The Plan for Robust and Accurate Potentials – PRAPs

PRAPs it’ll work and PRAPs it won’t!

Table of Contents:

1. [Introduction, Installation, and Runtime](#_Introduction,_Installation,_and) …………………………………………………...2
2. [Tutorial](#tutorial)…………………………………………………………………………………….5
   1. [CHf – Standard Run](#CHf)……………………………………………………………….5
   2. [CMo – Restarting](#CMo)………………………………………………………………….8
   3. [HfMo – DFT Convex Hulls](#HfMo)………………………………………………………11
3. [Input File](#_Input_file)…………………………………………………………………………………14
4. [Output Files and Directory Structure](#_Output_files_and)……………………………………………………..19
5. [Errors and Analysis](#_Errors_and_analysis)………………………………………………………………………20
6. [Convex Hulls](#_Convex_hulls)……………………………………………………………………………..22
7. [Build Structure](#_The_Build_Structure)…………………………………………………………………………...24
   1. [Utilities](#_The_Utilities)…………………………………………………………………………...24
   2. [Mliputils](#_Mliputils.py)………………………………………………………………………….32
8. [PRAPs-ID](#_PRAPs-ID)………………………………………………………………………………..37
9. [Final Remarks](#_Final_Remarks)……………………………………………………………………………38
10. [References](#references)………………………………………………………………………………..39

## Introduction, Installation, and Runtime

The Plan for Robust and Accurate Potentials (PRAPs) is a workflow ‘software’ that automates a lot of small tasks and keeps the big ones in the right order when training moment tensor potentials (MTPs) with the Machine Learned Interatomic Potentials package (MLIP).1,2

**Dependencies:**  
AFLOW v3+,3 VASP v5+,4 MLIP v2+1, Python v3+  
**Python’s dependencies:**  
Re, Numpy,5 Scipy,6 Matplotlib,7 Ternplot,8 Pandas,9 Statistics

To get started, obtain the TAR file and place it in the desired directory, referred to as the “install directory.” An example might be: /user/you/software/PRAPs/PRAPs.tar. Once placed, unpack it in the install directory and look over the README and the file *install.py*. This will provide instructions on how to complete the install. PRAPs is a collection of Bash and Python scripts, with no executable binary, whose styles have been chosen to be most appropriate for the task at hand. PRAPs is intended to be run from the same directory as installation, and designed around this centralization. This document uses the term “install directory” to refer to the installation path (internally given the variable $pth), “submit directory” as the path from which a job is submitted, and “working directory” as the path in which PRAPs will look for input and output.

The primary code to run PRAPs is located in *praps.sh* and is intended to be called from a job submission script such as *run\_praps.slurm*. When run, PRAPs navigates from the submit directory to the working directory to perform its tasks. The working directory should be specified by the user, otherwise, PRAPs will create a default working directory. The default will be based on the elements used in your system. For example, if you have a system containing carbon, hafnium, and tungsten, then the default working directory will be $submit\_directory/CHfW. The string “CHfW” is the “compound” ($cmpd) and the list of elements [C, Hf, W] are the “elements” ($els), which are passed as Python lists and Bash arrays. The working directory is internally referred to as the compound path ($cmpd\_pth).

When run, PRAPs will overwrite the contents of the working directory with the new run. Because of this default setting, if you perform two PRAPs runs from the same submit directory with the same compound, PRAPs will overwrite the first with the second. To avoid this, please set the working directory manually. The working directory and the elements can both be set in the file *inpraps.sh*, an example of which is provided in the PRAPs install directory.

After unpacking the TAR file, completing the install requires running *install.py*, which sets all of the path variables that PRAPs will require. This should be run as follows:

python adjust\_paths.py $pth $potpth $modpth $mtpth $email $pypth

|  |  |
| --- | --- |
| $pth | Install directory |
| $potpth | Location of POTCAR files |
| $modpth | Location of slurm modules for AFLOW, MLIP, etc. |
| $mtpth | Location of untrained MTP files |
| $email | Your email for slurm notifications |
| $pypth | Location of custom Python modules, optional |

The install should now be complete. There will be four folders in the install directory. *Par/* is for the parallel version of MLIP, *ser/* is for the serial version of MLIP, *tutorial/* contains the tutorial files, and *utils/* contains all of the various Python and Bash scripts that PRAPs will use during the run.

To perform a PRAPs run, you need two things: a PRAPs input file (*inpraps.sh*) and a collection of chemical data organized into a .cfg file. A sample inpraps file can be found in the *PRAPs/par/* or *PRAPs/ser/* folders. Place the inpraps file into the working directory and make sure the elements, working directory, and .cfg filename are all correct. The *inpraps.sh* file is described in Section 2. Then make sure the desired .cfg file is also in the working directory. You can then submit the PRAPs run as:

sbatch praps.slurm $working\_directory/inpraps.sh

or more directly from the primary file as:

bash $install\_directory/praps.sh $working\_directory/inpraps.sh

A .cfg file is MLIP’s way of storing chemical information in which multiple chemical structures are included in one file. This means that if users wish to partition a data set in two or more parts, perhaps for training and testing, they must do so themselves. PRAPs assumes that users will have up to three .cfg files:

* DFT\_CFG: This is a .cfg file containing chemical structures and compositions as well as energies, forces, and/or stresses obtained from prior computations.
* URX\_CFG: This is a .cfg file that only contains chemical structures and compositions.
* REF\_CFG: This .cfg file only contains the ground-state chemical structures of the *elements* present in DFT\_CFG and/or URX\_CFG. Providing energies is strongly advised, but not required.

PRAPs’ minimum requirements are either the DFT\_CFG or the URX\_CFG files, but we recommend users include all three for best performance. The .cfg files can be assembled in a variety of ways, such as being scraped from an online database or assembled from the users’ own quantum mechanical calculations and structure generation. The MLIP software includes support for conversion of LAMMPS and VASP output files. PRAPs adds support for VASP input files and AFLOW .json files. Other chemical data files are not yet supported but may be in the future.

A summary of the steps to accomplish a PRAPs run:

1. Place the *inpraps.sh* file and any relevant .cfg files into the working directory.
2. Edit the *inpraps.sh* file to have the correct working directory, elements, MTP level, and .cfg files (and anything else).
3. Go to your submit directory and open praps.slurm.
4. Make sure that the “bash praps.sh $1” command is pointing to the right place.
5. Submit your job as, “sbatch praps.slurm working\_directory/inpraps.sh”.

And that’s it! PRAPs will run and eventually finish. Or not. You can find the output logs in the submit directory. If it gets interrupted mid-run, you can restart the job (described in Section 2). In my experience, a PRAPs run that finishes successfully takes a few days, but this depends on many things such as the level of MTP chosen, active learning convergence criteria, available computational power, etc. All of the calculated outputs will be found in the working directory.

## Tutorial

After installing PRAPs, you will find a folder labeled *tutorial/* inside the PRAPs folder. This folder contains data for CHf, CMo, and HfMo and the input files needed to run PRAPs for each of them. To run, determine which version of MLIP you have installed, which can be verified by the “mlp help” command. Then, navigate to the *ser/* or *par/* folder, as appropriate. Check your slurm submission script,[[1]](#footnote-1) and then run as:

sbatch praps.slurm ../tutorial/CHf/inpraps.sh

PRAPs will be scheduled and run until completion. Below, we’ll take you through the three tutorials provided in the package, and the kinds of output to expect. Please note, because MLIP initiates some of its processes with random settings, some of your results may not be *exactly* the same as ours. But you should get results very close and, hopefully, will gain some familiarity with the sorts of inputs and outputs to expect from PRAPs.

## 2a. CHf – Standard Run

There are four input files: *aflow\_CHf.cfg*, *randspg\_CHf.cfg*, *ref.cfg*, and *inpraps.sh*. The first, *aflow\_CHf.cfg*, contains 10,231 configs. You can check this with the command:

grep BEGIN aflow\_CHf.cfg | wc -l

Likewise, *randspg\_CHf.cfg* contains 111 configs and *ref.cfg* contains two. The *inpraps.sh* file should contain a number of settings, most of which are described later in this document in Section 3. Open it and examine the setting marked “cmpd\_pth.” It should be set to “…/PRAPs/tutorial/CHf.” Edit the “…” to your install directory so that the full path now reads something like, “/root/user/you/PRAPs/tutorial/CHf”. For now, you can leave the remaining settings alone. They tell PRAPs what the elements are (carbon and hafnium), what MTP level to use (Level 16), the names of the other input files, and various other settings.

Now you can run PRAPs. Navigate to the *ser/* or *par/* folder, depending on your installation of MLIP, and run the submit command as shown above. As it runs, PRAPs will navigate to the working directory (what we wrote in cmpd\_pth earlier) and will eventually output (see full description in Section 4) some text files and a few graphs. Let’s go through them briefly.

