ESTEEM

Release 0.0.1

Dec 14, 2021

Contents:

ESTEEM is an open source package for calculations of excited state and spectroscopic properties of molecules in solvent environments, developed by Nicholas Hine at the University of Warwick.

It is built upon the python framework provided by the Atomic Simulation Environment and can call upon a number of different electronic structure and molecular dynamics codes, and machine-learning techniques.

The project is currently at a pre-release stage but I plan to release it soon with full documentation. The code's git repository on bitbucket contains the latest development version and can be used to report any issues.

The Explicit Solvent methodology that has been adapted to a workflow in ESTEEM is based on the ideas described in the papers below. If you find this project useful, I would appreciate if you cite these works:

> M.A.P. Turner, M.D. Horbury, V.G. Stavros, N.D.M. Hine, Determination of secondary species in solution through pump-selective transient absorption spectroscopy and explicit-solvent TDDFT, J. Phys. Chem. A 123, 873 (2019).
>
> T.J. Zuehlsdorff, P.D. Haynes, M.C. Payne, and N.D.M. Hine, Predicting solvatochromic shifts and colours of a solvated organic dye: The example of Nile Red, J. Chem. Phys. 146, 124504 (2017)

I hope to make ESTEEM installable via pip soon. This means you would be able to install with just:

```
$ pip3 install esteem
```

But this is not available yet.

Acknowledgements: Some parts of ESTEEM are based on earlier python scripting by members of my research group including Tim Zuehlsdorff, David Turban and Matt Turner.

Manual:

CHAPTER 1

Introduction and Background

An excellent introduction to the use of "explicit" representation of a solvent environment in the context of theoretical spectroscopy calculations can be found in Tim Zuehlsdorff and Christine Isborn's 2019 review:

Modeling absorption spectra of molecules in solution, Int J. Quant. Chem. 119, e25719 https://onlinelibrary.wiley.com/doi/full/10.1002/qua.25719

ESTEEM is an attempt to harness the flexibility provided by ASE, and the ease with which one can create "calculators" to perform atomistic simulation tasks, and use it to provide an API for explicit solvent calculations, including setting them up, running them on HPC systems, and analysing and plotting the results.

1.1 Explicit Solvent Calculations

The main explicit solvent code works on the basis of a set of predefined tasks representing steps in the workflow of a calculation of a solute in explicit solvent. These tasks are:

1. Calculate the structure and electronic excitations of a solute molecule in gas-phase and/or implicit solvent.
2. Calculate the structure (and excitations, if required) of the solvent molecules as above.
3. Solvate the solute molecule, i.e. surround it by a realistic, well-equilibrated representation of the solvent in a large periodic box.
4. Take a series of snapshots of the geometry of the solvated system during a long Molecular Dynamics trajectory, separated by a time interval long enough to decorrelate the snapshot geometries.
5. Extract clusters from the snapshots of the solvated system, centered on the solute molecule, containing the solute molecule and all solvent molecules within a certain distance of the solute.
6. Perform a Theoretical Spectroscopy (eg TDDFT) calculation on each extracted cluster. The size may necessitate the use of Linear-Scaling TDDFT.
7. Optionally, process the solute spectra generated in step 1 to determine a "spectral warp" mapping the result of one spectroscopy calculation (usually with a computationally-inexpensive level of theory) onto another (a state-of-the-art calculation with a high level of theory).

8. Produce a final predicted spectrum for the solvated system, by averaging over the extracted clusters. The spectral warp can be applied to each snapshot. This spectrum can be used to predict the colour of the molecule in solvent, or to identify what is present in a mixture.

In ESTEEM, steps 1. and 2. are performed by the 'Solutes' module, steps 3. and 4. are performed by the 'Solvate' module, steps 5. and 6. are performed by the 'Clusters' module, and steps 7. and 8. are performed by the 'Spectra' module.

A workflow for ESTEEM usually takes the form of a single python script which both defines the calculation parameters and calls driver routines which perform the above tasks.

1.2 Machine Learning Potential Energy Surfaces

The ESTEEM code also has functionality for machine learning of potential energy surfaces of molecules. This is still under development and not so extensively documented at the present time.

CHAPTER 2

Installation

ESTEEM is a python-based project, originally written within Jupyter notebooks, and is tightly integrated with the Atomic Simulation Environment (ASE). This integration means that a range of electronic structure, molecular dynamics and machine-learning packages with calculators available in ASE can be used by ESTEEM to perform its tasks. ASE functionality is accessed via Wrapper modules which create default settings for a package. These packages are each individually optional, though there is little that can be done without at least one MD code and one electronic structure code.

Apart from the calculators, the code has the following dependencies:

- Python, version 3.6 or above
- ASE
- NumPy
- Matplotlib for graphical outputs

Compiling the Python files from the ipynb notebooks requires IPython (though this may be dropped in future).

2.1 Manual installation

Currently the best way to install the code is to clone the git repository. In future I hope to make a PyPI package so that "pip install esteem" will work.

Install ASE.

It is highly advisable to use an up-to-date version of ASE. Many calculator bugs in relevant calculators have been fixed in versions 3.18-3.20 so older versions may well fail. This should be done before installing ESTEEM as currently ESTEEM will attempt to apply a tiny bugfix to ASE.

There are instructions at the ASE website for installing ASE. The following commands can be used to verify that ASE and other dependencies can be loaded successfully:

```
$ python3
>>> import ase
>>> import numpy
>>> import scipy
>>> import matplotlib
```

Clone the Bitbucket repository.

ESTEEM is still evolving, so releases are not yet numbered. Versions can be identified by their commit string.

The latest version of the code can be downloaded via the project's bitbucket page., but to ensure updates can be applied easily, it is better to clone the repository with git. Check out the code with:

```
$ git clone https://ndmhine@bitbucket.org/ndmhine/esteem.git
```

Starting in whatever path you wish to install to. This will create a directory called 'esteem'.

Convert notebooks into pure python

After you have downloaded the code, the fastest way to do this conversion is via the 'setup.py' script:

```
$ python setup.py
```

If that works, you should have a set of .py files in the esteem directory.

If the code changes, downloading updates can be achieved with:

```
$ git pull
```

after which you need to compile the notebooks again with the setup.py script.

Set the environment

You need to let Python know where to find the ESTEEM modules. The following line can be added to your '.bashrc' (or anywhere else that gets run automatically), with the appropriate path substituted for '~/' if required:

```
$ export PYTHONPATH=~/esteem:$PYTHONPATH
```

To check this works, start python and type the below command, checking that the location listed by the second command is what you expect:

```
>>> import esteem.drivers
>>> print(esteem.drivers.__file__)
```

CHAPTER 3

Using ESTEEM

There are two main modes of using ESTEEM: via the command-line tools, and via a script or notebook that imports the python driver modules. Both are useful under different circumstances: the command-line tools allow you to explore various options to discover what works, whereas a script allows you to create a documented, reproducible, adjustable workflow.

The overall model is: *Solutes module* -> *Solvate module* -> *Clusters module* -> *Spectra module*.

The use of spectral warping will require the *Solutes module* module to have been run twice, with different levels of theory (eg different XC functionals).

3.1 Command-line tools

Command-line use is helpful for setup and testing: naturally you must not run any large calculations on the login node of your HPC cluster! The scripts of ESTEEM can be invoked with commands such as this:

```
$ python ~/esteem/esteem/tasks/solutes.py --namefile my_solutes --basis 6-311++G** --
↪func PBE0
```

where `my_solutes` refers to a file which contains a list of the solutes to process. `solvate` can be replaced by any of the other tasks, namely `solvate`, `clusters`, `spectra` (as well as `qmd_trajectories`, `ml_training`, `ml_testing`, `ml_trajectories` for the Machine-Learning part of the package). If you intend to use this a lot, it might be simpler to make everything in `~/esteem/esteem/tasks/` executable with `chmod +x ~/esteem/esteem/tasks/*.py` and add this directory to your `$PATH`. That way you can simplify the above to:

```
$ solutes.py --namefile my_solutes --basis 6-311++G** --func PBE0
```

Help on the arguments for each of the commands can be found in the documentation or by invoking the command-line help text:

```
$ python ~/esteem/esteem/tasks/solvate.py --help
```

which gives a listing of all the available input variables and their default values.

Refer to the bash script in `/examples` to see an example of how to use the command line tools to perform steps of a calculation. It would be pperfectly possible to run a whole set of calculations this way, but the reality is that explicit solvent calculations are both computationally demanding, and demanding of human attention and intervention. If you find you need to repeat some part of the calculation, it would be fiddly to reproduce the whole chain of steps correctly.

In complex cases, there may be many combinations of solute and solvent to investigate, or you may need to perform a range of calculations for different combinations of XC functional, basis, various radii and other sizes, MD temperature etc etc. This is where writing a workflow script becomes highly preferable, and is the main use case of ESTEEM for generating publishable results.

3.2 Scripting with ESTEEM

Creating a workflow script or notebook to use ESTEEM in a semi-automated fashion only requires a small amount of coding. Even so, it may be preferable to start from one of the examples and modify it appropriately. The basic idea is that a single python script sets up all the calculation parameters, then picks which job to perform based on command line arguments. This script is also capable of writing HPC job submission scripts for SLURM or PBS which invoke specific tasks in parallel. Array jobs are particularly useful in this context, as will be discussed later.

