluster has terminated.

Manual intervention is required to find such jobs and mark them as finished; the following Fireworks command can be used:

```
``lpad detect_lostruns --fizzle``
```

This finds jobs which are “lost”, ie haven’t reported their status in a long time, and marks them as “FIZZLED”, the Fireworks term for failure.

Description of how all the magic works