The text files *err\_train\_16.txt* and *err\_predict\_16.txt* contain various statistical errors relating to training and testing. Looking at them, you’ll want to look for the line, “Average absolute difference” in the “Energy per atom” section. In *err\_train\_16.txt*, for instance (the training errors), the first calculation should display something around 0.060. This is a value given in eV/atom, and converts to 60 meV/atom. This value is the mean-absolute-error (MAE). Below is the root-mean-square-error (RMSE) and that should be around 90 meV/atom. If you exit the file and run the command:

mlp calc-errors pot\_16\_CHf.mtp train.cfg

Then you should obtain the same set of numbers, the same text, from the first entry in *err\_train\_16.txt*. The second entry is the training error from the robust potential (RP) and the third should be from the accurate potential (AP). The table below summarizes this information.

|  |  |  |  |
| --- | --- | --- | --- |
| Err\_train\_16.txt | MAE (RMSE)  (±20 meV/atom) | MTP filename | CFG filename |
| Pre-RP | 60 (90) | pot\_16\_CHf.mtp | train.cfg |
| RP | 110 (180) | pot\_als\_robust.mtp | als-robust.cfg |
| AP | 30 (50) | pot\_als\_acc.mtp | als-acc.cfg |

The second measure of success is the prediction error, found in *err\_predict\_16.txt*. This attempts to answer the question, “Does my MTP reproduce the DFT data?” Note that the way PRAPs answers this may not be what you prefer, so practicing the “mlp calc-errors” command will be important to making sure you get the right information. The table below summarizes PRAPs’ prediction errors:

|  |  |  |  |
| --- | --- | --- | --- |
| Err\_predict\_16.txt | MAE (RMSE)  (± 40 meV/atom) | MTP filename | CFG filename |
| RP-1 | 100 (140) | pot\_als\_robust.mtp | aflow\_CHf.cfg |
| RP-2 | 100 (140) | pot\_als\_robust.mtp | dft\_distfilt.cfg |
| AP-1 | 100 (120) | pot\_als\_acc.mtp | lowE\_robust\_relaxed.cfg |
| AP-2 | 160 (340) | pot\_als\_acc.mtp | dft\_distfilt.cfg |

We recommend paying attention most to RP-2 and AP-1. PRAPs assumes that your input DFT data may not be perfect, so it performs some filtration at the start of the process, resulting in *dft\_distfilt.cfg*. If nothing is removed, then RP-1 and RP-2 will be the same. If some configurations are removed, then these two measures will be different and RP-2 is more likely to be a better estimate of how the RP will perform on quality DFT data. AP-1 is better than AP-2 because the AP is trained to be good at low-energy structures, only, which is what we find in *lowE\_robust\_relaxed.cfg* (and *lowE\_vasp.cfg*), while *dft\_distfilt.cfg* includes high-energy structures, and thus the AP will not perform as well there.

If you want to obtain errors not found in these files, please use the “mlp calc-errors command.”

Finally, PRAPs will output a file called *highlow\_16.csv*. This is a way to determine if an MTP can correctly place the 10-highest-energy and 10-lowest-energy configurations. In other words, the MTP might not reproduce energy exactly, but it should still agree that low-energy configs are actually low-energy. This is described more fully in Section 5.

The last output are some convex hull plots. Even when, as in this example, the convex hull setting is turned off, PRAPs will still plot one set of hulls: the chullcans (short for convex-hull-candidates). These are the convex hulls generated by the MTP and your initial DFT data (if any). PRAPs will make these as individual plots as follows:

A screenshot of a graph

AI-generated content may be incorrect.

1. DFT: This is the hull generated by your initial DFT data after distance/volume filtration (*lowE\_vasp.cfg*).
2. RR: This stands for Robust-Relaxed and is generated from *lowE\_robust\_relaxed.cfg*. This is what you get when you relax your initial data (both DFT\_CFG and URX\_CFG) with the RP.
3. ALS\_AP\_v: This stands for Accurate-Predicted-VASP and is generated from *ALS\_AP\_v.cfg*. This hull represents the AP’s prediction of your initial DFT data (after distance/volume filtration).
4. ALS\_AP\_RR: This stands for Accurate-Predicted-Robust-Relaxed and is generated from *ALS\_AP\_RR.cfg*. This is the hull made by having the AP predict the contents of *lowE\_robust\_relaxed.cfg*.
5. ALS\_AR\_RR: This stands for Accurate-Relaxed-Robust-Relaxed and is generated from *ALS\_AR\_RR.cfg*. This is the hull made by having the AP relax the contents of *lowE\_robust\_relaxed.cfg*. This is the “most MTP” hull.

For binary systems like CHf, CMo, and HfMo, you can run the script *make\_2D\_hulls.py* manually to make these plots. If you do, you can actually set them to generate multiple plots at once and either overlay them, arrange them vertically, or arrange them horizontally (Figure 2). At this time, there is no way to plot multiple ternary plots (Ex. CHfTa) at once in a single image. We regret not having any scripts at the moment to plot higher-order convex hulls (Ex. CHfTaZr or CMoNbTaTi).

A screenshot of a computer screen

AI-generated content may be incorrect.

## 2b. CMo – Restarting

Sometimes life intervenes in our plans and our jobs are interrupted, or simply fail partway through. The checkpoint system allows you to restart a job from an interrupted position. Since many processes are really miniature workflows unto themselves, such as active learning, sometimes these restarts will also restart those miniature workflows and sometimes they’ll pick up almost exactly where they left off.

In the CMo directory, examine the *inpraps.sh* file. You will need to set the cmpd\_pth variable as you did before. But now at the end, you should add a line reading “CHK=2”. This is a checkpoint flag. PRAPs will read the most recent one and start a calculation from the indicated step (see Section 3 for more). Here, we are going to re-start a job that was theoretically interrupted after training the RP, but before the AP. Take a look at the files in here and you’ll notice a few differences.

We are going to start with both the Pre-RP (*pot\_16\_CMo.mtp*) and the RP (*pot\_als\_robust.mtp*) and their training sets (*train.cfg* and *als-robust.cfg*, respectively). You’ll notice we don’t have the error files yet. We can make them later. For now, let’s get PRAPs running. We’re going to do two things differently. First, copy (or build on your own), the input job submission script from the *ser/* or *par/* directory. Inside, change the command that runs PRAPs to read:

bash …/PRAPs/par/praps.sh inpraps.sh

This allows you to “point” the job at the PRAPs code from the submit directory without having to be in the installation directory. You can also add a full filepath to the inpraps.sh part of the command to “point” the job at a particular sub-directory. Just make sure to replace the “…” in the command above with the full path to your installation directory.

The other thing to do in here is to rename the job output script. Consider something like “praps\_CMo.out,” “CMo\_16.out,” or my personal format, “CMo\_16\_chk2-1.out.” This makes sure that when I submit multiple PRAPs jobs from the same directory the outputs don’t overwrite each other. Careful naming also lets me know that this file refers to a job with CMo, at Level 16, starting from Checkpoint 2, for the first time (sometimes, it happens to need chk4-3).

Once you’ve updated the name, you can now run PRAPs. If done correctly, PRAPs should see the checkpoint flag and start the process by preparing the *lowE\_robust\_relaxed.cfg* and subsequently training the AP. Once PRAPs is finished, let’s look at the output file to see how it went.

The output will start with header information, telling you about module loading, the CPU info, and the date. Then you’ll see some lines saying, “PRAPs found Checkpoint Tag 1, skipping Pre-training.” That tells us where PRAPs is starting. You can also search the output file for certain words. “Robust” will help you find where the RP training starts and ends. Searching “Accurate” will help you find the AP start and end. If you see a start line and not an end line, you know PRAPs was interrupted in the middle of that step. If you need to search inside an active learning step, try “Rabbit Hole” to jump between each iteration of active learning. Searching for “training set” will help you see how many configurations are being added to the training set at each iteration. And note that many steps are punctuated with date-time information so you can try and track how long the job is taking. Now let’s look at the error and plot job outputs.

Inside the error files you’ll notice that you’re missing the portions associated with the Pre-RP and RP since PRAPs skipped those. In normal usage, you’d probably still have those since this kind of job would have gone through those steps before being interrupted. But I chose to do this so you could go over the “mlp calc-errors” again. To add the missing errors, use the following:

mlp calc-errors pot\_16\_CMo.mtp train.cfg >> err\_train\_16.txt

mlp calc-errors pot\_als\_robust.mtp als-robust.cfg >> err\_train\_16.txt

mlp calc-errors pot\_als\_robust.mtp aflow\_CMo.cfg >> err\_predict\_16.txt

mlp calc-errors pot\_als\_robust.mtp dft\_distfilt.cfg >> err\_predict\_16.txt

Below is a summary table of all of the error values (for energy only) you should expect to see. Remember that some of these may be different from what we provide here. Beyond, we’ve also included sample plots that should have been generated.

|  |  |  |  |
| --- | --- | --- | --- |
| CMo Training | MAE (RMSE)  (+- 20 meV/atom) | MTP filename | CFG filename |
| AP | 20 (30) | pot\_als\_acc.mtp | als-acc.cfg |
| Pre-RP | 100 (190) | pot\_16\_CMo.mtp | train.cfg |
| RP | 125 (220) | pot\_als\_robust.mtp | als-robust.cfg |
| CMo Prediction | MAE (RMSE)  (+- 40 meV/atom) | MTP filename | CFG filename |
| AP-1 | 60 (90) | pot\_als\_acc.mtp | lowE\_robust\_relaxed.cfg |
| AP-2 | 350 (18,000) | pot\_als\_acc.mtp | dft\_distfilt.cfg |
| RP-1 | 130 (300) | pot\_als\_robust.mtp | aflow\_CMo.cfg |
| RP-2 | 130 (300) | pot\_als\_robust.mtp | dft\_distfilt.cfg |

A collage of images of a graph

AI-generated content may be incorrect.

1. DFT chullcans
2. RR chullcans
3. ALS\_AP\_v chullcans
4. ALS\_AP\_RR chullcans
5. ALS\_AR\_RR chullcans

## 2c. HfMo – DFT Convex Hulls

Calculating the final DFT convex hulls takes a long time. To help speed this up, we’ve provided you with all of the trained MTPs in this tutorial. Of course, you are welcome to re-train them from scratch following the advice above if you wish. To get started, investigate the inpraps.sh file and change the cmpd\_pth to the correct location. Then set custom\_relax=true, CHULL=true, and CHK=4.

The custom\_relax setting tells PRAPs whether to use your own settings for DFT or not. If false, PRAPs will call AFLOW, and if true, it will use whatever settings you have. By default, PRAPs assumes you are using VASP. Other codes will be available in the future. If you’d prefer to use AFLOW in this section, simply set custom\_relax=false. The CHULL setting controls whether PRAPs performs the final DFT relaxations to calculate the convex hulls or not. And the checkpoint tag sets us up to focus only on the convex hulls and not any of the training steps.