Let's say we want to define the solute to be catechol and set up calculations in explicit cyclohexane and methanol, and put the workflow in a script called `cate.py`. The necessary components of a workflow script or notebook are as follows:

First we define a dictionary whose keys are short-form names of the solutes you want to use. It is advisable to keep the names in here short, as otherwise filenames become very long. The entries for each key are long-form names for labelling plots and other outputs. If you wish the molecular structures to be downloaded from an existing database (see `get_xyz_files` in `solutes`), these can be full IUPAC names:

```
all_solutes = {'cate': 'catechol'}
```

Next we define a similar dictionary for the solvent molecules:

```
all_solvents = {'cycl': 'cyclohexane', 'meth': 'methanol'}
```

We import the top level 'drivers' module of ESTEEM:

```
from esteem import drivers
```

We generate a set of default arguments for each of the main tasks of the code, in `args` objects:

```
solutes_args, solvate_args, clusters_args, spectra_args = drivers.get_default_args()
```

We can now modify the members of each of these `args` according to our needs. Their members are the same as the command line arguments of the scripts as discussed above. For example, here we set the basis and functional for the Solutes task, choose a small box for the solvate task, and a small cluster radius for the clusters task:

```
solutes_args.basis = '6-311++G**'
solutes_args.func = 'PBE0'
solvate_args.boxsize = 15
clusters_args.radius = 3
```

We initialise any wrapper settings by calling setup routines for the wrappers we need (these may become classes in the near future). Here we are using Amber for MD, and NWChem for DFT and TDDFT:

```
from esteem.wrappers import amber
solvate_wrapper = amber.AmberWrapper()
```

(continues on next page)

(continued from previous page)

```
from esteem.wrappers import nwchem
nwchem.nwchem_setup()
solutes_wrapper = nwchem.NWChemWrapper()
from esteem.wrappers import onetep
clusters_wrapper = onetep.OnetepWrapper()
```

We set up choices enabling the code to write job scripts defining appropriate parallelisation (templates for machines in use by my group are provided in drivers, and can be copied and modified as required):

```
make_script = drivers.make_sbatch
solutes_script_settings = drivers.nanosim_1node
solvate_script_settings = drivers.nanosim_1node
clusters_script_settings = drivers.nanosim_1node
# make any changes you need to the script settings here
```

Finally, we invoke the top-level driver of ESTEEM, which unfortunately has a rather long argument list concatenating all the things we have just set up:

```
drivers.main(all_solutes,all_solvents,
             solutes_args,solvate_args,clusters_args,spectra_args,
             solutes_wrapper,solvate_wrapper,clusters_wrapper,
             make_script,solutes_script_settings,solvate_script_settings,clusters_
↪script_settings)
```

This top-level driver then works out, from the rest of the command-line arguments with which the script was invoked, what part of the calculation to invoke. For example, a script to invoke the solutes task for catechol (shortname `cate`) One of the most useful tasks is 'scripts', which writes HPC submission scripts for all the other tasks.

Let's say I named my top-level script 'cate.py', I would write scripts with:: $ python cate.py scripts

Then submit the resulting scripts to my queuing system, for example:: $ sbatch cate_solutes_sub

Internally, this job script will invoke the top-level script again, with a specific task:: $ python cate.py solutes cate

which runs the Solutes task.

After the invocation of the top-level driver, you might want to have the your script quit python:: exit()

so that in the same file you can define other post-processing functions or interactive cells in a notebook which require access to the workflow settings, but which you do not want to run every time the script is invoked.

3.3 Providing multiple sets of arguments

In the example above we provided a single set of arguments for each of the tasks. However, we might want to run with several sets of options, for example to investigate convergence with respect to basis size, check different XC functionals, run MD at different temperatures, or converge the cluster excitations with respect to cluster radius.

To enable this sort of study, if any of the argument lists to Drivers.main are python dictionaries, then the command-line argument `target` decides which entry in that dictionary to use. The scripts task will write a job script for each possible `target` in the list appropriate to each task.

It is recommended to make a set of "master" arguments first, copy that list and change what you need. The routine deepcopy is useful for this:

```
from copy import deepcopy
all_clusters_args = {}
for rad in [0,3,6,9]:
    target = f'solvR{rad}'
    all_clusters_args[target] = deepcopy(clusters_args)
    all_clusters_args[target].radius = rad
```

Then you simply pass 'all_clusters_args' rather than 'clusters_args' in the call to drivers.main()

3.4 Example Scripts

The package contains several example scripts of increasing complexity to illustrate the functionality of the code:

- `cate.py` - a simple script as described above to run solvated catechol in cyclohexane and methanol
- `cate_full.py` - a more complex script that includes PBE and PBE0 solutes calculations, followed by a test of different solvent shell radii, and a spectral warp of the cluster results from PBE to PBE0.

The page on *Examples of using ESTEEM* gives a walkthrough of how to submit these to a computing cluster with SLURM.

3.5 Locating binaries for packages

By default, ESTEEM assumes that any dependency codes have been loaded via a module system, and are available by their standard executable names from the command line, for example "nwchem", "onetep", "sander" to invoke NWChem, ONETEP and Amber's sander tool, respectively.

If this is not the case, for example if you have installed the codes yourself, you can adjust names of the executables when you setu

```
>>> onetep.onetep_setup(onetep_cmd='~/onetep/bin/onetep.archer2',mpirun='srun',
↪set_pseudo_path='~/NCP17_PBE_OTF/',set_pseudo_suffix="_NCP17_PBE_OTF.usp")
```

More details can be found in the documentation pages for each wrapper.

CHAPTER 4

Parallel Execution

Most HPC clusters run a job queue, usually based on either SLURM or PBS. ESTEEM can write job scripts for either of these, but I use SLURM on most of the machines I have access to so this is likely to be better-tested.

All the job scripts created by ESTEEM write to a log, with the name `<seed>_<task>_<array_id>.log`, for example `cate_solvate_3.log` for the 3rd job of the solvate task from the `cate` workflow script. This output is mirrored to the SLURM/PBS output file.

After running the `scripts` task for your workflow, you will see a different script for each combination of task and `target` (ie argument set). Each of these scripts is capable of running that task for all of the solute molecules, or all of the combinations of solute and solvent (as appropriate to the task). If you submit the job script with no array job specification, all the tasks will run sequentially, looping over solutes (and solvents, if applicable to the task).

4.1 Identifying array task IDs

Let's say we have 2 different solute molecules, and 3 possible solutes:

```
all_solutes = {'cate': 'catechol', 'tert': '4-tert-butylcatechol'}
all_solvents = {'watr': 'water', 'cycl': 'cyclohexane', 'meth': 'methanol'}
```

These become zero-indexed lists when the script is run, so within the `solutes` task, array task ID 0 refers to `cate`, and task ID 1 refers to `tert`.

In `solvate`, the number of possible task IDs is the product of the solutes and solvents, i.e. 6 here. Task ID 0 is `cate_watr`, task ID 1 is `cate_cycl`, 3 is `tert_watr` etc, up to 5, `tert_meth`.

In `clusters`, one first runs a top-level setup job, then the actual job script for a calculation is submitted from a directory that the top-level job has set up for you, which will have the name '{solute}_{solvent}_{exc_suffix}'. Therefore the array task ID's refer to which cluster index is to be calculated. These should range from `0` to `solvate_args.nsnaps - 1` as they are zero-indexed. If you have, say, 200 snapshots, you might wish at first just to run every 10th snapshot.

4.2 Array job submission

Array jobs, which are supported by most modern queuing systems, are very useful for many of the tasks in ESTEEM.

The syntax for array job for a range of job numbers in SLURM is (here we are running the `solvate` task for all 6 of our combinations of solute and solvent):

```
$ sbatch --array=0-5 cate_solvate_sub
```

To run a maximum of 3 tasks at a time, you can append `%3` to the range:

```
$ sbatch --array=0-5%3 cate_solvate_sub
```

To run every 10th entry among all the extracted `clusters` jobs between 0 and 90, we can append `:10` to the range:

```
$ sbatch --array=0-90:10 cate_clusters_sub
```

Most of the tasks in ESTEEM are set up to automatically resume a half-finished run, so if some jobs have timed out and failed, you may want to restart a specific selection. Let's say we wanted to restart jobs 3,5 and 7 of the `solutes` task:

```
$ sbatch --array=3,5,7 cate_solutes_sub
```

4.3 Predefined script settings

If you prefer, you can write your own HPC job script and invoke your workflow script, which should then launch parallel jobs inheriting the number of nodes, number of processes per node and number of cores per process from the job environment.

However, ESTEEM also comes with some predefined parallelisation setups that can can be supplied to the drivers from your workflow script. At present these represent the machines I have access to as of 2020. I am happy to add new definitions to this list for regular users of the code.

For example, if I wanted to launch on the `nanosim` queue on the University of Warwick's SCRTP Cluster of Workstations, I could use the `drivers.nanosim_1node` predefined settings dictionary. The definition is in `drivers.py`:

```
nanosim_1node = {'account': 'phspvr',
                 'partition': 'nanosim',
                 'nodes': 1,
                 'ntask': 24,
                 'ncpu': 1,
                 'time': '48:00:00',
                 'mem': '2679mb'}
```

If I wanted to tweak something (let's say the run time and the account name), I could do:

```
script_settings = deepcopy(nanosim_1node)
script_settings['time'] = '24:00:00'
script_settings['account']: 'mstdjh'
```

Existing definitions insider drivers.py include `athena_1node`, `athena_4node`, `athena_10node`, `nanosim_1node`, `nanosim_1core`, `archer2_1node`. Please feel free to modify these or write your own. A list of arguments understood by the `make_sbatch` routine can be found in its entry in the `drivers` documentation.

