
Frazier-PipeLine Documentation

Release 0.0.1

Henry Herbol

Oct 25, 2016

CONTENTS

1	Frazier PipeLine	3
1.1	Installing	3
1.2	Documentation	3
1.3	Using FPL	4
2	Indices and tables	7
	Python Module Index	9
	Index	11

Contents:

FRAZIER PIPELINE

Pipeline for submitting solubility simulations. Bonus output may include Unsaturated Mayer Bond Order amongst other things.

1.1 Installing

Currently installation involves first installing clancelot:

```
[user@local]~% cd ~; git clone git@github.com:clancylab/clancelot2.0.git
```

And then cloning this project:

```
[user@local]~% cd ~; git clone git@github.com:hherbol/frazier-pipeline.git
```

Note, you'll also have to append the frazier-pipeline/pys folder to your PYTHONPATH variable.

```
[user@local]~% echo '\nexport PYTHONPATH="/PATH/TO/FRAZIER/PIPELINE/pys:$PYTHONPATH"'  
↪>> ~/.zshrc
```

1.2 Documentation

Documentation is necessary, and the following steps MUST be followed during contribution of new code:

Setup

1. Download [Sphinx](#). This can be done simply if you have [pip](#) installed via `pip install -U Sphinx`
2. Wherever you have *frazier-pipeline* installed, you want another folder called *frazier-pipeline-docs* (NOT as a subfolder of *frazier-pipeline*).

```
[user@local]~% cd ~; mkdir frazier-pipeline-docs; cd frazier-pipeline-docs; git clone  
↪-b gh-pages git@github.com:hherbol/frazier-pipeline.git html
```

3. Forever more just ignore that directory (don't delete it though)

Adding Documentation

Documentation is done using [ReStructuredText](#) format docstrings, the [Sphinx](#) python package, and indices with autodoc extensions. To add more documentation, first add the file to be included in *docs/source/conf.py* under *os.path.abspath('example/dir/to/script.py')*. Secondly, ensure that you have proper docstrings in the python file, and finally run *make full* to re-generate the documentation and commit it to your local branch, as well as the *git gh-pages* branch.

For anymore information on documentation, here is a link to the followed [sphinx_tutorial](#).

1.3 Using FPL

Automated calculations for the Frazier Pipeline are as follows:

- `get_MBO()`
- `get_UMBO()`

The following is still prone to bugs. It should work for `num_solvents=1`; however, any more and it is prone to blowing up (ie, irrationally large enthalpy of solvations). For now, it has been hidden to avoid incorrect calculations.

- `get_enthalpy_solvation()`

For a quick start guide, in theory the following will suffice:

```
import fpl_auto

halides = [ "Cl", "Br", "I" ]
cations = [ "MA", "FA", "Cs" ]

# Due to not having all the solvent densities and dielectrics in
# fpl_constants, only the following are currently available.
solvents = ["acetone",
            "gbl",
            "DMF",
            "dmso",
            "nmp",
            "THTO"]

# Note, you can also do mixed halides by passing a list of halides
# such as h = ["Cl", "Cl", "I"]. In this example though, we just
# have h = "Cl" which is equivalent to h = ["Cl", "Cl", "Cl"]
h, c, s = halides[0], cations[0], solvents[0]

# Get the UMBO of the Oxygen-Carbon bond in a single Acetone molecule
# adsorbed onto a solute of PbCl3MA
umbo = fpl_auto.get_UMBO(h,c,s)
```

Note, parallelization has also been implemented and can be utilized as follows:

```
# Imports
import fpl_auto

# Variables
halide = "I"
cation = "MA"

# Loop through and generate "jobs"
running_jobs = []
for solvent in ["DMF", "DMSO"]:
    for ion in ["Pb", "Sn"]:
        for route_lvls in [1,2]:
            running_jobs.append(fpl_auto.get_
→UMBO(halide,cation,solvent,ion=ion,route_lvls=route_lvls,on_queue=True))

# Wait for all jobs to finish
```



```
print("Jobs have been submitted. Waiting for completion...")
for job in running_jobs:
    job.wait()

# Read in output from each job
for job in running_jobs:
    print("For job %s, the umbo is %lg" % (job.name, job.umbo()))
```

