xtp_job_control Documentation

Felipe Zapata

Mar 07, 2019

Contents:

CHAPTER 1

Installation

To install the *xtp_job_control* library type the following command:

```
pip install git+https://github.com/votca/xtp_job_control@master
```

1.1 Requirements

the *xtp_job_control* packages assumes that you have already install the votca and that the binaries and libraries are accessible.

Note: If you have install votca in a non-standard location, export the environment variable **VOTCASHARE** with the absolute path to the *Votca shared folder*.

CHAPTER 2

Tutorial

The *xtp_job_control* library contains a set of predifined *workflows* that workout of the box. But a user may also need further capabilities over the xtp functionality, for those cases the *xtp_job_control* allows a user to extend or create some missing functionality that can be integrated with the predefined workflows.

2.1 Available Workflows

The following family of workflows are defined in *xtp_job_control*:

- dftgwbse
- transport
- kmc

2.1.1 dftgwbse

The dftgwbse workflow performs either a point energy calculation (see energy input example) or geometry optimization (see optimization input example) using the GW-BSE method, check the **GW-BSE** entry of the manual for furtner information.

2.1.2 Transport

The transport workflow contains several steps to compute charge transport networks, using a combined coarse-grained and stochastic approach (see input transport example). For further reading, see secion *2.10* of the manual.

2.1.3 kmc

The kmc worklow performs a hopping simulation of charge carriers using a kinetic Monte Carlo approach (see *input kmc* example). For further information, see Chapter *2* of the manual.

2.2 Running a workflow

A workflow is run by executing the following command in the terminal:

```
run_xtp_workflow.py --input tests/DFT_GWBSE/dftgwbse_CH4/input_dft_gwbse_CH4.
yml
```

Where **run_xtp_workflow.py** is the python script that read, process and run the workflow; and the input is a file in yaml format.

After the command finishes it returns another yaml file called result_<workflow>_<time-stamp>.yml containing a summary of the workflow results and a file called `xtp.log` with the standard output and error returned by the *Votca-XTP* calculators.

2.2.1 How it works

First, the library scan the input and checks its validity (using a set of predifined schemas), then a *dependency graph_* is built between the different jobs involved in the workflow. This graph allows to run in parallel those jobs that do not dependent on each other, while creating explicit dependencies between jobs that need to run in a sequential mode, injecting the ouput of one job as input of the next one. Finally, the jobs are running in different folders while the dependecies between them are automatically track.

Both the construction and execution of the dependency graph is carried out by the Noodles library. When the `run_xtp_workflow.py` command is invoked, Noodles traverses the graph of job dependencies and checks against its internal database for a reference to the job results, if such reference does not exist the job is executed and the resulting output metainformation is stored in the database.

If the execution of the workflow is stopped by the user or fails for technical reasons, the generated database with metadata can be used to restart the workflows. _Noodles will walk through the dependencies tree in the same way as when started from scratch, but will query the database for already existing results and execute only the tasks that were not yet successfully completed.

2.3 Votca calculators options

The arguments and default values for running simulations with *Votca-XTP* are define in different xml files, leaving at the *VOTCASHARE* folder. When an *XTP* command is invoked, *Votca-XTP* reads from these *xml* files the available values. Since, *xml* files can have nested *xml* files it is a non-trivial task to setup correctly the simulation values for a given simulation.

In order to improved the aforemention situation, the *xtp_job_control* library allows the users to create a section called **votca_calculators_options** in the input file. Every subsection on it, corresponds to an xml file and subsequently subsections are values that the user wish to change. For example, see the next snippet taken from the input example for a single point energy calculation: .. code-block:: yaml

votca_calculators_options:

dftgwbse:

dftpackage: xtpdft.xml

xtpdft: threads: 1

It saids that in the *dftgwbse* xml option file, the argument `dftpackage` must be set equal to *xtpdft.xml*. While in the *xtpdft* xml option file, the number of `threads` should be set to 1.

2.3.1 How it works

Before the jobs are executed, all the Option files in the *VOTCASHARE* folder are copy to a temporary folder. These temporary files are combined with **votca_calculators_options** provided by the users, generating a new set of files containing the options to call the *Votca-XTP* functionality.