CHAPTER 5

Examples of using ESTEEM

In *Using ESTEEM* we wrote the following short script, `cate.py`, to run catechol in water:

```
from esteem import drivers
from esteem import parallel
from esteem.wrappers import nwchem, amber, onetep

# List solutes and solvents and get default arguments
all_solutes = {'cate': 'catechol'}
all_solvents = {'cycl': 'cyclohexane', 'meth': 'methanol'}
solutes_args, solvate_args, clusters_args, spectra_args = drivers.get_default_args()

# Some simple overrides for a quick job
solutes_args.basis = '6-31G'
solutes_args.func = 'PBE'
solutes_args.directory = 'PBE'
solvate_args.boxsize = 18
solvate_args.ewaldcut = 10
solvate_args.nsnaps = 20
clusters_args.radius = 3

# Setup parallel execution of tasks
make_script = parallel.make_sbatch
solutes_script_settings = parallel.athena_1node
solvate_script_settings = parallel.athena_1node
clusters_script_settings = parallel.athena_1node
solutes_wrapper = nwchem.NWChemWrapper()
solvate_wrapper = amber.AmberWrapper()
clusters_wrapper = onetep.OnetepWrapper()
solutes_wrapper.setup()

# Run main driver
drivers.main(all_solutes,all_solvents,
             solutes_args,solvate_args,clusters_args,spectra_args,
             solutes_wrapper,solvate_wrapper,clusters_wrapper,
```

(continues on next page)

(continued from previous page)

```
            make_script,solutes_script_settings,solvate_script_settings,
            clusters_script_settings)
exit()
```

The following section addresses how we would use that script to run a complete explicit solvent workflow.

First, we generate job submission scripts:

```
$ python cate.py scripts cate
```

`scripts` is the 'task', and `cate` is the 'seed' for the calculation (which should match the name of the script).

This will produce five job submission scripts: `cate_solutes_sub`, `cate_solvents_sub`, `cate_solvate_sub`, `cate_clusters_sub` and `cate_spectra_sub`.

5.1 Solutes and Solvents Tasks

Let's run the first two at the same time. I will assume we are using SLURM rather than PBS throughout this example:

```
$ sbatch cate_solutes_sub
$ sbatch cate_solvents_sub
```

As directed in the script above, this will launch the Solutes task for the list of solutes (1 entry: `cate`) and the list solvents (2 entries: `cycl` and `meth`).

These will create a new directory `PBE_6-31G` and run NWChem geometry optimisation and TDDFT calculations in there, first in gas phase then in implicit solvent. If your cluster does not allow `wget` to access the Chemical Structure Resolver, you will need to supply initial guess structures in `PBE_6-31G/xyz` for all three molecules. The output file for the geometry optimisations will be `PBE_6-31G/geom/cate/geom.nwo` and the resulting structure will be in `PBE_6-31G/opt`. Calculations will be repeated in water using the COSMO implicit solvent model to produce the final geometry `PBE_6-31G/is_opt_watr`.

5.2 Solvate Task

We now want to run the MD task. If we just ran:

```
$ sbatch cate_solvate_sub
```

This would launch the Solvate task twice, one after the other, for the two different solvents. A more likely scenario is that we want to run these as two separate jobs, so we can use an array task:

```
$ sbatch --array=0-1 cate_solvate_sub
```

Which launches two jobs, one for catechol in cyclohexane, one for catechol in ethanol. The results will go in separate directories, `cate_cycl_md` and `cate_meth_md`. There are many output files in each directory, for the different steps of setup and different MD runs, but the most important ones are the 'trajectory' of snapshots: `cate_cycl_solv.traj` and `cate_meth_solv.traj` which get used by the next step.

5.3 Clusters Task

The clusters task is next. We can run its setup task from the command line:

```
$ python cate.py clusters cate
```

This is a short task that sets up directories - it will set up `cate_cycl_exc` and `cate_meth_exc` in this case. In each it will put a script. Change into the first directory and list the contents:

```
$ cd cate_cycl_exc
$ ls
```

You should see four files: `cate.xyz`, `cycl.xyz`, `cate_cycl_solv.traj` and `cate_cycl_exc_sub`. The latter is the job script which can be used to launch a calculation for each of the snapshot clusters.

We would launch 10 calculations on equally-spaced snapshots from 0 to 90 (ie 0,10,20,30,. . . ,90) as follows:

$ sbatch –array=0-90:10 cate_cycl_exc_sub

The result will be a ONETEP calculation for each cluster. You can check their progress with `squeue` and `tail *.out`.

Once they have finished, they will produce files with names such as `cate_cycl_solv000.out` (and any other output files produced by the code) you can run the Spectra task to generate plots. You may want to inspect a few of these to check the behaviour is as expected.

5.4 Spectra Task

This is where the raw results from the Clusters task get turned into spectra for plotting purposes. You can run this task interactively at the command line to generate `.png` files on your compute cluster:

```
$ python cate.py spectra cate
```

or, if you prefer, you can transfer all the `.out` files from the clusters run back to another machine for interactive analysis in a notebook, by calling routines such as `spectra_driver()`

CHAPTER 6

Task drivers in ESTEEM

The task drivers in ESTEEM provide a means of looping over multiple sets of arguments for a given task, such that a set of calculations can be performed consistently over, for example, many combinations of molecule and solvent, or many different temperatures, or a range of cluster sizes.

There is also a main driver, `drivers.main()`, which determines which of the other drivers should be run, based on the command-line arguments. Invoking this main driver is the standard way of using ESTEEM in scripts.

However, for more advanced functionality, you can call the other drivers manually from your scripts. This page documents the main driver and the individual task drivers.

Drivers to run ESTEEM tasks in serial or parallel on a range of inputs.

These inputs will consist of solutes, solvents or pairs of solutes and solvent, depending on the task.

`esteem.drivers.`**`main`**(*all_solutes*, *all_solvents*, *all_solutes_tasks={}*, *all_solvate_tasks={}*, *all_clusters_tasks={}*, *all_spectra_tasks={}*, *all_qmd_tasks={}*, *all_mltrain_tasks={}*, *all_mltest_tasks={}*, *all_mltraj_tasks={}*, *make_script=<function make_sbatch>*)

Main driver routine which chooses other tasks to run as appropriate.

Control of what driver is actually called depends on command-line arguments with which the script was invoked: 'python <seed>.py <task> <seed> <target>'

all_solutes: dict of strings Keys are shortnames of the solutes. Entries are the full names.

all_solvents: dict of strings Keys are shortnames of the solvents. Entries are the full names.

all_solutes_tasks: namespace or class Argument list for the Solutes task - see solutes_driver documentation below for more detail.

all_solvate_tasks: namespace or class Argument list for the Solvate task - see solvate_driver documentation below for more detail.

all_clusters_tasks: namespace or class Argument list for the Clusters task - see clusters_driver documentation below for more detail.

all_spectra_tasks: namespace or class Argument list for the Spectra task - see spectra_driver documentation below for more detail.

make_script: dict of strings Routine that can write a job submission script. Usually parallel.make_sbatch.

esteem.drivers.**solutes_driver**(*all_solutes*, *all_solvents*, *task*)

Driver to run a range of DFT/TDDFT calculations on a range of solutes. If the script is run as an array task, it performs the task for the requested solute only.

all_solutes: dict of strings Keys are shortnames of the solutes. Entries are the full names.

all_solvents: dict of strings Keys are shortnames of the solvents (implicit only, here). Entries are the full names.

task: SolutesTask class Argument list for the whole Solutes job - see Solutes module documentation for more detail.

Arguments used only within the driver routine include:

`task.directory`: Directory prefix for where output of this particular 'target' of the Solutes calculation will be written

esteem.drivers.**solvents_driver**(*all_solvents*, *task*)

Driver to run a range of DFT/TDDFT calculations on a range of solvent molecules. See the Solutes module documentation for more info on what tasks are run.

all_solutes: dict of strings Keys are shortnames of the solutes. Entries are the full names.

all_solvents: dict of strings Keys are shortnames of the solvents. Entries are the full names.

task: SolutesTask class Argument list for the whole Solutes job - see Solutes module documentation for more detail.

Arguments used only within the driver routine include:

`task.directory`: Directory prefix for where output of this particular 'target' of the Solutes calculation will be written

wrapper: namespace or class Functions that will be used in the Solutes task - see Solutes module documentation for more detail.

esteem.drivers.**solvate_driver**(*all_solutes*, *all_solvents*, *seed*, *task*, *make_sbatch=None*)

Driver to run set up and run MD on solvated boxes containing a solute molecule and solvent. If the script is run as an array task, it performs the task for the requested solute/solute pair only, out of the nsolutes * nsolvents possible combinations.

all_solutes: dict of strings Keys are shortnames of the solutes. Entries are the full names.

all_solvents: dict of strings Keys are shortnames of the solvents. Entries are the full names.

task: SolvateTask class Argument list for the whole clusters job - see Solvate module documentation for more detail.