To start with, we need a few files. PRAPs will make some of these, but it’s good practice to make you do them as well to familiarize yourself with some of our scripts. So, before we run PRAPs, we’re going to make the ALS\_AP\_v file. This is a prediction of the low-energy structures from the original DFT structures, but now made by the AP. The name of the file means “The ALS-generated AP predicting a set of structures with EFS from VASP.” Cumbersome, but it’s possible to generate an AP from Basic Training as well as ALS, so we needed lots of words. Anyways, run the following commands:

mlp calc-efs pot\_als\_acc.mtp lowE\_robust\_relaxed.cfg post\_als\_acc\_mtp.cfg

python ../../utils/ID.py post\_als\_acc\_mtp.cfg I suffix=AP

python ../../utils/lowE\_configs.py post\_als\_acc\_mtp.cfg ALS\_AP\_RR.cfg F 0.05

The first command tells MLIP to use the AP to make a prediction and save it. PRAPs uses the “post\_” keyword in filenames to indicate a prediction has been made. Then we run *ID.py* (see Section 8) to update the PRAPs-ID’s of the configs inside the file. This can help you track history, or provenance, and identify where a specific config came from and what operations have been performed on it. The last command runs the energy filtration script, lowE\_configs.py, to filter out everything in the .cfg file by that last value—0.05 eV/atom, a.k.a the chull\_var from the inpraps file. This script will keep all ground-state configurations, and all configs within 0.05 eV/atom of that composition’s ground state. We chose 0.05 eV/atom as the default because it is roughly 2 kBT of energy, but you can change this in the inpraps file if you desire.

Now you can run PRAPs. Go into your submit file and set the name to something like HfMo\_16\_chk4-1.out. This seems excessive, but you will very likely need to re-run this a few times. The DFT is *long*. If you are using Slurm, this is all you need to do. But if you are using a different scheduler, you need to identify your submission command: the equivalent to sbatch. Go into praps.sh in either the *ser/* or *par/* folder (whichever you’re using) and go to Line 452. Change the “sbatch” command to the appropriate command for your system. This is a small issue that will be patched in the future.

Now you can run PRAPs. Expect this to take a long time. PRAPs will do all of its work in the *…/HfMo/chulls* folder and will try to run five jobs at once. Each of these will take one of the following: *lowE\_vasp.cfg* (DFT), *lowE\_robust\_relaxed.cfg* (RR), *ALS\_AP\_v.cfg*, *ALS\_AP\_RR.cfg*, and *ALS\_AR\_RR.cfg*. Each job will make a storage and working sub-directory inside chulls. In the storage directory, PRAPs will convert the .cfg to POSCAR files, add elements to them, extract Feature tags for later, and ask AFLOW to determine the AFLOW prototype of each (a label that tells us the composition and space group). Then we use the compositions and space groups to identify and remove duplicate structures, keeping only the lowest energy structure in each composition and space group. This dramatically reduces the amount of DFT we’ll have to perform. From there we run a DFT relaxation on the remaining structures.

If the custom\_relax setting is true, PRAPs will look for *INCAR\_rx*, *INCAR\_st*, and a *KPOINTS* file and will run VASP automatically. We have provided a sample set of these for you. If custom\_relax is set to false, PRAPs will try to call AFLOW to manage these instead. Both options are set to perform two calculations: a geometry optimization and a single-point calculation. If you set the save\_outcars tag to true, PRAPs will try to save all of the OUTCAR files, but since this takes a lot of space, this setting is off by default.

After the DFT relaxations are done, PRAPs will convert the OUTCAR back into a .cfg file. The process is a bit cumbersome, but has mostly been concentrated into ID.py. This ensures correct type-element matching, re-inserts all of the Feature tags from the original .cfg file, and updates the PRAPs-ID’s. Finally, PRAPs will clean everything up from the chulls directory and remove it. If you saved the OUTCAR files, they’ll be in the working directory.

After all the DFT, the last thing PRAPs does is make the convex hull plots. For each of the five jobs, PRAPs makes three plots total. The first are the “chullcans,” which we have seen from the previous tutorials. These are the convex hull candidates. Next are the “dftrelaxed.” These are the direct result of the DFT relaxations PRAPs just ran. Finally, there is the “DFT+dftrelaxed.” This involves concatenating *DFT\_dftrelaxed.cfg* onto the other four (RR, ALS\_AP\_v, ALS\_AP\_RR, and ALS\_AR\_RR) “dftrelaxed” files and re-drawing the convex hull. The purpose of this is to directly compare the “literature” convex hull from your original DFT data against all of the new data you just calculated. In many cases, the convex hulls are basically identical, but sometimes this is not true. How you use these is up to you. For more information, see Section 6.

## Input file

PRAPs uses an input file called *inpraps.sh*, which is a Bash file, and it should be kept as such. Python writes lists as [a, b, c] while Bash does it like (a b c), which is important for the elements tag. This file sets a lot of the variables and settings that PRAPs uses, some of which are required (Req), and some that are optional (Opt).

|  |  |  |
| --- | --- | --- |
| **Tag** | **Description** | **Optionality** |
| [els](#els) | The elements in your system | Req |
| [LevMTP](#LevMTP) | The level of MTP to be used for training | Req |
| [DFT\_CFG](#DFT_CFG) | A .cfg file containing structures and energy, force, and/or stress data | Opt\* |
| [URX\_CFG](#URX_CFG) | A .cfg file containing only structural data | Opt\* |
| [REF\_CFG](#REF_CFG) | A .cfg file containing only ground-state elemental data | Opt |
| [cmpd\_pth](#cmpd_pth) | The working directory | Opt |
| [chull\_var](#chull_var) | An amount of energy used for determining what structures to use when training the AP. | Opt |
| [mindist](#mindist) | The smallest acceptable nearest neighbor distance | Opt |
| [maxdist](#maxdist) | The largest acceptable nearest neighbor distance | Opt |
| [relax\_settings](#relax_settings) | Settings used for MLIP’s relaxation command | Opt |
| [training\_settings](#training_settings) | Settings used for MLIP’s training command | Opt |
| [basic\_acc](#basic_acc) | Whether or not you want an AP trained with basic training | Opt |
| [CHULL](#chull_tag) | Whether or not you want convex hulls | Opt |
| [save\_outcars](#save_outcars) | Whether or not you want every OUTCAR saved | Opt |
| [custom\_relax](#custom_relax) | Whether or not you want to use PRAPs’ built-in DFT settings | Opt |
| [filter\_trajectories](#filter_traj) | Whether or not you want to keep only the final-relaxed structures found in DFT\_CFG | Opt |
| [filter\_volumes](#filter_volumes) | Whether or not you want to filter DFT\_CFG by volume | Opt |
| [volume\_scaling](#vol_scaling) | How much can the volume differ from that of the final-relaxed structure found in DFT\_CFG | Opt |
| [filter\_forces](#filter_forces) | Whether or not to filter DFT\_CFG by forces | Opt |
| [ALS\_conv](#ALS_conv_tag) | Which convergence criteria should PRAPs use to stop active learning | Opt |
| [CHK](#CHK) | A checkpoint tag, used to re-start a run | Opt |

\* At least one of these is required.

**els**: Required   
The els variable is a bash array identifying the elements used in the system. Elements must be present in type-order. For example: els=(C Mo W) means that PRAPs will set type-0 atoms as carbon, type-1 atoms as molybdenum, and type-2 atoms as tungsten.

**LevMTP**: Required   
The LevMTP is short for “Level of MTP” and must be an even integer from 2 to 26, inclusive. This tells PRAPs which level of MTP to use for the training.

**DFT\_CFG**: Semi-Required   
The name of the .cfg file containing chemical structures with energy, forces and stresses (EFS) obtained from quantum mechanical calculations. PRAPs will use this file in the pre-training process and will combine it with URX\_CFG (below) to make relax.cfg. Users skipping the pre-training step can omit this tag so long as they include URX\_CFG.

**URX\_CFG**: Semi-Required   
The name of the .cfg file containing chemical structures without EFS information. PRAPs will combine this file with the DFT\_CFG file to make relax.cfg. Users may omit this tag as long as they include DFT\_CFG.

**REF\_CFG**: Optional   
The name of the .cfg file containing only the elemental chemical structures. PRAPs will use this to find the elemental energies for calculating the enthalpies of formation. If omitted, PRAPs will use an internal dictionary of energies (previously obtained from the AFLOW database).

**cmpd\_pth**: Optional, Default = install dir/cmpd   
This is the directory in which PRAPs will perform most of its tasks, and where it will read inputs and write outputs. This allows users to organize their data and point PRAPs to particular sub-directories. If a user wishes to set this to the submit directory, set to “$PWD”. If a user forgets to set this, PRAPs will default to the install directory/cmpd.

**chull\_var**: Optional, Default = 0.05 eV/atom   
This is a decimal value, interpreted in eV/atom. PRAPs uses this value to determine the energy filtering for the Accurate Potential. This value is also used to determine which configurations appear on the convex hull plots.

**mindist**: Optional, Default = 1.1 Å   
This is a decimal value, interpreted in Angstroms. PRAPs uses this value to determine the lower bound for the acceptable minimum interatomic distance. Configurations with a minimum interatomic distance smaller than this value are removed from training and relaxation.

**maxdist**: Optional, Default = 3.1 Å   
This is a decimal value, interpreted in Angstroms. PRAPs uses this value to determine the upper bound for the acceptable minimum interatomic distance. Configurations with a minimum interatomic distance larger than this value are removed from training and relaxation. In other words, PRAPs only keeps configurations if the smallest distance between any two atoms is within the range: *mindist* < *d* < *maxdist*.

**relax\_settings**: Optional   
This string should include any settings used in MLIP’s relaxation, in the form “–relax-setting=setting”

**training\_settings**: Optional   
This string should include any settings used in MLIP’s training in the form “–train-setting=setting”.

**basic\_acc**: Optional, Default = false   
This tells PRAPs whether to generate an Accurate Potential with basic training or not. It will generate an Accurate Potential with active learning regardless of this setting.

**CHULL**: Optional, Default = false   
This determines whether or not PRAPs runs the post-training convex hull analysis. If false, PRAPs will only generate the convex hulls from the MTP-generated data and DFT\_CFG. If true, PRAPs will perform additional DFT calculations using AFLOW and VASP to generate additional convex hulls. PRAPs will, in either case, make subdirectories containing the structures appearing on the convex hulls in POSCAR format.

**save\_outcars**: Optional, Default = false   
This only applies if CHULL = true. If true, this will save each OUTCAR file generated during DFT relaxation. Note that this will take up a lot of space.

