GDPy (gdpx)

Release 0.0.2

Jiayan Xu

Apr 30, 2024

INTRODUCTION:

GDPy stands for **Generating Deep Potential with Python**, including a set of tools and Python modules to automate the structure exploration and the model training for **machine learning interatomic potentials** (MLIPs). It is developed and maintained by Jiayan Xu under supervision of Prof. P. Hu at Queen's University Belfast.

CHAPTER

ONE

SUPPORTED POTENTIALS

`eann`, `deepmd`, `lasp`, `nequip` / `allegro`

CHAPTER

TWO

SUPPORTED EXPEDITIONS

`molecular dynamics`, `genetic algorithm`, `grand canonical monte carlo`, `graph-theory adsorbate configuration`, `artificial force induced reaction`

2.1 About

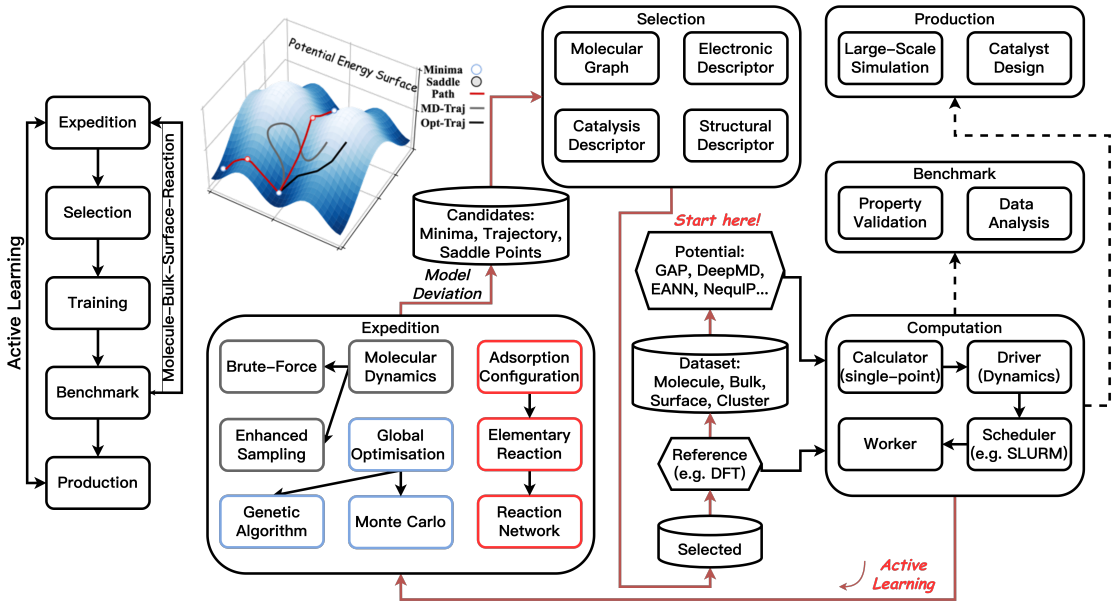
GDPy stands for **Generating Deep Potential with Python**, including a set of tools and Python modules to automate the structure exploration and the model training for **machine learning interatomic potentials** (MLIPs). It is developed and maintained by Jiayan Xu under supervision of Prof. P. Hu at Queen's University Belfast.

2.1.1 Features

- A unified interface to various MLIPs.
- Versatile exploration algorithms to construct a general dataset.
- Automation workflows for dataset construction and MLIP training.