Arguments used within this routine include:

`task.md_suffix`: Directory suffix for where the MD runs will take place

`task.md_geom_prefix`: Directory prefix for where the geometries from the Solutes calculation should be obtained.

wrapper: class Wrapper that will be used in the Solvate task - see Solvate module documentation for more detail.

esteem.drivers.**clusters_driver**(*all_solutes*, *all_solvents*, *seed*, *task*, *make_sbatch=None*)

Driver to extract isolated clusters from solvated models, for a range of solute/solvent pairs.

Takes MD results from the directory {solute}_{solvent}_{task.md_suffix} and performs excitation calculation in the directory {solute}_{solvent}_{task.exc_suffix}.

If invoked from the base directory, rather than the excitation directory, it sets up the excitation directory and writes a job script then exits.

If invoked from the excitation directory, it performs the excitation calculations for all the extracted clusters. If the script is run as an array task, it performs the task for the requested cluster only.

Arguments

all_solutes: dict of strings Keys are shortnames of the solutes. Entries are the full names.

all_solvents: dict of strings Keys are shortnames of the solvents. Entries are the full names.

seed: str Overall 'seed' name for the run - used in creation of job scripts for the calculation

task: ClustersTask class Argument list for the whole clusters job - see Clusters module documentation for more detail.

Arguments used predominantly in the driver rather than the task main routine include:

`task.md_suffix`: Directory where MD outputs are to be found.

`task.exc_suffix`: Directory where results of Cluster excitation calculations will go.

make_sbatch: function Function that writes a job submission script for the clusters jobs

Output:

On first run, from the base directory of the project, the script will create subdirectories for each solute-solvent pair, with path '{solute}_{solvent}_{exc_suffix}'. The default value of `exc_suffix` is 'exc'.

To each directory, this routine will copy the trajectory file '{solute}_{solvent}_{md_suffix}/{solute}_{solvent}_solv.traj' which consists of `task.nsnaps` snapshots.

It will then create a job script using `make_sbatch` and `settings` with the name '{solute}_{solvent}_{exc_suffix}_sub' and exit.

The user then needs to run the individual job scripts from each subdirectory, probably as a job array: the array task ID should range from 0 to `task.nsnaps - 1`

The output of those runs will be the excited state energies of the clusters, which can be averaged over in the Spectra task.

`esteem.drivers.`**`spectra_driver`**(*all_solutes*, *all_solvents*, *task*)

Driver to plot spectra (and calculate spectral warping parameters) for a range of solute/solvent pairs

all_solutes: dict of strings Keys are shortnames of the solutes. Entries are the full names.

all_solvents: dict of strings Keys are shortnames of the solvents. Entries are the full names.

task: SpectraTask class Argument list for the whole spectra job - see Spectra module documentation for more detail.

Arguments used only in the driver include:

`task.exc_suffix`: Directory in which results of excitation calculations performed by the Clusters task can be found. The pattern used to find matches is: '{solute}_{solvent}_{exc_suffix}/{solute}_{solvent}_solv*.out'

`task.warp_origin_ref_peak_range`: Peak range searched when looking for 'reference' peaks. in the origin spectrum for spectral warping.

`task.warp_dest_ref_peak_range`: Peak range searched when looking for 'reference' peaks. in the destination spectrum for spectral warping.

task.warp_broad: Broadening to be applied to origin and destination spectra.

task.warp_inputformat: Format of the files to be loaded for origin and destination spectra. [TODO: May need to be adjusted to allow separate task.warp_ref_inputformat and task.warp_dest_inputformat]

task.warp_files: File pattern to search for when looking for origin and destination spectra for spectral warping.

task.merge_solutes: Dictionary: each entry should be a list of solute names that will be merged into the corresponding key

CHAPTER 7

Solutes module

7.1 SolutesTask

class esteem.tasks.solutes.**SolutesTask**(***kwargs*)

run(*namelist*)

Main routine for the Solutes task. Iterates through the sub-tasks, some of which are optional:

Geometry optimisation, rotamer checks, rotation to plane, excited state calculations, and vibrational frequency calculations.

namelist: list of str Short names of the solutes to be optimised. Initial geometries should be present in './xyz' directory.

self: namespace or class Member variables contain argument list for the whole job, with members including:

`solvent`, `target`, `rotate`, `rotamers`, `vibrations`, `charges`, `directory` `solvent_settings`, `basis`, `func`, `nroots`

Generate default arguments first with a call to solutes.make_parser(), then adjust the member variables as required.

wrapper: class Wrapper for running components of the job, with members including:

`singlepoint`, `geom_opt`, `excitations`, `freq`

Outputs for each species:

In the directory 'xyz': initial geometries (either input, or from get_xyz_files)

In the directory 'geom': outputs of geometry optimisations.

In the directory 'opt': optimised gas-phase geometries.

In the directory 'is_opt': optimised implicit solvent geometries. These may be used as inputs for the Solvate task.

In the directory 'is_tddft_{solvent}': implicit solvent tddft results. These may be used as inputs for spectral warping in the Spectra task.

get_xyz_files(*namelist*, *out_path*)

Downloads initial geometries from the NCI's webserver cactus, based on their IUPAC names.

These geometries are usually not great, but are a reasonable starting point for optimisation.

Visit https://cactus.nci.nih.gov/chemical/structure to see what works and check your names before use.

Uses `wget`, so if the machine you are using does not have access to this command, this routine will fail, in which case put starting point geometries in the directory 'xyz'.

namelist: dict Keys are shortnames (eg "cate"), entries are full names (eg "catechol" or "1,2-dihydroxybenzene")

out_path: str String for directory name where xyz files will be written (eg "xyz"). Created if not present

geom_opt_all(*solute_names*, *in_path*, *out_path*, *geom_opt_func*, *calc_params*, *driver_tol='default'*, *solvent=None*, *charges={}*)

Geometry optimise all of a list of solutes

solute_names: list of str Short names of the solutes to be optimised

in_path: str Directory where .xyz files are expected to be found. Any not present are skipped.

out_path: str Directory where optimised structure .xyz files are written. Created if not present.

geom_opt_func: function A function wrapping creation of an ASE calculator and using it to perform geometry optimisation.

calc_params: dict Contents varies between different wrappers, but generally specifies basis, functional etc

driver_tol: str Geometry optimisation tolerance level (eg in NWChem)

target: int Excited state index, or None for ground state

solvent: str Implicit solvent name, or None for gas-phase

charges: dict Keys are strings corresponding to some or all of the entries in solute names, entries are net charges on each molecule

find_best_rotas(*solute_names*, *in_path*, *out_path*, *singlepoint_func*, *geom_opt_func*, *calc_params*, *solvent=None*, *charges={}*)

Finds the lowest energy rotamer for each of a list of solutes. Proceeds by identifying -OH groups attached to C-C units, and tries 'flipping' the dihedral, then optimising the resulting geometry if it within a certain tolerance of the original energy. If any lower energy structure is found, this will be returned instead of the original one.

solute_names: list of str Short names of the solutes to be tested

in_path: str Directory where .xyz files are expected to be found. Any not present are skipped.

out_path: str Directory where best rotamer structure .xyz files are written. Created if not present.

singlepoint_func: function A function wrapping creation of an ASE calculator and using it to perform a singlepoint calculation.

geom_opt_func: function A function wrapping creation of an ASE calculator and using it to perform geometry optimisation.

calc_params: dict Contents varies between different wrappers, but generally specifies basis, functional etc

solvent: str Implicit solvent name, or None for gas-phase

charges: dict Keys are strings corresponding to some or all of the entries in solute names, entries are net charges on each molecule

rotate_all_to_xy_plane(*solute_names*, *in_path*, *out_path*, *target=None*)
Rotates each of a list of solutes so that the longest two C-C distances lie in the xy plane.

solute_names: list of str Short names of the solutes to be tested

in_path: str Directory where .xyz files are expected to be found. Any not present are skipped.

out_path: str Directory where rotated structure .xyz files are written. Created if not present.

target: int Excited state index, or None for ground state

find_range_sep(*solute_names, in_path, out_path, wrapper, calc_params, solvent=None, charges={}, rs_range=[0.1, 0.2, 0.3], all_readonly=False*)
Optimises the range separation parameter gamma in a range-separated Hybrid functional with Yukawa-switching. See 'Using optimally tuned range separated hybrid functionals in ground-state calculations: Consequences and caveats', Andreas Karolewski, Leeor Kronik, and Stephan Kümmel J. Chem. Phys. 138, 204115 (2013) https://aip.scitation.org/doi/10.1063/1.4807325 https://pubs.acs.org/doi/abs/10.1021/ct5000617 and 'Electronic Band Shapes Calculated with Optimally Tuned Range-Separated Hybrid Functionals' B. Moore, et al, D. Jacquemin, J. Chem. Theory Comput. 2014, 10, 10, 4599 https://pubs.acs.org/doi/10.1021/ct500712w

Minimises J^2 = Sum_i=0^1 [eps_H[N+i] + IP(N+i)))]^2 where IP(N) = E(N-1) - E(N) so J^2 = Sum_i=0^1 [eps_H[N+i] + E(N-1+i) - E(N+i)]^2

= [eps_H[N] + E(N-1) - E(N)]^2 + [eps_H[N+1] + E(N) - E(N+1)]^2

calc_all_excited_states(*solute_names*, *in_path*, *out_path*, *excit_func*, *calc_params*, *nroots*, *solvent=None*, *charges={}*, *plot_homo=None*, *plot_lumo=None*, *plot_trans_den=None*)
Calculate excited states for each of a list of solutes

solute_names: list of str Short names of the solutes to be tested

in_path: str Directory where .xyz files are expected to be found. Any not present are skipped.