**custom\_relax**: Optional, Default = false   
If true, this will avoid AFLOW during the DFT relaxation steps, and will allow users to employ their own relaxation parameters instead. This also allows users to modify the active learning single-point calculations. A pair of INCAR files has been provided for convenience, and the nomenclature should remain the same: INCAR\_rx for relaxation and INCAR\_st for single-point calculations.

**filter\_trajectories**: Optional, Default = false   
This setting arises after relaxing the configurations using the Robust Potential (RP), as PRAPs sets up the training set for the Accurate Potential (AP). If true, PRAPs will only select the final relaxed configuration from each structure to send onward, even if the initial configuration or those in the middle of the relaxation meet the low-energy criteria.

**filter\_volumes**: Optional, Default = false   
If true, PRAPs will filter out configurations based on cell volume rather than minimum interatomic distance. If true, PRAPs will look for the next tag as well.

**volume\_scaling**: Optional, Default = 1.25   
If the previous tag is true, this setting determines the volumes that PRAPs will allow. PRAPs maintains an internal dictionary of volumes for each element in units of Å3/atom. It will use those values to determine a reference volume for each composition in relax.cfg. The value of this variable is the amount of acceptable variation from that reference volume: for example, a value of 1.25 means configurations will be accepted if their volume is within the range 75% < *V* < 125% of the reference volume.

**filter\_forces**: Optional, Default = false   
This setting should be a float, if desired. This will remove any configuration with at least one force term outside the range -*x* < *F* < *x* where *x* is the value set here and *F* the values of the forces inside the configuration. This filtration occurs after the distance/volume filtration.

**ALS\_conv**: Optional, Default = 0   
This setting controls how PRAPs ends the active learning process. There are several options available. PRAPs checks this after performing the relaxation and selection steps, but before performing the single-point DFT calculations and re-training the MTP. If option 1-4 is selected, but the conditions for option 0 are met first, then active learning will end anyway.

0: Active learning is stopped when MLIP no longer selects any new configurations for training. This is the default setting.

1: Active learning is stopped when the number of structures selected to be added to the training set is less than 1% of the size of the training set.

2: Active learning is stopped when the energy root-mean-square error in eV/atom is less than the chull\_var (defined above).

3: Active learning is stopped when the difference in energy root-mean-square error between the previous iteration and the current iteration is less than 0.1 meV/atom.

4: Active learning stops after 50 iterations.

**CHK**: Optional, Default = 0   
This is the checkpoint tag and should be omitted for a completely clean PRAPs run. Over time, PRAPs will add tags in to the *inpraps.sh* file. When PRAPs is run, it will look for the largest CHK and skip all steps before that one. This is intended to be used in two ways: to avoid the pre-training step by setting CHK=1, or to re-start the job at roughly the same point it was stopped in the event of a crash or interruption (such as exceeding a wall-time-limit). The checkpoint tags are described below.

0: Set after initial variable and directory setup. Loading from here will start or continue the Pre-Training step.

1: Set after the Pre-Training has finished. Loading from here will start or continue the Robust Potential active learning step.

2: Set after the Robust Potential training has finished. Loading from here will start or continue the Robust Relaxation and (if turned on) the Basic Accurate Potential training step.

3: Set after the Robust Relaxation is produced and (if turned on) the Basic Accurate Potential training has finished. Loading from here will start or continue the Accurate Potential active learning step.

4: Set after the Accurate Potential training has finished. Loading from here will start or continue the analysis steps including (if turned on) the convex-hull analysis.

For PRAPs to run it is important to remember to set: els, LevMTP, cmpd\_pth, DFT\_CFG, and/or URX\_CFG. Everything else has defaults. You may want to adjust them, of course, but PRAPs probably won’t crash if you don’t.

## Output Files and Directory Structure

After the PRAPs run, there will be several more files and sub-directories in your working directory, including a TAR file holding everything. The sub-directories named “1-5” hold information about the Pre-Training step and can usually be safely discarded. The Pre-Trained potential is named *pot\_lev\_cmpd.mtp*, the RP is named *pot\_als\_robust.mtp*, and the AP is named *pot\_als\_acc.mtp*. Their associated training sets are, respectively: *train.cfg, als-robust.cfg,* and *als-acc.cfg*. If you opted for training a Basic Accurate potential as well, you’ll see *pot\_basic\_acc.mtp* and *basic-acc.cfg*. A few new text files will be present along with your input files. The various training errors are provided in *err\_train.txt* and the prediction errors are in *err\_predict.txt*. The highlow information (see Section 5) is given in *highlow.csv*.

If you opted to perform a convex hull analysis, you’ll see a lot of sub-directories with names like, “RR\_dftrx+DFT.” This is covered in more detail later (Section 5), but a brief description is included here. PRAPs will make several .cfg files, a few directories, and the convex hull images as .svg files. The images are what you really care about, and the directories contain POSCARs for the structures in the images (labeled using their AFLOW prototype labels). PRAPs will generate several convex hulls: one set using the DFT\_CFG data, one set using the Robust Potential, and three sets using the Accurate Potential. In other words, it uses both the RP and the AP to make several convex hull predictions. For each one of those, it also generates three hulls: the chullcans, the dft\_relaxed, and the dftrx+DFT. Chullcans means convex hull candidates and are the raw MTP-predictions before any relaxation with DFT. The dft\_relaxed are the structures in the chullcans after going through relaxation with DFT. And the dftrx+DFT includes both the dft\_relaxed and the structures in the DFT\_CFG convex hull.

## Errors and Analysis

The error files contain information that can be obtained using the “mlp calc-errors pot.mtp file.cfg” command. PRAPs uses this command to obtain the errors for a particular MTP as calculated against a particular .cfg file (containing EFS data). There are roughly two kinds of errors: training and prediction errors.

A training error is calculated as follows. An MTP is trained using a particular .cfg file. The MTP is then used to predict the energy, forces, and stresses of the same .cfg file that it was trained on. The MLIP program performs certain statistical calculations to compare the energy, forces, and stresses in the original .cfg file against the MTP’s prediction. A prediction error is calculated in a similar way, except that the .cfg file used is not the same one that the MTP was trained on. Note that for either of these, the .cfg file must contain energy, force, and/or stress data.

Typically, a user will be more interested in the expected prediction errors. PRAPs will calculate training errors for each MTP it trains and put them into the *err\_train.txt* file. The errors for the Pre-Trainined MTPs are also obtained, and you can find them in the *1-5* sub-directories, while the best one will be in the *err\_train.txt* file. PRAPs includes comparison of the Pre-Trained MTP’s performance vs. the DFT\_CFG, the RP vs. the DFT\_CFG, the RP vs. the distance-filtered DFT\_CFG, the AP vs. the low-Energy-DFT\_CFG, and the AP vs. the whole DFT\_CFG. In Ref [2], we reported a different set of prediction errors than these, where we compared the predictions made with various potentials against the final-relaxed-structure-AFLOW data (because we had a very particular set of data to work with). PRAPs does not automatically calculate this final-relaxed-structure error. If you’d like to do the same thing, run *filter.py* on the DFT\_CFG set using the trajectory filtering and then run the “mlp calc-errors” command.

The *highlow.csv* file contains another means of determining the goodness of an MTP’s predictions. Instead of seeing if the MTP can precisely reproduce the DFT data, we determine if MTP can predict the DFT-trends.10 We take the ten highest-energy configurations and the ten lowest-energy configurations from DFT\_CFG (and a few others) and write down their indices in a file. Then we grab the same top and bottom ten configurations from a .cfg with MTP-predicted EFS data, and compare the two sets of indices, then output it in a string like, “5-2/6-0”. This output means that the MTP predicted 5/10 of the lowest-energy configurations correctly and put two of the highest-energy configurations in the low-energy section. Then it predicted 6/10 of the high-energy configurations correctly and none of the low-energy configurations were found in the high-energy section. Here’s an example below:

* DFT\_CFG:
  + 10 Lowest-Energy Configurations
    - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
  + 10 Highest-Energy Configurations
    - 91, 92, 93, 94, 95, 96, 97,98, 99, 100
* MTP-Predicted DFT\_CFG:
  + 10 Lowest-Energy Configurations
    - 4, 7, 8, 92, 1, 3, 22, 25, 96, 34
  + 10 Highest-Energy Configurations
    - 63, 71, 72, 73, 91, 93, 94, 100, 95, 97
* Highlow Calculation:
  + 1, 3, 4, 7, 8 found on both 10-low-lists = 5
  + 92 and 96 found on DFT 10-high and MTP 10-low = 2
  + 91, 94, 94, 95, 97, and 100 found on both 10-high-lists = 6
  + No overlap between DFT 10-low and MTP 10-high = 0
  + Final score = 5-2/6-0

Ultimately, users will have to decide on their own how to interpret the error and highlow data; PRAPs uses it to select the best pre-trained potential. To do this, PRAPs examines each of the five pre-trained potentials’ highlow data (determined against the original DFT\_CFG) and prediction errors (against a random validation set selected from the DFT\_CFG). PRAPs selects the MTP with the lowest prediction error (Energy-per-atom root mean squared error, RMSE), the MTP with the best low score from the highlow data, and the MTP with the best high score from the highlow data. If the best-high and the best-low are the same MTP, that one is returned as the best. Otherwise, it’s the one with the lowest prediction error. While other metrics could have been defined to determine what the “best” MTP is, the one that is currently employed by PRAPs was used successfully in Ref [2]. Users may decide to modify the criteria used to choose the MTP employed in subsequent calculations.