2.1.2 Overview

The modules are:



2.2 Installation

2.2.1 Requirements

Must:

- Python 3.9
- matplotlib 3.5.0
- numpy 1.21.2
- scipy 1.7.3
- scikit-learn 1.0.1
- ase 3.22.1
- dscribe 1.2.1
- joblib 1.1.0
- tinydb 4.7.0
- pyyaml 6.0
- networkx 2.6.3
- omegaconf 2.3.0
- h5py 3.7.0

Optional:

- jax 0.2.27

- pytorch 1.10.1
- sella 2.0.2
- plumed 2.7.3

2.2.2 From Source, Conda or Pip

```
# Create a python environment

# Install the latest RELEASED version from anaconda
$ conda install gdpx -c conda-forge

# or from pypi
$ pip install gdpx

# Install the latest development version
# 1. download the MAIN branch
$ git clone https://github.com/hsulab/GDPy.git
#    or the DEV branch
$ git clone -b dev https://github.com/hsulab/GDPy.git

# 2. Use pip to install the an editable version to
#    the current environment
$ cd GDPy
$ pip install -e ./

# 3. Update the source code
$ cd GDPy
$ git fetch
$ git pull
```

2.3 Getting Started

Here, we would introduce several basic components of GDPy, namely **potential**, **driver**, **worker**. This section demonstrates how to use **gdp** to computate a number of structures.

The related commands are

```
# gdp -h for more info
$ gdp -h

# --- run simulations on local nodes or submitted to job queues
$ gdp -p ./worker.yaml compute ./structures.xyz

# - if -d option is used, results would be written to the folder `./results`
$ gdp -d ./results -p ./worker.yaml compute ./structures.xyz
```

An example input file (*worker.yaml*) is organised as follows:

```
potential:
    ... # define the backend, the model path and the specific parameters
```

(continues on next page)

(continued from previous page)

```
driver:
    ... # define the init and the run parameters of a simulation
scheduler:
    ... # define a scheduler
```

2.3.1 Units

We use the following units through all input files:

Time `fs`, Length `AA`, Energy `eV`, Force `eV/AA`.

2.3.2 Potential

We have supported several MLIP formulations based on an `AbstractPotentialManager` class to access **driver**, **expedition**, and **training** through workflows.

The example below shows how to define a **deepmd** potential using the **ase** backend in a **yaml** file:

```
# -- ase interface
potential:
    name: deepmd # name of the potential
    params: # potential-specifc params
        backend: ase # ase or lammps
        model: ./graph.pb
```

See *Potentials* section for more details.

2.3.3 Driver

After potential is defined, we need to further specify what simulation would be perfomed in the **driver** section. A driver (`AbstractDriver`) is the basic unit with an attacthed **ase** `calculators` for basic dynamics tasks, namely, minimisation, molecular dynamics and transition-state search. Through a driver, we can reuse the input file to perform the same simulation with several different backends.

The example below shows how to define a **driver** in a **yaml** file:

```
driver:
    backend: external # this means using the same backend as the calc
    task: md # molecular dynamics (md) or minimisation (min)
    init:
        md_style: nvt # thermostat NVT
        temp: 600 # temperature, Kelvin
        timestep: 1.0 # fs
    run:
        steps: 100
```

See *Driver* section for more details.

2.3.4 Scheduler

With **potential** and **driver** defined, we can run simulations on local machines (directly in the command line). However, simulations, under most circumstances, would be really heavy even by MLIPs (imagine a 10 ns molecular dynamics). The simulations would ideally be dispatched to high performace clusters (HPCs).

The example below shows how to define a **scheduler** in a **yaml** file:

```
scheduler:
    # -- currently, we only have slurm :(
    backend: slurm
    # -- scheduler script parameters
    partition: k2-hipri
    ntasks: 1
    time: "0:10:00"
    # -- environment settings
    environs: "conda activate py37\n"
```

2.3.5 Worker

Worker that combines the above components is what we use throughout various workflows to deal with computations.

The example below shows how to define a **worker** in a **yaml** file:

```
potential:
    name: deepmd # name of the potential
    backend: ase # ase or lammps
    params: # potential-specifc params
        model: ./graph.pb
driver:
    backend: external
    task: md # molecular dynamics (md) or minimisation (min)
    init:
        md_style: nvt # thermostat NVT
        temp: 600 # temperature, Kelvin
        timestep: 1.0 # fs
    run:
        steps: 100
scheduler:
    backend: slurm
    partition: k2-hipri
    ntasks: 1
    time: "0:10:00"
    environs: "conda activate py37\n"
```

to run a **nvt** simulation with given structures by **deepmd** on a **slurm** machine

```
# -- submit jobs...
#    one structure for one job
$ gdp -p ./worker.yaml compute ./frames.xyz
nframes:  2
@@@DriverBasedWorker+run
cand100 JOBID: 10206151
```