out_path: str Directory where output files are written. Created if not present.

excit_func: function A function wrapping creation of an ASE calculator and using it to perform a electronic excitation calculations.

calc_params: dict Contents varies between different wrappers, but generally specifies basis, functional etc

nroots: int Number of excitations to find

target: int Excited state index, or None for ground state

solvent: str Implicit solvent name, or None for gas-phase

charges: dict Keys are strings corresponding to some or all of the entries in solute names, entries are net charges on each molecule

calc_vib_freq(*solute_names*, *in_path*, *out_path*, *freq_func*, *calc_params*, *solvent=None*, *charges={}*)
Calculate vibrational frequencies for each of a list of solutes

solute_names: list of str Short names of the solutes to be tested

in_path: str Directory where .xyz files are expected to be found. Any not present are skipped.

out_path: str Directory where output files are written. Created if not present.

freq_func: function A function wrapping creation of an ASE calculator and using it to perform a vibrational frequency calculation.

calc_params: dict Contents varies between different wrappers, but generally specifies basis, functional etc

target: int Excited state index, or None for ground state

solvent: str Implicit solvent name, or None for gas-phase

charges: dict Keys are strings corresponding to some or all of the entries in solute names, entries are net charges on each molecule

chdir()
Change the current working directory to the specified path.

path may always be specified as a string. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

getcwd()
Return a unicode string representing the current working directory.

makedirs(*name [, mode=0o777][, exist_ok=False]*)
Super-mkdir; create a leaf directory and all intermediate ones. Works like mkdir, except that any intermediate path segment (not just the rightmost) will be created if it does not exist. If the target directory already exists, raise an OSError if exist_ok is False. Otherwise no exception is raised. This is recursive.

np = <module 'numpy' from '/home/docs/checkouts/readthedocs.org/user_builds/esteem/env

path = <module 'posixpath' from '/home/docs/checkouts/readthedocs.org/user_builds/este

read(*index: Any = None*, *format: str = None*, *parallel: bool = True*, *do_not_split_by_at_sign: bool = False*, ***kwargs*) → Union[ase.atoms.Atoms, List[ase.atoms.Atoms]]
Read Atoms object(s) from file.

filename: str or file Name of the file to read from or a file descriptor.

index: int, slice or str The last configuration will be returned by default. Examples:

- `index=0`: first configuration
- `index=-2`: second to last
- `index=':'` or `index=slice(None)`: all
- `index='-3:'` or `index=slice(-3, None)`: three last
- `index='::2'` or `index=slice(0, None, 2)`: even
- `index='1::2'` or `index=slice(1, None, 2)`: odd

format: str Used to specify the file-format. If not given, the file-format will be guessed by the *filetype* function.

parallel: bool Default is to read on master and broadcast to slaves. Use parallel=False to read on all slaves.

do_not_split_by_at_sign: bool If False (default) `filename` is splited by at sign `@`

Many formats allow on open file-like object to be passed instead of `filename`. In this case the format cannot be auto-decected, so the `format` argument should be explicitly given.

rotate_and_center_solute(*nat_solute=None*, *boxsize=None*)
Rotates a cluster model so the solute is centered and lies in the xy plane

write(*images: Union[ase.atoms.Atoms, Sequence[ase.atoms.Atoms]], format: str = None, parallel: bool = True, append: bool = False, **kwargs*) → None
Write Atoms object(s) to file.

filename: str or file Name of the file to write to or a file descriptor. The name '-' means standard output.

images: Atoms object or list of Atoms objects A single Atoms object or a list of Atoms objects.

format: str Used to specify the file-format. If not given, the file-format will be taken from suffix of the filename.

parallel: bool Default is to write on master only. Use parallel=False to write from all slaves.

append: bool Default is to open files in 'w' or 'wb' mode, overwriting existing files. In some cases opening the file in 'a' or 'ab' mode (appending) is useful, e.g. writing trajectories or saving multiple Atoms objects in one file. WARNING: If the file format does not support multiple entries without additional keywords/headers, files created using 'append=True' might not be readable by any program! They will nevertheless be written without error message.

The use of additional keywords is format specific. write() may return an object after writing certain formats, but this behaviour may change in the future.

7.2 Standalone module routines

Runs DFT and TDDFT calculations on Solute (or Solvent) molecules, in implicit solvent

7.3 Command-line usage

CHAPTER 8

Solvate module

8.1 SolvateTask

class esteem.tasks.solvate.**SolvateTask**(***kwargs*)

setup_amber()
Handles setup of the solvated model using AmberTools.

The setup has 5 main sections:

1. Counterions are set up
2. Amber input files for the solute molecule are created
3. Amber input files for the solvent molecule are created
4. A box of solvent is added to the solute, and counterions added
5. Restraints are calculated for the counterions

args: namespace or class

Input variables to the routine - see listing under Command-Line usage for details

Generate with a call to solvate.make_parser()

wrapper: class Wrapper to run calculations of this task

setup_lammps()
Handles setup of the solvated model for a LAMMPS calculation (also uses AmberTools)

args: namespace or class input variables to the routine - see listing under Command-Line usage for details

run()
Handles running of MD on a solvated model using an MD Wrapper (currently Amber or LAMMPS).

There are five phases to the task:

Setup, Heating, Density Equilibration, Equilibration and Snapshot Generation

Constraints are turned on and off as appropriate in different phases:

SHAKE constraints on -H during heating and density equil, no constraints on -H during equil and snapshots

Counterions are restrained from coming too close to the solute

args: namespace or class Argument list for the whole job, with members including:

wrapper: namespace or class Wrapper for running components of the job, with members including:

`singlepoint`, `minimise`, `heatup`, `densityeq`, `equil` and `snapshots`

***Outputs*:** A trajectory file, named '{solute}_{solvent}_{md_suffix}/{solute}_{solvent}_solv.traj' which contains `self.nsnaps` geometries of the solvated box, each spaced by `self.nsteps` steps of `self.timestep` units of time (wrapper-dependent).

An example for catechol in water with the default `md_suffix`

8.2 Standalone module routines

Sets up and runs solvated Molecular Dynamics in Explicit Solvent

`esteem.tasks.solvate.`**`counterion_charge`**(*counterions*)

Calculate the net charge on the solute, given a set of counterions

8.3 Command-line usage

CHAPTER 9

Clusters module

9.1 ClustersTask

class esteem.tasks.clusters.**ClustersTask**(***kwargs*)

run()

Main routine for processing clusters and running them for excitations.

The steps involved are:

1. Load the trajectory file of MD snapshots, which for a given solvent solute pair it expects to find at '{solute}_{solvent}_solv.traj' in the current directory

2. 'Carve' spheres out of the trajectory, that is to say:

   a. Delete counterions (checking they are not within the sphere first)

   b. Delete any whole solvent molecules for which no atoms of the solvent molecule are within `self.radius` of any atom in the solute molecule.

   c. Label solvent molecules with a tag if they are within `self.kernel` by the criteria above.

   d. Rotate and center the cluster in the simulation cell, using the two most-distant pairs of carbon atoms in the solute to find a common 'plane' for the solute snapshots.

3. Calculate electronic excitations for each cluster. This can currently be done with the ONETEP wrapper functions or with NWChem but other wrappers can be added.

Arguments

self: Argument list for the task, with attributes including:

`solute`, `solvent`, `radius`, `output`, `task_id`, `counterions`, `charges kernel`, `basis`, `func`, `boxsize`, `impsolv`,''nroots'', `cleanup`, `continuation`

wrapper: class Wrapper to use in the Clusters task

Output:

If `self.task_id` is None, excited state calculations for the whole trajectory. If has a int value, then an excited state calculation for just that frame in the trajectory.

carve_spheres(*soluseed*, *solvseed*, *counterions=None*, *solvent_radius=0.0*, *kernel_radius=0.0*, *nmol_solvent_targ=None*, *boxsize=None*, *task_id=None*)
Carves out spheres from a periodic solvent model, centered on the solute

calc_all_excitations(*soluseed*, *solvseed*, *traj_carved*, *excit_func*, *charge=0*, *calc_params={}*, *impsolv=None*, *nroots=1*, *writeonly=False*, *continuation=False*, *cleanup=False*)
Loop over trajectory frames and do an excited states calculation for each one

calc_all_vibrations(*soluseed*, *solvseed*, *traj_carved*, *geom_opt_func*, *freq_func*, *charge=0*, *calc_params={}*, *impsolv=None*, *nroots=1*, *writeonly=False*, *continuation=False*, *cleanup=False*)
Loop over trajectory frames and do a vibrational frequency calculation for each one