## Convex hulls

Convex hull plots can be used to determine structures that are thermodynamically stable, and those that are metastable for a particular elemental combination. In addition, these plots can indicate the convex hull distance of each structure, related to how far in energy each structure is from thermodynamic stability. The structures employed and the level of theory used to calculate their energies can result in different convex hull plots. To produce convex hulls, you must set CHULL=true in the input file and you must provide a REF\_CFG, which is a .cfg file containing ground-state elemental configurations with energies, forces, and stresses. PRAPs will use the energies in REF\_CFG to calculate enthalpies of formation as follows:

[1]

Where Δ*H*f,*i* is the enthalpy of formation of some configuration *i*, *E*i is the energy of that configuration as determined by MTP or DFT, *x*j is the fractional composition of element *j* in configuration *i*, and εj is the energy of that element as recorded in REF\_CFG. Please note that if REF\_CFG is not provided, PRAPs contains an internal dictionary of energies obtained from the AFLOW website’s convex hull tool. These values are not up-to-date, nor do they reflect the level of theory employed by the user. They are provided to allow generation of an estimated hull and to prevent certain types of crashes, and we strongly suggest you include a REF\_CFG in your calculations.

PRAPs produces three sets of convex hulls that are named: chullcans, dftrelaxed, and dftrx+DFT. Chullcans are the convex hull candidates—the structures predicted to be on or near the convex hull before any DFT calculations are performed. Dftrelaxed are the plots obtained after relaxing the chullcans by DFT (by default this structural relaxation is managed via AFLOW, but you can source your own VASP files if you like). The dftrx+DFT adds to dftrelaxed the structures from DFT\_CFG, after DFT relaxation. Therefore, for each action by each potential, PRAPs will generate three convex hulls (except for the DFT\_CFG, which only makes two).

The first set of convex hulls to be generated will come from the DFT\_CFG. PRAPs will first prepare the DFT\_chullcans directly from the .cfg file. Then it will prepare the DFT\_CFG for DFT relaxation by removing duplicate structures. PRAPs determines the AFLOW prototype label for each structure (a string containing the composition and space group). If multiple structures share the same label and composition, PRAPs keeps the minimal energy structure and discards the others. After this filtering, the structures are relaxed. By default, this relaxation is managed with AFLOW, but users may submit their own relaxation via the inpraps file. After relaxation, the results are collated into the DFT\_dftrelaxed set and a convex hull is plotted. For DFT\_CFG the process stops here, but in all other cases, the file *DFT\_dftrelaxed.cfg* is concatenated onto the *\*\_dftrelaxed.cfg* file to form *\*\_dftrx+DFT.cfg* and a third convex hull is plotted from that. This final plot allows the user to compare the results from the actions of MTP and DFT calculations against the original input data.

PRAPs then repeats this process four more times, generating three convex hulls in each. First is the Robust Relaxed set (RR), generated by relaxing both the DFT\_CFG and URX\_CFG with the RP. Next is the AP\_v set, generated by using the AP to predict the low-energy structures found in DFT\_CFG. Third is the Accurate-Predicted Robust Relaxed set (AP-RR), which is generated by having the AP predict the low-energy structures from the RR set. And finally we produce the Accurate-Relaxed Robust Relaxed set (AR-RR), where the AP is used to relax the low-energy structures from the RR set.

A short discussion of the technical details will complete this section. The main PRAPs script will outsource the DFT relaxations to *dft\_chulls.sh* via separate sbatch commands to slurm to allow for speedier processing. The plotting is performed by *tri\_phase\_point2.py* and *trivex\_NAME.py*. There are two styles of convex hull available based on scripts written by my coworkers. The one used to make the plots in Ref. [2] was written by Masashi Kimura and can be accessed with the ‘M’ argument when running *tri\_phase\_points2.py*. It requires a particular matplotlib library called Ternary Plot and if you don’t have that you should run the ‘X’ mode to get plots made by Xiaoyu Wang. Masashi’s plots are generated by *trivex\_masashi.py* and Xiaoyu’s plots are made by *trivex\_xiaoyu.py*. By default, images are saved in the SVG file format. Despite the presence of the chull\_var tag in the inpraps file, structures are selected for plotting if they are on or within 52 meV/atom of the convex hull (corresponding to 2 kBT).

After plotting the hulls, PRAPS will prepare a file named *\*\_mapped.cfg* where the \* is replaced by the name of the input file (like RR\_chullcans or ALS\_AP\_v\_dftrelaxed). This file contains only the configurations that appear on the convex hull image. After generating the images, PRAPs uses these *\*\_mapped.cfg* files to generate summary .csv files which contain information like the energy, formation enthalpy, and hull distances of each structure appearing on the convex hull plot. PRAPs will also put all of those configurations into a *\*\_mapped/* sub-directory, labeled by their AFLOW prototypes, in case you need to grab them for visualization later. All of these things, and the images, will get put into the final TAR file at the end.

## Build Structure

Because MLIP can be installed in two versions, there are two versions of PRAPs. The files supporting the serial version are in the *PRAPs/ser/* folder. The files supporting the parallel version are in the *PRAPs/par/* folder. Then there’s the *PRAPs/utils/* folder, which holds all of the Python and other scripts and files that are independent of those version differences. A few files are outside of the folders, including the *adjust\_paths.py* and the README. Inside the *ser/* and *par/* folders are *praps.sh*, which runs PRAPs, *active\_learning.sh*, which runs the active learning, *accurate\_potentials.sh*, which handles robust relaxation and the basic accurate potential, and *dft\_chulls.sh*, which handles the DFT relaxation as described above. There are also .slurm files to actually run each one of those, and a sample *inpraps.sh* file.

## 6a. Utilities

This sub-section describes all of the files in the PRAPs/utils folder, except for *mliputils.py*, which is discussed in Section 6b. Many of these operate by converting .cfg files into Pandas DataFrames, manipulating those DataFrames, and then writing the output as a .cfg file again. Even though these are scripts and not functions, they are used as functions, so their arguments are given in parentheses, like functions. For variable length argument strings, a + is included.

|  |  |
| --- | --- |
| **Name** | **Description** |
| [adjust\_poscar\_coords.py](#adjust_poscar_coords) | Moves one atom slightly |
| [als-conv.py](#alsconv) | Tests active learning Convergence criteria |
| [config\_summary.sh](#config_summary) | Writes information about convex hull configurations |
| [config\_summary.py](#config_summary_py) | Interfaces with above |
| [extract\_features.py](#extract_features) | Pulls Features out of a .cfg file for DFT relaxation |
| [filter.py](#filter) | Filters .cfg files by several methods |
| [fix\_RSPG.py](#fix_RSPG) | Fixes type-element mismatching |
| [get\_RMSE.py](#get_RMSE) | Returns 2\*RMSE from a calc-errors command |
| [how\_did\_training\_end.py](#how_did_training_end) | Checks if basic training is converged |
| [ID.py](#ID) | Handles the PRAPs-ID system |
| [INCAR/KPOINTS](#INCAR) | Default VASP input files |
| [insert\_elements.py](#insert_elements) | Inserts elements into a POSCAR file |
| [insert\_mindist.py](#insert_mindist) | Updates mindist and N\_species in .mtp files |
| [lowE\_configs.py](#lowE_configs) | Filters for low-energy configurations and writes highlow |
| [make\_2D\_hulls.py](#make_2D_hulls) | Makes 2D convex hull plots. |
| [make\_plots.sh](#make_plots) | Shortcut for the convex hull plotting scripts |
| [mliputils.py](#mliputils) | Library for manipulating .cfg files |
| [old\_poscar\_to\_cfg.py](#old_pos_to_cfg) | Converts POSCAR to .cfg using basic read/write |
| [poscar\_to\_cfg.py](#pos_to_cfg) | Converts POSCAR to .cfg using mliputils |
| [pymatgen\_prep.py](#pymatgen_prep) | Helps make convex hulls via Pymatgen |
| [remove\_duplicates.py](#remove_duplicates) | Detects duplicate AFLOW prototypes in a .cfg |
| [select\_good\_pot.py](#select_good_pot) | Selects the ‘best’ Pre-Trained potential |
| [test\_train.py](#test_train) | Makes training, testing, validation sets from a .cfg |
| [tri\_phase\_points2.py](#tri_phase_points2) | Makes convex hulls for ternary systems. |
| [trivex\_masashi.py](#trivex_masashi) | Plots ternary convex hulls in Masashi-style |
| [trivex\_xiaoyu.py](#trivex_xiaoyu) | Plots ternary convex hulls in Xiaoyu-style |

**adjust\_poscar\_coords.py**(POSCAR):   
This will open the given POSCAR and move the first atom by 0.001 Å in +*x* and +*y* direction, and in the –*z* direction. This is used during *dft\_chulls.sh* to help fix certain VASP errors.

**als-conv.py**(ALS\_conv, Numbers+):  
This tests the various mathematical conditions behind the ALS-convergence criteria and returns a Boolean. Used during *active\_learning.sh* to determine when to stop the active learning.

**config\_summary.sh**(input.cfg):  
This produces a summary of a particular .cfg file by doing two things. First it expands the .cfg into POSCARs and names each of them after their AFLOW prototype, placing them into their own sub-directory within your working directory. Then it produces a file labeled *input\_summary.csv,* which contains each of the configurations present in the original file and a bit of information about them. This information includes their energy, formation enthalpy, minimum interatomic distance, cell volume, distance to the convex hull, and (if applicable) distance above the DFT convex hull. This is only used if you set CHULL=true in *inpraps.sh* and is used just after generating each convex hull using the \*\_mapped.cfg file.

**config\_summary.py**(input.cfg, elements+):  
This performs most of the work mentioned in the previous entry. In particular, it reads the *input.cfg* file, converts it to a dataframe, and processes it to produce the *input\_summary.csv* file mentioned above.

**extract\_features.py**(original.cfg, POSCARn):  
Given a POSCAR, which is labeled with a number *n*, which is the index of that POSCAR’s appearance inside *original.cfg*, this script reads the configuration’s meta-data (Features) from *original.cfg* and writes them to a file *confign.csv* where *n* in the index mentioned above. There is some juggling of 0-indexing vs. 1-indexing, please do not be alarmed. This is used during DFT relaxation, generally only if you set CHULL=true. This script allows for the features to be put back inside a .cfg file after the DFT relaxation has finished, especially important for keeping track of things via PRAPs-ID.

**filter.py**(mode, input.cfg, output.cfg, etc+):  
This applies a variety of filtering techniques to *input.cfg* and writes to *output.cfg*. If they are the same, then the input is overwritten. The additional arguments required depend on the mode selected. This is used frequently in PRAPs in many locations, but especially at the beginning of a run. Note that this does not do energy-based filtering.