CHAPTER 3

Workflow components

As mentioned in the tutorial, Noodles is the workflow engine used both to create the *dependency graph* between the jobs and to run such graph. Noodles provides a *python decorator* call `schedule` that when applied to a function or method returns a *promise* or *future* object (see noodles schedule tutorial).

The *xtp_job_control* library, wraps the different *XTP* calculators into their own functions decorated with `schedule`. These *scheduled* functions can then be organized in different workflows by injecting the output of one function as the input of another function. For example, the *single point energy* workflow is implemented like:

```
def dftgwbse_workflow(options: dict) -> object:

 # create results object
 results = Results({})

 # Run DFT + GWBSE
 results['dftgwbse'] = run_dftgwbse(results, options)

 # Compute partial charges
 results['partialcharges'] = run_partialcharges(
     results, options, promise=results["dftgwbse"]["system"])

 output = run(results)
```

In the previous example, both functions `run_dftgwbse` and `run_partialcharges` are *scheduled* functions implemented in the xtp_workflow module. `Results`, is a subclass of the Python dictionary extended with functionality to handle the jobs booking.

Notice that jobs are stored as nodes in the `results` dictionary and also the *promised* object `results['dftgwbse']` contains a *system* property that can be passed to a *partial charges* calculator.

The resulting *dependency graph* in this particular case, contains two nodes, one for each job and a edge representing the system dependency.

3.1 runner

The `run` function in the previous snippet is implemented in the runner module and encapsulate the noodles details. Noodles offers a different variety of runner for different architectures and purposes (see runners). Currently, the *xtp_job_control* library use a parallel multithread runner with an sqlite interface for storing the jobs metadata.

CHAPTER 4

Creating Your Own Workflow

if the available workflows do not provided simulation that you want to perform, you can create your own worklow by glueing together the available functions at the xtp_workflow.

If non of the functions at the xtp_workflow modules satifies your needs, you can create your own function using the *xtp_job_control.workflows.workflow_components.call_xtp_cmd()* and *xtp_job_control.workflows.workflow_components.call_xtp_cmd()*. The following code snippet, ilustrates the creation of a call to the `xtp_map` command using a promised *system* argument provided by another job called *job_system*.

```
 results = Results({})

 # Other jobs executed here
 ...

args = create_promise_command(
    "xtp_map -t {} -c {} -s {} -f {}",
    topology, trajectory,
    results['job_system']['system'], path_state)

results['job_state'] = call_xtp_cmd(args, workdir, expected_output={'state': 'state.
↪sql'})
```

the `expected_output` argument in the function, search for output files created by the command. In the current case, the `xtp_map` command generates a file called *state.sql*. The ouput files can be access by other jobs using the name provided in the dictionary. For example, the *state.sql* is available using the following notation:

```
state_file = results['job_state']['state']
```

4.1 Command line wrappers

The following functions create an schedule call to a *Votca-XTP* command.

xtp_job_control.workflows.workflow_components.**call_xtp_cmd**(*cmd: str*, *workdir: str*, *expected_output: dict = None*)

(scheduled) Run a bash *cmd* in the *workdir* folder and search for a list of *expected_output* files.

xtp_job_control.workflows.workflow_components.**create_promise_command**(*string: str*, **args*) → str

(scheduled) Use a *string* as template command and fill in the options using possible promised *args*

CHAPTER 5

XML Editor

The *xtp_job_control* library offers different funcionality to edit and manipulates the entries of the *xml* option files.

5.1 XML editing function

Funcionality to edit the content of the *xml* option files for *Votca-XTP*.

xtp_job_control.xml_editor.**edit_xml_options**(*sections: dict, path_optionfiles: pathlib.Path*) → Dict[KT, VT]

Go through the *options* file: sections dictionary and edit the corresponding XML file by replacing *sections* in the XML file.

xtp_job_control.xml_editor.**edit_xml_file**(*path: str, xml_file: str, sections: Dict[KT, VT]*) → str

Parse the *path* XML file and replace the nodes given in *sections* in the XML tree. Finally write the XML tree to the same file

xtp_job_control.workflows.workflow_components.**edit_options**(*options: Dict[KT, VT], names_xml_files: List[T], path_optionfiles: str*) → Dict[KT, VT]

(scheduled) Edit a list of XML files *names_xml_files* that are located in the *path_optionfiles* using a set of user-defined *options*.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

C

E