(continues on next page)

(continued from previous page)

```
cand96 JOBID: 10206152
@@@DriverBasedWorker+inspect
cand100 is running...
cand96 is running...
@@@DriverBasedWorker+inspect
cand100 is running...
cand96 is running...
@@@DriverBasedWorker+retrieve

# -- wait a few minutes...
#    if jobs are not finished, run the command would retrieve nothing
$ gdp -p ./worker.yaml worker ./frames.xyz
nframes:  2
@@@DriverBasedWorker+run
@@@DriverBasedWorker+inspect
cand100 is running...
cand96 is running...
@@@DriverBasedWorker+inspect
cand100 is running...
cand96 is running...
@@@DriverBasedWorker+retrieve

# -- retrieve results...
$ gdp -p ./worker.yaml worker ./frames.xyz
nframes:  2
@@@DriverBasedWorker+run
@@@DriverBasedWorker+inspect
cand100 is finished...
cand96 is finished...
@@@DriverBasedWorker+inspect
@@@DriverBasedWorker+retrieve
*** read-results time:   0.0280 ***
new_frames: 2 energy of the first: -92.219757
nframes: 2
statistics of total energies: min    -108.5682 max     -92.2198 avg    -100.3940
```

Note: If **scheduler** is not set in the **yaml** file, the default `LocalScheduler` would be used. In other words, the simulations would be directly run in the command line.

2.4 Potentials

We can define a **potential** in a unified input file (*worker.yaml*) for later simulation and training. The MLIP calculations are performed by **ase** `calculators` using either **python** built-in codes (PyTorch, TensorFlow) or File-IO based external codes (e.g. **lammps**).

2.4.1 Formulations

We have already implemented interfaces to the potentials below:

Suported MLIPs:

MLIPs are the major concern.

Name	Representation	Backend	Notes
eann	(Recursive) Embedded Atom	Python, LAMMPS	
lasp	Atom-Centred Symmetry Function	LASP, LAMMPS	
schnet	Graph Neueral Network	Python	
deepmd	Deep Descriptor	Python, LAMMPS	Only potential model.
nequip	E(3)-Equivalent Message Passing	Python, LAMMPS	Allegro is supported as well.
mace	Equivalent Graph Neueral Network	Python	

Note: GDPy does not implement any MLIP but offers a unified interface. Therefore, certain MLIP could not be utilised before corresponding required packages are installed correctly.

Other Potentials:

Some potentials besides MLIPs are supported. Force fields or semi-empirical potentials are used for pre-sampling to build an initial dataset. *Ab-initio* methods are used to label structures with target properties (e.g. total energy, forces, and stresses).

Name	Description	Backend	Notes
reax	Reactive Force Field	LAMMPS	
xtb	Tight Binding	xtb	
vasp	Density Functional Theory	VASP	
cp2k	Density Functional Theory	cp2k	
plumed	Collective-Variable Enhanced Sampling	plumed	

2.4.2 Simulation

We should define the potential in *worker.yaml* before running any simulation.

Basic Definition

For most potentials, **type_list** and **model** are two required parameters that are used by different backends. If the **lammps** backend is used, **command** must be set to specify how to run *lammps*. The example below shows how to define a **potential** (eann, deepmd, nequip, reax) in a **yaml** file (*worker.yaml*):

```yaml
# -- ase interface
potential:
  name: deepmd # name of the potential
  params: # potential-specifc params
    backend: ase # ase or lammps
    type_list: ["H", "O"]
    model: ./graph.pb

# -- lammps interface
potential:
  name: deepmd
  params:
    backend: lammps
    command: lmp -in in.lammps 2>&1 > lmp.out
    type_list: ["H", "O"]
    model: ./graph.pb
```

Note: Allegro can be accessed through the **nequip** potential but with an extra parameter **flavour: allegro** in the **params** section.