* D = distance (cfg\_mindist, preferred\_mindist, preferred\_maxdist)
  + This filters a .cfg file by minimum interatomic distance. Users can get this value from MLIP’s mindist function, which finds the minimum interatomic distance present in the .cfg file across all configs. Alternatively, set the cfg\_mindist = X to have *mliputils.py* attempt to find it. If that fails, the average of the preferred\_mindist and preferred maxdist will be used and written to the .cfg file instead.
  + The script will remove all configs whose minimum interatomic distance (as recorded in their features) is less than your preferred\_mindist or greater than your preferred\_maxdist.
  + You can set the preferred\_mindist and preferred\_maxdist using the *inpraps.sh* file, but they default to 1.1 Å and 3.1 Å, respectively.
  + This is mainly used at the beginning of PRAPs to remove unphysical configurations from DFT\_CFG. You may only filter by distances or volumes in that step, not both.
* V = volume (volume\_scaling\_factor)
  + This filters by volume, as described below.
  + We assume you have a (Feature, from) in your .cfg that contains AFLOW prototype information, and that all configurations in the .cfg with the same prototype in that same (Feature, from) string are all part of the same relaxation trajectory.
  + This script removes structures whose volume is outside the range:   
     , where
    - *VSF* = Volume Scaling Factor (the argument to this script), which is set in the *inpraps.sh* file and defaults to 0.25
    - *Vf* is the volume of the final relaxation step for that structure.
    - *Vi* is the volume of a particular configuration of index *i*.
  + This is used at the beginning of a PRAPs run to remove unphysical configurations from DFT\_CFG. You can set this behavior in the *inpraps.sh* file, but it is off by default. If you turn it on, the default is 0.25
* T = trajectory ()
  + No extra arguments needed.
  + Assumes that the configuration has a (Feature, from) that contains AFLOW prototype information and that configurations with the same prototypes are from a single relaxation trajectory.
  + This keeps only the final relaxed configurations from each trajectory.
  + This may be set in the *inpraps.sh* file and is only used during the Robust Relaxation step, though you may find uses outside as well.
* G = grade (mingrade, maxgrade)
  + This filters configurations whose MLIP-assigned grades are outside the minimum and maximum.
  + This is not natively used during PRAPs as MLIP will generally do this during relaxation processes, but you are welcome to use as you desire.
* F = forces (force\_limit)
  + Removes any configuration that has at least one force term outside the range:

-*force\_limit* < *F* < *force\_limit*.

* + We recommend setting a value from 5-10 eV/Å.
  + This is set in the *inpraps.sh* file and defaults to false. If set to true, this is used at the beginning of PRAPs after the distance/volume filtration to remove configurations with undesirable force terms.

**fix\_rspg.py**(input.cfg, output.cfg)  
If you used RandSPG11 at any point in your preparation process, you should run this. RandSPG writes elements in order of quantity, which causes type-element mismatches when changing the POSCARs to .cfg files. This script fixes that mismatch. PRAPs does not natively run this because PRAPs doesn’t generate URX\_CFG.

**get\_RMSE**(mlip\_errors.txt)  
This looks at a text file containing the output of a single “mlp calc-errors” command and returns 2\*RMSE in units of energy per atom. This is used during active learning to evaluate ALS convergence criteria.

**how\_did\_training\_end.py**(training\_log.out)  
This parses the output log of a “mlp train” command placed into a separate file to see how the training process ended and returns a Boolean. This is primarily used during the basic Pre-Training step to make sure that each of the five Pre-Trained potentials finish their training process correctly before moving along.

**ID.py**(input.cfg, mode, keywords+)  
This is the main way to handle the PRAPs-ID system. More details can be found in Section 7. PRAPs uses this extensively to label configurations and keep track of their changes. The modes are I, R, IR, O, and IO. The shortest thing to say here is that when you have a new .cfg file, you should run this script in the I mode with any desired prefixes and suffixes. The R mode is for fixing Features after MLIP’s relaxation process. The O mode fixes Features after VASP’s relaxation process.

**INCAR, INCAR\_st, INCAR\_rx, KPOINTS, KPOINTS\_st**  
These are various VASP input files that we feel are reasonable for most purposes. The \_st indicates that this file is used for single-point calculations (st for static) and the \_rx indicates it is used for relaxations. If you want to use your own, you can modify these files, or set custom\_relax=true in the *inpraps.sh* file and place your own INCAR\_st and INCAR\_rx in the working directory. PRAPs uses the single-point calculations during active learning and uses the relaxations when calculating convex hulls.

**insert\_elements.py**(POSCAR, elements+)  
This places the elements into a POSCAR file. PRAPs uses this anytime we make a POSCAR, mostly during active learning and convex hull analysis. Fun fact: I think this is the only Python script with a help docstring.

**Insert\_mindist.py**(MTP, mindist+, Number\_of\_elements)  
This edits a .mtp file to add the correct minimum interatomic distance and number of elements. You could do this on your own before a PRAPs run, but I always forgot to do it. So I have a script. Note that the mindist argument is actually four space-separated strings, so if you use this outside of PRAPs, it’s supposed to be the output of the “mlp mindist” command.

**lowE\_configs.py**(input.cfg, output.cfg, mode, energy\_threshold, elements+)  
This has two functions: filter by energy and find/write the highlow data, depending on the mode used.

* F = filter
  + This looks at the energy\_threshold and the compositions present in the .cfg file. For each composition, this mode keeps the lowest energy configuration and any with *E* < *E*min + *E*threshold. Specifically, this uses energy-per-atom, not total energy.
  + The threshold can be set in the *inpraps.sh* file as the chull\_var and defaults to 0.05 eV/atom.
  + You can also set this threshold to “Hf” (but not in the inpraps file) and this mode will calculate the formation enthalpies and keep only things with negative formation enthalpies.
* H = highlow
  + This calculates the highlow data and places it into highlow.csv, appending to whatever file is already there.
  + Most of the time, this will just produce the top-ten and bottom-ten lists
  + Certain inputs will pair with others to produce a highlow score. PRAPs is built with these pairs in mind and so this part should be handled automatically. I’ll list them anyways if you want to run it manually.
    - Any file with the following strings in its name will be treated as a reference and no score will be calculated: lowE\_robust\_relaxed, filt.cfg, lowE\_vasp.
    - Any file with “\_mtp” in its name will be scored against lowE\_robust\_relaxed
    - Any file with “\_vasp” in its name will be scored against lowE\_vasp
    - Any other filename will be scored against filt.cfg
    - Run this script with the reference config first, then run it again with the scoring config to correctly generate score.
    - Everything gets stored in a dictionary, so it’s ok to build a large highlow.csv with a lot of different configs in it as long as they are all named according to the above.
  + When you run this mode, you can put something silly, like lowout.cfg for the outpt.cfg argument.
* B = both
  + Runs both the filtration and the highlow together.

**make\_2D\_hulls.py**(input.cfg+, els+, mult=mult output.svg)  
This makes a 2D convex hull plot. The first input.cfg will be plotted in black circles on the bottom layer. If you include multiple, consider adding the mult keyword as (overlay, vert, or hor). Overlay means each subsequent input.cfg will be plotted on top of the first in blue, green, magenta, cyan, and yellow dots, respectively. Vert and Hor mean each subsequent input.cfg will be plotted vertically below or horizontally to the right of the first using blue, green, magenta, cyan, and yellow circles, respectively. Including multiple inputs but omitting the mult keyword will result in the overlay style by default. To make multiple, individual plots all in black, run this for each input, separately. Note that no legend is generated for a single input.

**make\_plots.sh**()  
This is a utility script to quickly make the convex hull plots and the config summary directories in case you’ve lost the plots, but still have the .cfg files. If you don’t have the .cfg files, you should instead run PRAPs with CHK=4. While PRAPs does do everything this script does, PRAPs does not call this script natively. It is provided as a backdoor/shortcut in case you need it.

**mliputils.py**()  
This is a large Python library full of many different functions that assist in reading, writing, and manipulation of .cfg files. As it is very large and important, please refer to Section 6b. Note that during installation, if you have some special place on your system for custom Python libraries like this, and if you inform the install script of that, it will place this library for you and make sure all the paths are correct. Otherwise, this library will remain in the install directory and so will the pathing.

**old\_poscar\_to\_cfg.py**(input\_POSCAR, output.cfg)  
This script reads a POSCAR and appends it to the end of *output.cfg*. It will include (Feature, from) and (Feature, elements) tags, but not a PRAPs-ID. Despite the name, this is still used during active learning because it’s faster than the newer one (below). That’s because this script just reads the POSCAR and writes to the .cfg directly while the newer one has proper support for *mliputils.py*.

**poscar\_to\_cfg.py**(input\_POSCAR, output.cfg, sg, els, ffrom)  
This is a version of the previous script that incorporates *mliputils.py* and its dataframes. This makes it more consistent, but slower and maybe less useful. The optional arguments are keyword arguments: sg = space group, els = compound, ffrom = a preferred (Feature, from) string. If included, they get added as features to the .cfg file. Do not include spaces in any of these arguments.

**pymatgen\_prep.py**(input.cfg, els+)  
This is slightly deprecated in that PRAPs does not call Pymatgen directly, but there is legacy support for it in *mliputils.py* to assist in making ternary convex hull diagrams. The input.cfg must contain energy data and the script does not look for reference configs, it only uses the default dictionary. If you want to use this, it may take some work to update it properly.

**remove\_duplicates.py**(input.cfg, protos.txt, els+)  
This is meant for PRAPs’ internal use and is not meant for individual use. It’s main goal is to search for structures that may be duplicates in a .cfg file and outputs a string for Bash to use to remove their POSCAR files. PRAPs uses this at the start of the DFT relaxation step described above. The input.cfg file is converted to POSCARs and each POSCAR is scanned by AFLOW for their prototype labels, which are deposited in *protos.txt*. This script reads *protos.txt* and *input.cfg* and identifies the configurations that have the same composition and prototype and sends their names out for removal, keeping those in each set that have the lowest energy. This process helps to reduce the number of DFT relaxations.