9.2 Standalone module routines

Performs processing of Molecular Dynamics Trajectories to extract cluster models centered on the solute molecule and including all solvent molecules within a given range

esteem.tasks.clusters.**delete_counterions**(*t*, *solvent_radius*, *nat_tot*, *nat_solute*, *nat_counterions*)
Deletes counterions from an Atoms model. Assumes they appear in the range [nat_solute:nat_solute+nat_counterions]

esteem.tasks.clusters.**delete_excess_molecules**(*t*, *nat_tot*, *nat_solvent*, *nat_solute*, *nmol_solvent_targ*)
Deletes solvent molecules

esteem.tasks.clusters.**delete_distant_molecules**(*t*, *solvent_radius*, *nat_tot*, *nat_solvent*, *nmol_solvent*, *nat_solute*)
Deletes solvent molecules beyond a certain radius from the solute from an Atoms model Assumes that the first nat_solute atoms are the solute, and after that solvent atoms are arranged in nmol_solvent groups of size nat_solvent

esteem.tasks.clusters.**label_nearby_molecules**(*t*, *kernel_radius*, *nat_tot*, *nat_solvent*, *nmol_solvent*, *nat_solute*)
Adds a tag to solvent molecules within a certain radius from the solute to an Atoms model Assumes that the first nat_solute atoms are the solute, and after that solvent atoms are arranged in nmol_solvent groups of size nat_solvent

esteem.tasks.clusters.**rotate_and_center_solute**(*t*, *nat_solute=None*, *boxsize=None*)
Rotates a cluster model so the solute is centered and lies in the xy plane

9.3 Command-line usage

CHAPTER 10

Spectra module

10.1 SpectraTask

class esteem.tasks.spectra.**SpectraTask**(***kwargs*)

main(*fig=None*, *ax=None*, *plotlabel=None*, *rgb=array([0., 0., 0.])*)

Main routine for plotting a spectrum. Capable of applying Gaussian broadening to a stick spectrum to produce a broadened spectrum, and also of applying spectral warping to shift/scale the stick spectrum.

Arguments:

args: namespace or class All arguments for the job - see documentation below.

fig, ax: matplotlib objects Figure and axis objects for matplotlib.pyplot. Initialised anew if they have value None on entry.

plotlabel: Label to add the key for this dataset.

rgb: Colour of the line/points for this dataset. If set to (-1.0,-1.0,-1.0), the RGB colour will be calculated from the spectrum.

Returns:

broad_spectrum,spec_plot,fig,ax,all_transition_origins

Output:

Plots the spectrum to 'args.output' as a png file if requested.

find_spectral_warp_params(*dest_spectrum*, *origin_spectrum*, *arrow1_pos*, *arrow2_pos*)

Finds spectral warping parameters via a range of schemes. See spectra documentation page for more detail.

Arguments

dest_spectrum: numpy array of floats Contains the spectral warp 'destination' spectrum (usually a high level of theory that can only be afforded for the solute molecule)

origin_spectrum: numpy array of floats Contains the spectral warp 'origin' spectrum (usually a cheap level of theory, same as in the Clusters job)

args.warp_scheme: str The scheme used for the warping. Allowed values: 'beta', 'alphabeta', 'betabeta'

args.warp_origin_ref_peak_range: list of 2 floats Peak range searched when looking for 'reference' peaks in the origin spectrum for spectral warping.

args.warp_dest_ref_peak_range: list of 2 floats Peak range searched when looking for 'reference' peaks in the destination spectrum for spectral warping.

args.broad: Energy gaussian broadening applied to the stick spectra, useful to merge peaks that you want to treat as one peak in spectral warping.

Returns

[beta], [alpha, beta], [beta1, beta2, omega1o, omega2o]: floats describing spectral warp parameters

Also sets arrow1_pos, arrow2_pos for use in spectral warp plots.

10.2 Standalone module routines

Task that generates and plots uv/vis Spectra for solutes in solvent. Also contains routines to calculate spectral warp parameters and RGB colours from spectra

`esteem.tasks.spectra.`**`RGB_colour`**(*spectrum*, *args*)

Finds the RGB colour corresponding to a given absorption spectrum and illumination.

spectrum: numpy array Spectrum for which to find the colour

args: namespace or class Full set of arguments for the spectra task. Relevant to this routine are:

`args.XYZresponse` and `args.illuminant` which supply the Color Space XYZ response spectra and the illuminant spectrum, respectively.

These must be on the same wavelength scale as the spectrum.

10.3 Command-line usage

CHAPTER 11

Trajectories module

Low-level tools for handling trajectory files.

11.1 Python library usage

Functions to help generate and manipulate trajectories

esteem.trajectories.**generate_md_trajectory**(*model*, *seed*, *target*, *traj_label*, *trajname_suffix*, *md_func*, *count_snaps*, *count_equil*, *md_steps*, *md_timestep*, *temp*, *calc_params*, *constraints=None*, *dynamics=None*, *singlepoint_func=None*)

Runs equilibration followed by statistics generation MD for a given model.

This generic function gets called by both AIMD and by MLMD.

model: ASE Atoms Initial geometry for the MD trajectory

seed: str String containing name of molecule and excited state index

target: int or None Excited state index

traj_label: character or str String labelling trajectory (usually A,B,C..)

seed: str String containing name of molecule and excited state index

trajname_suffix: str String appended to `seed_state_traj` to give full trajectory filename

md_func: function Wrapper function that runs `md_steps` steps of MD on `model`

count_snaps: int Number of snapshot runs

count_equil: int Number of equilibration runs

md_steps: int Number of molecular dynamics timesteps in each run above

md_timestep: float Molecular dynamics timestep within the runs

temp: float Thermostat temperature (NVT ensemble)

calc_params: Control-parameters for the wrapper (for QM this is the basis, functional, and target; for ML this is the calculator seed, suffix, prefix and target)

constraints: wrapper-dependent type Variables controlling the constraints

dynamics: wrapper-dependent type Variables controlling the dynamics

esteem.trajectories.**cycle_step_labels_and_restarts**(*seed*, *traj_label*, *prevdir*, *currdir*, *nextdir*, *prevtarg*, *currtarg*, *nexttarg*, *prevstep*, *currstep*, *nextstep*, *count*, *prevcount*, *rst_ext=None*)

Move restart files around so that each step continues from the previous one

esteem.trajectories.**find_initial_geometry**(*seed*, *geom_opt_func*, *calc_params*, *which_traj=None*)

Obtains a suitable initial geometry for the current seed and state. Optimises it if not present.

seed: str String indicating name of molecule, used to find xyz file

geom_opt_func: function Wrapper function that runs a geometry optimisation

calc_params: Control-parameters for the wrapper (for QM this is the basis, functional, and target; for ML this is the calculator seed, suffix, prefix and target)

esteem.trajectories.**recalculate_trajectory**(*seed*, *target*, *traj_label*, *trajname_suffix*, *input_target*, *input_suffix*, *singlepoint_func*, *calc_params*, *input_traj_range=None*, *output_traj_offset=0*, *solvent=None*)

Loads snapshots from a trajectory and recalculates the energy and forces with the current settings seed: str

String containing name of molecule and excited state index

target: int or None Excited state index

traj_label: character or str String labelling trajectory (usually A,B,C..)

esteem.trajectories.**cycle_restarts**(*seed*, *traj_label*, *trajname_suffix*, *prevtarg*, *currtarg*, *prevstep*, *currstep*)

Move db files around so that each step continues from the previous one

esteem.trajectories.**get_trajectory_list**(*ntraj*)

Returns a list of characters to be used as trajectory labels Currently ABCDEDFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz (52)

esteem.trajectories.**merge_traj**(*trajnames*, *trajfile*)

Merges a list of trajectories supplied as a list of filenames, and writes the result to another trajectory supplied as a filename

esteem.trajectories.**split_traj**(*input_traj_file*, *output_trajs=None*, *ntrajs=None*, *randomise=False*, *start=0*, *end=-1*)

Takes in a trajectory filename, and splits it into sections, randomly or sequentially

esteem.trajectories.**fit_atom_energies_to_traj**(*traj*, *atom_traj_file*)

Uses linear regression to best fit atom energies to trajectory with varying numbers of atoms of each species

esteem.trajectories.**subtract_atom_energies_from_traj**(*traj*, *atom_traj*, *trajout*)

Subtracts the energies associated with isolated atoms from the total energies in a trajectory

esteem.trajectories.**compare_traj_to_traj**(*trajx*, *trajy*, *plot_file=None*, *xlabel=None*, *ylabel=None*)

Compares two trajectories to each other and calculates statistics for how much they differ in energy and force

trajx: ASE Trajectory Trajectory whose energy is plotted along x-axis

trajy: ASE Trajectory Trajectory whose energy is plotted along y-axis

plot_file: Filename to write plot image to.

xlabel, ylabel: str Axis labels for plots.

CHAPTER 12

QMD_Trajectories module

12.1 QMDTrajTask

class esteem.tasks.qmd_trajectories.**QMDTrajTask**(***kwargs*)

run()

Sets up and runs an ab initio molecular dynamics run on a given molecule (whose name is provided by `args.seed`) in the ground or excited state (specified by `args.target`).

Results, including a trajectory file with the stored snapshots, are saved to files appending `args.suffix` to the seed and state, for use in future runs.

The run is divided into equilibration (`args.nequil` runs of `args.qmd_steps` MD steps each, with timestep `args.qmd_timestep`), then snapshot generation (`args.nsnap` runs of `args.qmd_steps` MD steps each, with timestep `args.qmd_timestep`).

A thermostat (wrapper-dependent) at temperature `args.temp` means we stay in the NVT ensemble.

Constraints can be applied using `args.constraints` - the meaning depends on the underlying wrapper.

Optionally can be used to recalculate singlepoint energies for the steps of a pre-existing trajectory.

args: namespace or class Full set of arguments to the QMD_Trajectories task - see below for a listing.