Other features of the `cancelot Job` class apply here as well. That is, you should also be able to poll if the job has finished using `is_finished()` instead of hanging the way `wait()` does.

```
fpl_auto.get_MBO(halide, cation, solvent, ion='Pb', num_solvents=1, route_lvls=1, avg=True, criteria=[['O', 'C'], ['O', 'N'], ['O', 'S']], on_queue=False, queue='batch', nprocs=1, xhost=None)
```

Get the mayer bond order. The Mayer Bond Order (see `get_UMBO()`) for more details.

Parameters

- halide:** *str* The halide within the perovskite.
- cation:** *str* The cation within the perovskite.
- solvent:** *str* The solvent of the system.
- ion:** *str, optional* The ion of the perovskite. By default this is Pb, but can also be Sn.
- num_solvents:** *int, optional* The number of solvents to model explicitly (implicit is always on in background).
- route_lvls:** *int or list, int, optional* The level of theory to use. If an integer is passed, all levels are assumed the same.
- avg:** *bool, optional* Whether to average together all UMBO's that match the given criteria.
- criteria:** *list, list, str, optional* A list of lists, each list holding a list describing what bonds you want the UMBO for. By default, it is every bond with an oxygen atom involved.
- on_queue** [*bool, optional*] Whether to run this simulation on the queue.
- queue** [*str, optional*] Which queue to run the simulation on.
- nprocs** [*int, optional*] How many processors to use.
- xhost** [*list, str or str, optional*] A list of processors, or a single processor for which to submit the simulations (on the queue).

Return

- MBO:** *list, float, or float* The Mayer Bond Order. If avg is False and there are more than one MBO matching criteria, a list is returned.

```
fpl_auto.get_UMBO(halide, cation, solvent, ion='Pb', offset=2.0, num_solvents=1, route_lvls=1, avg=True, criteria=[['O', 'C'], ['O', 'N'], ['O', 'S']], on_queue=False, queue='batch', nprocs=1, xhost=None)
```

Get the unsaturation (average?) mayer bond order. The Mayer Bond Order (MBO) is well described [here](#). In short, it is a numerical representation of the probability of how many electrons partake in a bond. For instance, a single bond would have a theoretical bond order of 1.0; however, in practice it may have more or less depending on how electrons distribute across the molecule. The MBO helps describe this, and the Unsaturated MBO (UMBO) helps represent this in a more understandable fashion. That is, if the UMBO is larger than zero, the bond is weaker than theory. If the UMBO is less than zero, then the bond is stronger than theory.

Parameters

halide: *str* The halide within the perovskite.

cation: *str* The cation within the perovskite.

solvent: *str* The solvent of the system.

ion: *str, optional* The ion of the perovskite. By default this is Pb, but can also be Sn.

num_solvents: *int, optional* The number of solvents to model explicitly (implicit is always on in background).

route_lvls: *list, int, optional* The level of theory to use.

offset: *float, optional* The offset supplied to get the UMBO. In most cases we consider, this is 2.0 as that is the theoretical bond order of a double bonded oxygen to sulfur.

avg: *bool, optional* Whether to average together all UMBO's that match the given criteria.

criteria: *list, list, str, optional* A list of lists, each list holding a list describing what bonds you want the UMBO for. By default, it is every bond with an oxygen atom involved.

on_queue [*bool, optional*] Whether to run this simulation on the queue.

queue [*str, optional*] Which queue to run the simulation on.

nprocs [*int, optional*] How many processors to use.

xhost [*list, str or str, optional*] A list of processors, or a single processor for which to submit the simulations (on the queue).

Return

UMBO: *list, float, or float* The Unsaturation Mayer Bond Order. If avg is False and there are more than one UMBO matching criteria, a list is returned.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

f

fpl_auto, 4

F

`fpl_auto` (module), 4

G

`get_MBO()` (in module `fpl_auto`), 5

`get_UMBO()` (in module `fpl_auto`), 5