**select\_good\_pot.py**(LevMTP, cmpd, cmpd\_pth)  
This is another script that is intended to be run by PRAPs. This is run at the end of the Pre-Training step to find the ‘best’ Pre-Trained potential. The full process is described in Section 4. If you are running this on your own, it should be run in the working directory after you have generated the 1-5 sub-directories and potentials. You will need highlow data from DFT\_CFG as well as highlow and error data inside each 1-5 sub-directory. By default this script is looking for the Energy-per-atom RMSE of the 1-5 potentials with respect to their validation sets, and the highlow from the 1-5 predictions with respect to DFT\_CFG.

**test\_train.py**(input.cfg, train.cfg, test.cfg, validation.cfg, N\_configs)  
This script reads input.cfg and outputs a training, testing, and validation set. NB: it’s called test-train, but the order output is train-test. This script reads through input.cfg and at each config it rolls a random number using Numpy, which decides if the config is being placed in one of the output sets or not. After a config is selected for any of the three, none of the next three consecutive configs can be selected. This was chosen so that if your data contains closely-related configs next to each other, such as from a relaxation procedure, you don’t get over-representation of those structures. The last argument, the number of configs in the input.cfg, is there so that the random number thresholds are set to the right value to ensure that there are ~800 configs in the training set and ~200 configs in the testing and validation sets.

**tri\_phase\_points2.py**(mode, input.cfg, output\_name, els+)  
This reads the input.cfg and extracts the information needed for making a ternary convex hull plot and places it into output\_name.txt or output\_name.csv (depending on the mode). The mode you choose will determine the style of the output and of the output diagrams. Use X for Xiaoyu’s style, M for Masashi’s style, or F for a forensics output (plotted in Masashi’s style). The X and M are functionally the same as they just plot the configurations that are on the convex hull or those within 52 meV/atom of the convex hull. The F mode will plot two additional plots. The first will plot all structures in input.cfg with those on or close to the hull plotted in circles. Structures with negative enthalpies of formation but not within 52 meV/atom of the hull will use triangles, and those with positive enthalpies of formation will use ×’s. Structures in this first plot will be colored by their distance to the hull. The second plot also plots all structures, but without the tie-lines, shapes, or black circles. All structures will be colored circles with the color representing their enthalpy of formation. These additional plots are available so you can tell why certain structures may have appeared on a convex hull before a DFT or MTP process, and why they are not on the hull after that process. PRAPs uses this if you set CHULL=true and defaults to Masashi’s style. Changing styles automatically in *inpraps.sh* is not currently supported.

**trivex\_masashi**(input.csv, output\_file\_extension)  
This creates a ternary convex hull diagram in Masashi’s style using the output from *tri\_phase\_points2.py*. The second argument should be the file extension/image type, but without the period: e.g., png or svg and not .png or .svg. We plot using Matplotlib, so make sure it’s a supported filetype. This script processes one file at a time.

**trivex\_xiaoyu**(input.txt+)  
This script takes input.txt files prepared by *tri\_phase\_points2.py* and makes ternary convex hull plots in Xiaoyu’s style. The script contains an example of the input.txt files it’s looking for, which includes information about the color scheme and the output file extension. As before, it’s using Matplotlib, so ensure compatibility. This script will process as many inputs as you give it with the outputs being named the same: e.g., input.txt 🡪 input.png. PRAPs does not use Xiaoyu’s style natively, but it’s still here if you want to.

## 6b. Mliputils

This is a Python library that handles reading, writing, and manipulating .cfg files. The main purpose is to read .cfg files, turn them into Pandas dataframes, manipulate those, and re-write them as .cfg files. There are some other things, like reading POSCARs, making convex hulls, and calculating energies.

|  |  |
| --- | --- |
| **Functions** | **Description** |
| [init\_configs](#init_configs) | Returns an empty Pandas MultiIndex to hold information from the .cfg files. |
| [read\_cfg\_from\_file](#read_cfg) | Reads a *.cfg* file, and returns a DataFrame. |
| [read\_json](#read_json) | Reads a *.json* file from AFLOW and returns a DataFrame. |
| [lat\_transform](#lat_transform) | Transforms a lattice with a volume greater than 1.0 Å3 into the equivalent lattice with volume of 1.0 Å3 |
| [read\_cfg\_from\_poscar](#read_poscar) | Given a structure in POSCAR file format, adds a new row in a DataFrame. |
| [write\_cfg](#write_atoms) | Writes a DataFrame to a .*cfg* file. |
| [organize\_atom](#organize_atoms) | Arranges atoms in *bc*-planes, stacked along the *a* axis. Sorts the atoms by [type, *x*, *y*, *z*]. |
| [get\_low\_E](#lowE) | Returns a new DataFrame with configurations below the energy limit. |
| [get\_comp](#get_comp) | Adds a composition column. |
| [cfg\_dist](#cfg_dist) | Returns minimum interatomic distance. |
| [clean\_df](#clean_df) | Performs energy filtering. |
| [get\_ground\_states](#get_ground_states) | Identifies minimal-energy configurations. |
| [get\_min\_endpoints\_from\_cfg](#endpts_cfg) | Returns elemental ground state energies, if present in the DataFrame. |
| [get\_min\_endpoints\_from\_els](#endpts_els) | Returns elemental ground state energies from the internal library. |
| [get\_Hf](#Hf) | Adds the enthalpy of formation column. |
| [get\_RMSE](#rmse) | Returns 2\*RMSE from MLIP’s calc-errors output. |
| [write\_highlow](#highlow) | Calculates and writes highlow data. |
| [get\_crystal](#Bravais) | Returns Bravais Lattice of a structure. |
| [get\_volumes](#volumes) | Adds a Volume column to the DataFrame. |
| [chull\_prep](#chull_prep) | Removes non-minimal energy configurations. |
| [convexhull](#chull) | Generates a convex hull using Scipy |
| [chull\_dist](#chull_dist) | Gets hull distances for configurations above the hull. |

**Global Dictionaries**:

*endpts*: A mapping of metal elements to their ground-state DFT energies. These were obtained from the AFLOW database and may not be up to date. Energies are in units of eV/atom. The dictionary contains the transition metals as well as boron, carbon, nitrogen, and oxygen.

*typedict*: Initially empty, this stores the type-element mapping from els. Example:  
els=(C Mo W)  
typedict = {C:0, Mo:1, W:2, 0:C, 1:Mo, 2:W}

**Functions:**

**init\_configs**():   
Returns an empty Pandas MultiIndex designed to hold information from the .cfg file. This object will be henceforth referred to as the cfg-df. Note that configurations will be read in as 1-indexed, not 0-indexed.

**read\_cfg\_from\_file**(filename):   
Given a filename of a .cfg file, this reads the file, invokes init configs(), and returns the filled cfg-df. Most fields are filled in by strings. A few columns will be noted for particular interest here: the AtomData columns are the equivalent columns in the .cfg file; (AtomData, cd) holds a ‘c’ for Cartesian and a ‘d’ for Direct; (Energy, eV/atom) calculates and holds the energy in eV/atom; and the Feature columns hold any Features present in the .cfg file. Note that configurations are read in as 1-indexed, not 0-indexed.

**read\_json**(filename):   
Reads a .json file that might be output from AFLOW into the cfg-df, similar to above. Note that configurations are read in as 1-indexed, not 0-indexed.

**lat\_transform**(lattice, V):   
Transforms a lattice with a volume greater than 1.0 Å3 into the equivalent lattice with volume of 1.0 Å3. Used below.

**read\_cfg\_from\_poscar**(filename, cfg=None, sg=None, els=None, ffrom=None):   
Given a POSCAR, reads it and appends it to the end of a cfg-df (if cfg is given) or makes a new cfg-df (if cfg=None). Other arguments include: sg = space group, becomes a Feature; els = elements, used to make the typedict; ffrom = (Feature, from), a string used to hold the origin of a particular configuration. (Feature, from) strings should have the following syntax:  
 “Software/Library/Prototype/Elements/Misc”  
 Software = AFLOW, VASP, MLIP, etc.  
 Library = one of AFLOW’s libraries, OUTCAR, POSCAR, etc.  
 Prototype = AFLOW prototype labels, space group, or other structure name.  
 Elements = the elements, underscore-separated.  
 Misc = Any other information, such as ionic step-number.

**write\_cfg**(cfg, filename, mode=’w’, start=None, stop=None):   
Given the cfg-df and the output filename, this writes a .cfg file from the cfg-df. Special note that if a config in the cfg-df has (AtomData, cd) = False, then it is skipped and not written. Arguments include: mode = ’w’ for write and ’a’ for append (piped into Python’s open() function); start and stop = indices in the cfg-df where users want to start and stop (start-inclusive, stop-exclusive) in case users only wish to write parts of a cfg-df and not the entire thing.

**organize\_atoms**(cfg):   
Somewhat deprecated, this arranges atoms in bc-planes, stacked along the a axis. Technically, this sorts the atoms by [type, x, y, z].

**get\_comp**(cfg, style=0, typedict=None):   
Given a cfg-df, this function reads the atom types and adds a column: (Size, Comp). This column contains the composition of each config in one of three styles. For this example, the elements are C, Mo, and W and the atomic typing is 0 0 1 2.   
 Style-0 gives “0 0.5 1 0.25 2 0.25 ”.   
 Style-1 gives “C2Mo1W1”.   
 Style-2 gives a dictionary {’C’:2, ’Mo’:1, ’W’:1}.   
 Note that styles 1 and 2 require a typedict.

**cfg\_dist**(config):   
Given a specific configuration from a cfg-df, this function returns the interatomic distances as a Numpy array via Scipy’s pdist function.

**clean\_df**(cfg):   
This removes outliers from a cfg-df using a variety of methods. Note this is not implemented as part of the standard PRAPs workflow, but users may find this helpful. The method is a string of the form "set+center+width." Set may be "comp" or "all", which indicates whether to calculate statistics for each composition present in the cfg-df indepedently or calculate one batch of statistics for the entire cfg-df. Center may be "med" for the median, "avg" for the mean, or a particular number in units of eV/atom. Width may be "sd" for standard deviation, "err" for standard error, or a number in units in of eV/atom. This function calculates the center and width with regards to (Energy, E/atom), and only keeps those configurations with energies within two widths of the center. If "comp" is set, it will do this calculation and filtration for each composition independently. If "all" is set, it will only do this calculation and filtration once for the entire cfg-df.