Key arguments include `basis`, `func`, `qmd_timestep`, `qmd_steps`, `nsnap`, `nequil`, `temp`, `constraints`

Generate with a call to qmd_trajectories.make_parser()

wrapper: namespace or class List of functions for running components of the job, with members including:

`singlepoint`, `geom_opt` and `qmd`.

12.2 Standalone module routines

12.3 Command-line usage

CHAPTER 13

ML_Training module

13.1 MLTrainingTask

class esteem.tasks.ml_training.**MLTrainingTask**(***kwargs*)

run()
Main routine for the ML_Training task

13.2 Standalone module routines

Defines a task to train a Machine Learning calculator on a trajectory of snapshots by calling the train() function of the MLWrapper

13.3 Command-line usage

CHAPTER 14

ML_Testing module

14.1 MLTestingTask

class esteem.tasks.ml_testing.**MLTestingTask**(***kwargs*)

run()
Main routine for the ML_Testing task

14.2 Standalone module routines

Defines a task to test a Machine Learning calculator by comparing its results to those of an existing trajectory or set of trajectories

esteem.tasks.ml_testing.**compare_calc_to_traj**(*calc*, *trajin*, *trajout_file*, *plot_file=None*, *e_offset=0.0*)
Compare the energy and force predictions of a calculator to results in an existing trajectory

14.3 Command-line usage

CHAPTER 15

ML_Trajectories module

15.1 MLTrajTask

class esteem.tasks.ml_trajectories.**MLTrajTask**(***kwargs*)

run()
Main routine for the ML_Trajectories task

15.2 Standalone module routines

Defines a task to use a Machine Learning calculator to generate Molecular Dynamics trajectories

esteem.tasks.ml_trajectories.**load_trajectory_dipole**(*seed_state_str*, *traj_suffix*, *ntraj*, *nsnaps*, *mdsteps*)
Loads a set of saved trajectory files and extracts the dipole moment as a function of time

esteem.tasks.ml_trajectories.**calculate_ir_spectrum**(*mu_t*, *dt*, *freq_scale_fac*, *sigma*)
Processes the dipole moment as a function of time for a collection of trajectories to calculate IR absorption spectrum

15.3 Command-line usage

CHAPTER 16

Wrappers available in ESTEEM

16.1 Amber Wrapper

class esteem.wrappers.amber.**AmberWrapper**(*nprocs=None*)

Sets up the AMBER Calculator (via ASE) for Molecular Dynamics runs

prepare_input(*seed*, *netcharge=0*, *offset=0*)

Prepares input parameters and topologies for Amber calculations

add_solvent_box(*solute*, *solvent*, *counterions*, *solvatedseed*, *box_size*)

Loads an Amber mol2 file for a solute and solvent, and creates a solvated box

reimage(*seed*, *infile*, *outfile*)

Runs the cpptraj program with a simple script to center the frame on residue 1 (the solute) and translate molecules back into the home cell

dipole(*seed*, *crdfile*)

Runs the cpptraj program with a simple script to get the dipole moment from the final trajectory

fix_amber_pdb(*seed*)

An awk-based hack to fix AMBER pdb files to make them work with ASE

singlepoint(*model*, *seed*, *calc_params={}*, *forces=False*, *solvent=None*, *readonly=False*, *continuation=False*)

Runs a singlepoint calculation with the Amber ASE calculator

geom_opt(*model*, *seed*, *calc_params={}*, *solvent=None*)

Runs a geometry optimisation calculation with the Amber ASE calculator

heatup(*model*, *seed*, *calc_params={}*, *nsteps=100*)

Runs a heatup temperature-ramp calculation with the Amber ASE calculator

densityequil(*model*, *seed*, *calc_params={}*, *nsteps=100*)

Runs a density equilibration calculation with fixed hydrogens with the Amber ASE calculator

equil(*model*, *seed*, *calc_params={}*, *nsteps=100*)

Runs an equilibration calculation at constant volume with flexible hydrogens with the Amber ASE calculator

snapshots(*model*, *seed*, *calc_params={}*, *nsnaps=1*, *nsteps=100*, *start=0*)
Runs a long MD trajectory for snapshot generation with the Amber ASE calculator

16.2 LAMMPS Wrapper

Defines the LAMMPSWrapper class

class esteem.wrappers.lammps.**LAMMPSWrapper**
Sets up the LAMMPS Calculator (via ASE) for Molecular Dynamics runs

lammps_setup(*lammps_cmd=None*, *nprocs=None*)
Prepares run commands etc for LAMMPS calculations

load(*seed*, *target=None*, *prefix=''*, *suffix=''*, ***kwargs*)
Loads an existing AMP Calculator and converts it into a PROPhet-LAMMPS calculator

seed: str

target: int

suffix: str

kwargs: dict other keywords to pass to AMP.load

geom_opt(*model*, *seed*, *calc_params*, *driver_tol='default'*, *solvent=None*, *readonly=False*)
Runs a singlepoint calculation with the LAMMPS ASE calculator

model: ASE Atoms

seed: str

suffix: str

dummy: str

target: int or None

solvent: str or None

readonly: bool

run_mlmd(*model*, *mdseed*, *calc_params*, *md_steps*, *md_timestep*, *superstep*, *temp*, *solvent=None*, *restart=False*, *readonly=False*, *constraints=None*, *continuation=None*)
Runs a Molecular Dynamics calculation with the AMP ASE calculator.

model: ASE Atoms

seed: str

target: int

suffix: str

md_steps: int

md_timestep: float

superstep: int

temp: float

target: int or None

solvent: str or None

restart: bool

readonly: bool

singlepoint(*model*, *seed*, *calc_params*, *target=None*, *solvent=None*, *readonly=False*)
Runs a singlepoint calculation with the LAMMPS ASE calculator

model: ASE Atoms

seed: str

suffix: str

dummy: str

target: int or None

solvent: str or None

readonly: bool

minimise(*solvated*, *minimised*, *seed*)
Runs a geometry optimisation calculation with the LAMMPS ASE calculator

heatup(*minimised*, *heated*, *seed*, *nsteps*)
Runs a heatup temperature-ramp calculation with the LAMMPS ASE calculator

densityequil(*heated*, *densityeq*, *seed*, *nsteps*)
Runs a density equilibration calculation with fixed hydrogens with the LAMMPS ASE calculator

equil(*densityeq*, *equbd*, *seed*, *nsteps*)
Runs an equilibration calculation at constant volume with flexible hydrogens with the LAMMPS ASE calculator

snapshots(*seed*, *snapin*, *snapout*, *nsnaps*, *nsteps*)
Runs a long MD trajectory for snapshot generation with the LAMMPS ASE calculator

16.3 NWChem Wrapper

class esteem.wrappers.nwchem.**NWChemWrapper**
Sets up and runs the NWChem Calculator (via ASE) for DFT and TDDFT calculations

setup(*nprocs=None*, *nwchem_cmd=None*, *nwchem_top=None*)
Sets up the internal variables of the NWChemWrapper class, including run command

cleanup(*seed*)
Cleans up temporary files created by a NWChem run that are of no further use

singlepoint(*model*, *label*, *calc_params={}*, *solvent=None*, *charge=0*, *spin=0*, *forces=False*, *continuation=False*, *readonly=False*, *calconly=False*, *cleanup=True*)
Runs a singlepoint calculation with the NWChem ASE calculator

geom_opt(*model_opt*, *label*, *calc_params={}*, *driver_tol='default'*, *solvent=None*, *continuation=False*, *charge=0*, *readonly=False*, *calconly=False*, *cleanup=True*)
Runs a Geometry Optimisation calculation with the NWChem ASE calculator

freq(*model_opt*, *label*, *calc_params={}*, *solvent=None*, *charge=0*, *temp=300*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=True*)
Runs a Vibrational Frequency calculation with the NWChem ASE calculator

run_qmd(*model*, *steplabel*, *calc_params*, *qmd_steps*, *qmd_timestep*, *superstep*, *temp*, *solvent=None*, *charge=0*, *continuation=False*, *readonly=False*, *constraints=None*, *dynamics=None*, *cleanup=True*)
Runs a Quantum Molecular Dynamics calculation with the NWChem ASE calculator

excitations(*model*, *label*, *calc_params={}*, *nroots=1*, *solvent=None*, *charge=0*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=True*, *plot_homo=None*, *plot_lumo=None*, *plot_trans_den=None*)
Calculates TDDFT excitations with the NWChem ASE calculator

read_excitations(*calc*)
Read Excitations from nwchem calculator.

read_freq(*calc*)
Read Vibrational Frequencies and Normal Modes from results of nwchem calculator.

