A toolkit, framework and application for automating DFT calculations

1 Overview

This package provides Python objects and functions for planning and carrying out VASP calculations, handling INCAR logic, and organizing VASP runs: [1]

```
poscar.incar('opt, cluster=nersc').run().get_contcar()
      .incar('dos, cluster=nanaimo')...
```

It also provides a GUI (Figure 1).

What it doesn't do is heavy lifting. It doesn't generate adsorption structures or do machine learning. For those, specialized tools exist. Rather, it aims to facilitate a smooth, unobstructed VASP workflow, especially in places where existing solutions like `ase` or MedeA, in my experience, work awkwardly.

Its functionality is provided through 3 separate modules: the `toolkit`, the `framework`, and the `application`.

2 The toolkit

This module provides plain Python objects and functions for running VASP.

We provide a bare-bones `poscar(unit_cell, positions)`, and rely on `ase.io` for IO. `ase.Atoms` would've been perfect if it were more straightforward and `pickle`-able.

We provide a `getopt`-style `incar(opt, metal, cluster=nersc)`. Using python's `exec` built-in, the user can easily enforce custom rules on the `incar`:

```
if relaxation and metal:
    assert ismear == 1 or 2
```

To run VASP, simply call functions:

```
write_files(incar, poscar); submit(); if complete(): retrieve()

# alternatively, fluent interface
poscar.incar('opt, cluster=nersc').submit().retrieve()
```

No more `Calculator` classes with multiple inheritance. Also, `ase` somehow doesn't allow `calculator.run('nersc')`; we have `submit()` for that.

3 The framework

In software engineering, a `toolkit` provides tools, while a `framework` overarchingly modifies the working of the application.

[1] It'll work with any DFT software, but "INCAR" is shorter than "input parameters including k-mesh", and "SLURM" shorter than "supercomputing cluster queueing systems".

Python is synchronous. Your first `ase.calc.get_energy()` command will block the session for however long it takes for the VASP computation to complete, before you can enter the second command. For everyday research, that's inconvenient. It's better to separate the planning stage from the execution stage:

```
# planning
>>> poscar.config('opt, cluster=nersc').run().get_contcar()
          .config('dos, cluster=nanaimo').run()...
>>> poscar2.config('bands, cluster=auto')...

# 1 hour later
>>> step()
retrieving opt. submitting dos
```

`Tensorflow 1.x` does exactly this: define a computational graph first, then execute it. `dask.delayed`, a graph-parallel execution library, allows something similar. We borrow from their design pattern. `Tensor`s hold values, and `Operation`s link `Tensor`s to form a computational graph.

To use the `framework`, wrap objects in `Tensor`s, and functions in `Operation`s. If desired, synchronous behavior can be restored similarly to `tf.eager_execution` in `Tensorflow 2.x`, or by using the `toolkit` stand-alone.

4 The application

The graph-parallel `framework` naturally introduces a data structure for managing VASP folders: nested directed graphs. Compared to `ase`'s list of entries and Materials Studio's project folders, a graph database is both scalable and human friendly (method of loci).

Our web-based GUI (Figure 1) helps visualize the graph database, creating a persistent "look and feel" of a project, and provides access to common functionalities. Figure 2 explains its backend/frontend architecture.

The `application` is a work in progress. `LinkuriousJS` has been deprecated, the backend API changed, and the ever-increasing database size highlights communication overhead issues.

In addition to the 3 main components, `plugins` provide additional extensible python API, currently including save/load, periodic table, electronic structure post-processing (past work), and MLqueue (past work).

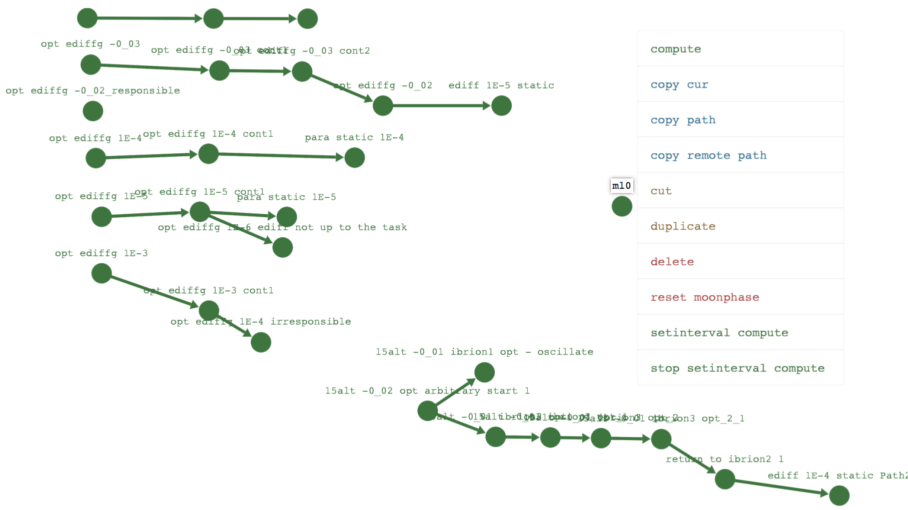

Figure 1: GUI, WIP, picture taken from 2019 version

$$\text{Framework} \xrightleftharpoons[\text{function call}]{\text{graph}} \text{Flask} \xrightleftharpoons[\text{HTTP request}]{\text{JSON}} \text{LinkuriousJS} \xrightleftharpoons[\text{mouseDown}]{\text{Visualization}} \text{User}$$

Figure 2: Data flow between the database and the UI