**get\_ground\_states**(cfg):   
Iterates through each composition in the given cfg-df and marks the minimal energy structures with (Feature, min config) = True (and False for the rest).

**get\_min\_endpoints\_from\_cfg**(cfg):   
This function makes the endpts dictionary from a given cfg-df, provided it contains elemental configurations. The endpts dictionary stores energies in eV/atom for the elements, similar to the global one described above. This is most commonly used with REF\_CFG.

**get\_min\_endpoints\_from\_els**(els):   
This function makes the endpts dictionary from the els, using the global endpoints dictionary described above. Use if you don’t have REF\_CFG.

**get\_Hf**(cfg, endpts):   
This adds the enthalpy of formation (in eV/atom) as the column (Energy, Hf) to a given cfg-df using an endpts dictionary (see above). This function requires running get comp(cfg, style=0) first, and performs the calculation given in Equation 1 above. Since the energy values are already in units of eV/atom, this function does not divide by the number of atoms present.

**get\_low\_E**(cfg, lim=0.05):   
This function looks through a given cfg-df and returns a new DataFrame with only the low-energy configurations present: lowE-df. It iterates through each composition and sets aside all configurations with energies equal to the lowest energy in that composition plus the value indicated by lim (default 0.05 eV/atom, PRAPs usually fills this in with the chull var). If the user states ’Hf’ or some other non-float value for lim, this function will grab all configurations with negative enthalpies of formation. If there is a (Feature, Pressure) column in the cfg-df, the function will re-calculate the enthalpy using the value present and then select the low-energy configurations using the newly calculated enthalpy.

**get\_RMSE**(filename):   
Given a file that contains only the error output from MLIP’s calc-errors command, this returns two times the energy root-mean-square error in eV/atom.

**write\_highlow**(cfg, cfg name, filename=’highlow.csv’, N=10):   
Goes through a cfg-df (with label cfg name) and writes the indices of the N highest and N lowest energy configurations to a highlow.csv file.

**get\_crystal**(X,Y,Z):   
Given three lattice vectors X, Y, and Z, this calculates the lattice parameters and returns the crystal family the lattice belongs to. This does not convert between primitive and conventional cells, so the output may not be what you are expecting.

**get\_volumes**(cfg):   
This adds the (Lattice, Volume) column to a cfg-df.

**chull\_prep**(cfg, remove elements=True):   
This prepares a cfg-df for convex-hull plotting by only removing all structures that are not the minimal energy structure for their composition and, if selected, removing single elements. This is intended to be used with PRAPs’ other convex-hull plotting scripts.

**convexhull**(cfg):   
Given a cfg-df, this returns the convex-hull. Note that you must run get\_Hf() first. Three objects are returned: hull, chull, and indices. Hull is the Scipy ConvexHull object, chull is the cfg-df representation, and indices is a list containing the indices of the selected hull points within the original cfg-df.

**chull\_dist**(hull, points=None):   
Given a Scipy ConvexHull object, this returns the distance-to-the-hull for each point within the hull. If a hull.points object is specified, this will calculate the distance-to-the-hull for each of those points instead. This allows a user to overlay two convex hulls and obtain hypothetical distance measures. In either case, a list will be returned of distances in eV/atom to three decimal places.

## PRAPs-ID

The PRAPs-ID system is a way for users to keep track of their configurations as they change and shift and get relaxed and predicted and so on. The system is managed by *ID.py*, which users should always run on new .cfg files or when changing them. That said, PRAPs will handle most of that for you, running *ID.py* on your input .cfg files without asking, and doing a lot in the middle of the run as well.

The PRAPs-ID takes the form: Prefixes\_ID-Suffixes, where prefixes are separated by underscores and suffixes by hyphens. The ID number in the center is the index of that configuration in the .cfg file, starting at 1. So, a hypothetical AgAu.cfg containing 100 configurations might have PRAPs-IDs that read DFT\_1, DFT\_2, DFT\_3, …, DFT\_98, DFT\_99, DFT\_100. After you use an MTP to predict their EFS data, you might re-run *ID.py* and the new PRAPs-IDs might read DFT\_1-RP, DFT\_2-RP, DFT\_3-RP, …, DFT\_98-RP, DFT\_99-RP, DFT\_100-RP. In the .cfg file, the PRAPs-IDs are stored as (Feature, PRAPs-ID), so they will be read and preserved by *mliputils.py* operations and *extract\_features.py*.

The *ID.py* file has three main modes, which can be combined for different effects: I, R, and O. The I mode will index the configurations, if not already indexed. You can use the keywords *prefix*=word and *suffix*=word to set prefixes and suffixes, which will be appended to the PRAPs-ID as appropriate. Combine with the other modes using IR and IO to use prefixes and suffixes as you like. The R mode is for relaxation, and is intended to be used after you relax something using MLIP. This mode will index the configurations if not already done, and will fix the (Feature, from) tag. The R mode will not add any suffixes, so call IR if you want to do that. PRAPs does that after relaxations to append ‘RR’ or ‘AR’ to indicate Robust Relaxation or Accurate Relaxation, respectively.

The O mode is to be used after using VASP and getting an OUTCAR. MLIP’s file conversion does not preserve any features, so you will need to run *extract\_features.py* first, and then perform your VASP calculation. After the VASP run has concluded, use MLIP to convert the OUTCAR back into a .cfg, and then run *ID.py* on that .cfg to restore all of the features. This mode has several keywords:

* *proto* = AFLOW prototype label
* *els* = comma-separated list of elements
* *cmpd* = the compound, which will be overwritten by els
* *source* = a name for (Feature, from) that should take the form “./DFT/POSCAR2”
  + The script is looking for three strings separated by /
  + The last string must have a number that corresponds to the original POSCAR and it’s index in the original .cfg
* *features* = The *configXX.csv* file produced by *extract\_features.py*. Note that the index number on this file, and the index number in the ‘source’ POSCAR above, will be off-by-one due to indexing differences. Don’t panic.
* *car* = a POSCAR or CONTCAR file with the elements in it. This is necessary to fix type-element mismatch errors
  + If you have a ternary, ABC, then MLIP will record their types as 0, 1, 2.
  + A ternary POSCAR will go in, and come out, as ABC and 0, 1, 2, but binaries won’t.
  + A binary AB will go in and come out as AB 🡪 0, 1, but a binary BC or AC will also come out as 0, 1 when it needs to be 1, 2 or 0, 2 instead.
* *last* = a number indicating the final step of the OUTCAR. Use this if you are only grabbing the final relaxation step of the OUTCAR. If you had a relaxation that ran for 12 steps, the (Feature, from) would include “relax-step=12”. If you omit this tag, we assume there are multiple steps and will index them accordingly, starting from 1.

The most important keywords for the O mode are *els*, *car*, and *features*. To include additional prefixes and suffixes, using the IO mode with the *prefix* and *suffix* keywords. Hopefully, using the PRAPs-ID system wisely will help you track your configurations as they move throughout PRAPs and beyond. If you like, you’re welcome to use or adjust the system and *ID.py* to handle MLIP’s .cfg files more generally outside of PRAPs, a note that really applies to anything in the PRAPs package.

## Final Remarks

I hope that PRAPs is, in some way or other, useful. At the beginning of this project I joked with my advisor that “PRAPs it will work, and PRAPs it won’t!” (This joke relies on the similarity between the acronym PRAPs and the word “perhaps”.) While it has been useful for our group, forming the basis of a recently published manuscript,2 we invite you to modify the parts of PRAPs that you find useful to suit your needs.

## References and Notes

[1] I. S. Novikov, K. Gubaev, E. Podryabinkin, V, A. Shapeev, V., “The MLIP Package: Moment Tensor Potentials with MPI and Active Learning,” *Mach. Learn.: Sci. Technol.* **2 (2)** (2021) 025002.

[2] J. Roberts, B. Rijal, S. Divilov, J. P. Maria, W. G. Fahrenholtz, D. E. Wolfe, D. W. Brenner, S. Curtarolo, E. Zurek, “Machine Learned Interatomic Potentials for Ternary Carbides Trained on the AFLOW Database,” *npj Comput. Mater.* **10** (2024) 142 (1–10).

[3] C. E. Calderon, J. J. Plata, C. Toher, C. Oses, O. Levy, M. Fornari, A. Natan, M. J. Mehl, G. Hart, M. B. Nardelli, S. Curtarolo, “The AFLOW Standard for High-Throughput Materials Science Calculations,” *Comp. Mater. Sci.* **108 (A)**, (2015), 233–238.

[4] G. Kresse, J. Hafner, “*Ab Initio* Molecular Dynamics for Liquid Metals,” *Phys. Rev. B.* **47 (1)**, (1993), 558.

[5] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. “Array Programming with NumPy,” *Nature* **585**, (2020), 357–362.

[6] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, CJ Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E.A. Quintero, C. R Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, “Fundamental Algorithms for Scientific Computing in Python.” *Nature Methods,* **17(3),** (2020), 261-272.

[7] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Comput. Sci. Eng.*, **9(3)**, (2007), 90-95

[8] Y. Ikeda, “MPLtern,” *Zenodo*: yuzie007/mpltern: 1.0.4, (2024) doi: 10.5281/zenodo.11068993.

[9] The pandas development team, “Pandas,” *Zenodo*: pandas-dev/pandas: 2.2.3 (2024) doi: 10.5281/zenodo.3509134

[10] C. Risko, Conversations in 2018.

[11] P. Avery, E. Zurek, “RandSpg: An Open-Source Program for Generating Atomistic Crystal Structures with Specific Spacegroups,” *Comput. Phys. Commun.,* **213**, (2018), 208-216.

1. While this text assumes that you are using the Slurm scheduler, PRAPs can be run with others. You will need to set up your own submission script, but the fundamental commands should be the same. [↑](#footnote-ref-1)