16.4 ONETEP Wrapper

class esteem.wrappers.onetep.**OnetepWrapper**
Sets up and runs the ONETEP Calculator (via ASE) for DFT and TDDFT calculations

setup(*nprocs=None*, *mthreads=None*, *onetep_cmd=None*, *mpirun=None*, *set_pseudo_path=None*, *set_pseudo_suffix=None*)
Sets up the internal variables of the OnetepWrapper Class

singlepoint(*model*, *label*, *calc_params*, *solvent=None*, *charge=0*, *forces=False*, *restart=False*, *readonly=False*, *writeonly=False*, *calconly=False*)
Runs a singlepoint calculation with the ONETEP ASE calculator

geom_opt(*model_opt*, *label*, *calc_params*, *driver_tol='default'*, *solvent=None*, *charge=0*, *readonly=False*, *writeonly=False*, *calconly=False*)
Runs a Geometry Optimisation calculation with the ONETEP ASE calculator

excitations(*model*, *label*, *nroots=1*, *solvent=None*, *charge=0*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=False*)
Calculates TDDFT excitations with the ONETEP ASE calculator

path = <module 'posixpath' from '/home/docs/checkouts/readthedocs.org/user_builds/este

16.5 ORCA Wrapper

class esteem.wrappers.orca.**ORCAWrapper**
Sets up and runs the ORCA Calculator (via ASE) for DFT and TDDFT calculations

setup(*nprocs=None*, *orca_cmd=None*, *ORCA_top=None*)
Sets up the internal variables of the ORCAWrapper class, including run command

cleanup(*seed*)
Cleans up temporary files created by a ORCA run that are of no further use

singlepoint(*model*, *label*, *calc_params={}*, *solvent=None*, *charge=0*, *spin=0*, *forces=False*, *continuation=False*, *readonly=False*, *calconly=False*, *cleanup=True*)
Runs a singlepoint calculation with the ORCA ASE calculator

geom_opt(*model_opt*, *label*, *calc_params={}*, *driver_tol='default'*, *solvent=None*, *continuation=False*, *charge=0*, *readonly=False*, *calconly=False*, *cleanup=True*)
Runs a Geometry Optimisation calculation with the ORCA ASE calculator

freq(*model_opt*, *label*, *calc_params={}*, *solvent=None*, *charge=0*, *temp=300*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=True*)
Runs a Vibrational Frequency calculation with the ORCA ASE calculator

run_qmd(*model*, *steplabel*, *calc_params*, *qmd_steps*, *qmd_timestep*, *superstep*, *temp*, *solvent=None*, *charge=0*, *continuation=False*, *readonly=False*, *constraints=None*, *dynamics=None*, *cleanup=True*)
Runs a Quantum Molecular Dynamics calculation with the ORCA ASE calculator

excitations(*model*, *label*, *calc_params={}*, *nroots=1*, *solvent=None*, *charge=0*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=True*, *plot_homo=None*, *plot_lumo=None*, *plot_trans_den=None*)
Calculates TDDFT excitations with the ORCA ASE calculator

read_excitations(*calc*)
Read Excitations from ORCA calculator.

read_freq(*calc*)
Read Vibrational Frequencies and Normal Modes from results of ORCA calculator.

16.6 PhysNet Wrapper

class esteem.wrappers.physnet.**PhysNetWrapper**(***kwargs*)
Sets up, trains and runs a PhysNet Neural Network Calculator to represent a potential energy surface.

load(*seed*, *target=None*, *prefix=''*, *suffix=''*, *atoms=None*)
Loads an existing PhysNet Calculator

seed: str

target: int

suffix: str

kwargs: dict other keywords

train(*seed*, *prefix=''*, *suffix=''*, *trajfile=''*, *target=None*, *restart=False*, ***kwargs*)
Runs training for PhysNet model using an input trajectory as training points

seed: str

target: int

suffix: str

trajfile: str

restart: bool

kwargs: dict

run_mlmd(*model*, *mdseed*, *calc_params*, *md_steps*, *md_timestep*, *superstep*, *temp*, *solvent=None*, *restart=False*, *readonly=False*, *constraints=None*, *dynamics=None*, *continuation=None*)
Runs a Molecular Dynamics calculation with the PhysNet ASE calculator.

model: ASE Atoms

seed: str

calc_params: dict

md_steps: int

md_timestep: float

superstep: int

temp: float

target: int or None

solvent: str or None

restart: bool

readonly: bool

geom_opt(*model*, *seed*, *calc_params*, *driver_tol='default'*, *solvent=None*, *charge=0*, *spin=0*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=False*, *traj=None*)
Runs a geometry optimisation calculation with the PhysNet ASE calculator

model: ASE Atoms

seed: str

calc_params: dict

dummy: str

driver_tol:

target: int or None

solvent: str or None

readonly: bool

freq(*model_opt*, *seed*, *calc_params*, *solvent=None*, *charge=0*, *temp=300*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=True*)
Runs a Vibrational Frequency calculation with the PhysNet ASE calculator

model_opt: ASE Atoms

seed: str

suffix: str

dummy: str

driver_tol:

target: int or None

solvent: str or None

temp: float

readonly: bool

singlepoint(*model*, *seed*, *calc_params*, *solvent=None*, *charge=0*, *spin=0*, *readonly=False*, *cleanup=True*)
Runs a singlepoint calculation with the PhysNet ASE calculator

model: ASE Atoms

seed: str

suffix: str

dummy: str

target: int or None

solvent: str or None

readonly: bool

16.7 AMP Wrapper

Defines the AMPWrapper Class

class esteem.wrappers.amp.**AMPWrapper**

Sets up and runs the AMP Calculator (via ASE) to use a neural-network representation of a potential energy surface.

load(*seed*, *target=None*, *prefix=''*, *suffix=''*, ***kwargs*)

Loads an existing AMP Calculator

seed: str

target: int

suffix: str

kwargs: dict other keywords to pass to AMP.load

train(*seed*, *prefix=''*, *suffix=''*, *trajfile=''*, *target=None*, *restart=False*, ***kwargs*)

Runs training for AMP ASE calculator using an input trajectory as training points

seed: str

target: int

suffix: str

trajfile: str

restart: bool

kwargs: dict

run_mlmd(*model*, *mdseed*, *calc_params*, *md_steps*, *md_timestep*, *superstep*, *temp*, *solvent=None*, *restart=False*, *readonly=False*, *constraints=None*, *dynamics=None*, *continuation=None*)

Runs a Molecular Dynamics calculation with the AMP ASE calculator.

model: ASE Atoms

seed: str

calc_params: dict

md_steps: int

md_timestep: float

superstep: int

temp: float

target: int or None

solvent: str or None

restart: bool

readonly: bool

geom_opt(*model*, *seed*, *calc_params*, *driver_tol='default'*, *solvent=None*, *readonly=False*)

Runs a geometry optimisation calculation with the AMP ASE calculator

model: ASE Atoms

seed: str

calc_params: dict

dummy: str

driver_tol:

target: int or None

solvent: str or None

readonly: bool

freq(*model_opt*, *seed*, *calc_params*, *solvent=None*, *charge=0*, *temp=300*, *writeonly=False*, *readonly=False*, *continuation=False*, *cleanup=True*)
Runs a Vibrational Frequency calculation with the AMP ASE calculator

model_opt: ASE Atoms

seed: str

suffix: str

dummy: str

driver_tol:

target: int or None

solvent: str or None

temp: float

readonly: bool

singlepoint(*model*, *seed*, *suffix*, *dummy*, *target=None*, *solvent=None*, *readonly=False*)
Runs a singlepoint calculation with the AMP ASE calculator

model: ASE Atoms

seed: str

suffix: str

dummy: str

target: int or None

solvent: str or None

readonly: bool

A wrapper in ESTEEM is an intermediate layer between a task and ASE calculator. It allows the task to call upon a number of different electronic structure codes, without needing to store details of how to invoke each of them within the module for that task.

The idea is that all the standard calculator setup for a particular theoretical spectroscopy task can be hidden inside the wrapper, and the functions of the wrapper, which might include "singlepoint" energy evaluations, geometry optimisation, or calculation of electronic excitations, can then be passed into the task as arguments.

The wrappers fall into three categories at present: electronic structure wrappers (currently NWChem and ONETEP), Molecular Dynamics wrappers (currently Amber and LAMMPS) and Machine Learning wrappers (currently PhysNet and AMP). Details of these wrappers can be found on their individual pages.

If you have a new code you would like ESTEEM to be able to use, then adding a new wrapper for it should be fairly easy, as long as there is already an ASE calculator for your code. The wrapper is simply a set of default settings for a typical run - effectively a mapping of arguments passed to the task, to input varibles passed to the calculator. The arguments might mean very different things to different calculators: for example the "basis" argument when passed to NWChem is a single string representing a basis set such as "6-31G*", whereas when passed to ONETEP it is a tuple specifying cutoff energy, NGWF radius, and conduction band energy range.

16.8 Molecular dynamics wrappers

Need to specify functions for: `singlepoint`, `minimise`, `heatup`, `densityeq`, `equil`, `snapshots`

16.9 Electronic Structure wrappers

Need to specify functions for: `singlepoint`, `geom_opt`, `excit`, `freq`. If used in combination with the ML-based tasks, they should also provide `run_qmd`

16.10 Machine Learning wrappers

Need to specify functions for: `train`, `load`, `geom_opt`, `run_mlmd`

CHAPTER 17

Other Wrappers

Writing a wrapper is a pretty easy task for anyone with ASE and general python skills. Any electronic structure, MD or Machine-Learning package that already has an ASE interface can probably be given an appropriate wrapper within a day or so's work.

I am very happy to accept contributions of new wrappers to the code. Please have a look at the code most similar package to what you want to add and adapt it as required.

17.1 Future Plans

Upon request, if presented with a use case, I could probably find time to add wrappers for any of the following codes, some of which I am reasonably familiar with already:

CASTEP, QuantumEspresso, SchNarc, CP2K, BigDFT, Gaussian.

Get in touch with me at n.d.m.hine@warwick.ac.uk to discuss the possibilities.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

Index

F

G

H

L

M

N

O

P

Q

R

S

T

W