For **vasp**, the input can be much different as:

```yaml
potential:
  name: vasp
  params:
    # NOTE: below paths should be absolute/resolved
    pp_path: _YOUR-PSEUDOPOTENTIAL-PATH_
    vdw_path: _YOUR-VDWKERNEL-PATH_
    incar: _YOUR-INCAR-PATH_
    # - system depandent
    kpts: [1, 1, 1] # kpoints, mesh [1,1,1] or spacing 30 AA^-1
    # - run vasp
    commad: mpirun -n 32 vasp_std 2>&1 > vasp.out
```

After setting a **driver** in the input file (*worker.yaml*), we can run simulations with the defined potential. See *Driver* section for more details.

Mixing Potentials

Sometimes the simulation requires several potentials, for example, enhanced sampling. A *mixer* potentical can be defined to realise this. Currently, it only supports the *ase* backend. The parameter accepts a list of potential definitions.

The example below uses a *deepmd* model in tandem with *plumed* that adds external forces defined in an input file *./plumed.inp*.

```
potential:
  name: mixer
  params:
    backend: ase
    potters:
      - name: deepmd
        params:
          backend: ase
          type_list: ["H", "O"]
          model:
            - ./graph.pb
      - name: plumed
        params:
          backend: ase
          inp: ./plumed.inp
```

2.4.3 Training

See Trainer for more details.

2.5 Trainers

2.5.1 Related Commands

```
# - train a mode in the current directory
$ gdp train ./config.yaml

# - explore configuration space defined by `config.yaml`
#   training outputs will be written to the `m0` folder
#   a log file will be written to `m0/gdp.out` as well
$ gdp -d m0 train ./config.yaml
```

2.5.2 Configuration File

The *config.yaml* requires two sections *dataset* and *trainer* to define a training process as

```
dataset:
  name: xyz
  dataset_path: ./dataset
  train_ratio: 0.9
  batchsize: 16
```

(continues on next page)

(continued from previous page)

```
  random_seed: 1112
trainer:
  name: deepmd
  command: dp
  freeze_command: dp
  config: ./dpconfig.json
  type_list: ["H", "O"]
  train_epochs: 100
  random_seed: 1112
```

In the *dataset* section, a *dataset* is defined. **gdp** will load the structures in the dataset and convert to the proper format required by the trainer.

- name: Dataset format. (Only *xyz* is supported now.)
- dataset_path: Dataset filepath.
- train_ratio: Train-valid-split ratio.
- batchsize: Training batchsize.
- random_seed: Random seed that affects how to split structures into train and test sets.

A typical *xyz* dataset looks like

```
$ tree ./dataset
./dataset/
└── water-H128O64
    ├── init_aimd.xyz
    └── iter_dpmd.xyz
```

In the *trainer* section, a *trainer* is defined. The parameters related to the model architecture is defined in *config*, which may be different by models. **gdp** will automatically update some parameters in the *config*, which include the training dataset section and training epochs.

For example, if one is training *deepmd*, *training.training_data* and *training.validation_data* in the *./dpconfig.json* can be left empty. **gdp** will convert *dataset* into deepmd-format and update the file path. Moreover, *deepmd* uses *numb_steps* instead of *epochs*. **gdp** will compute the number of batches based on the input dataset and multiply it with *train_epochs* to give the value of *numb_steps*.

- name: Trainer target.
- commad: Command to train.
- freeze_command: Command to freeze/deploy the trained model.
- config: Model architecture configuration.
- type_list: Type list of the model.
- train_epochs: Number of training epochs.
- random_seed: Random number generator to generate random numbers in the training.

2.5.3 List of Trainers

Read notes for the training of a specific MLIP formulation.

deepmd

Warning: This trainer requires an extra package *dpdata*. Use *conda install dpdata -c deepmodeling* to install it.

gdp converts structures into the deepmd format stored in two folders *train* and *valid* based on *dataset* and writes a training configuration *deepmd.json*. The training will be performed by *dp train deepmd.json*.

Some parameters in the *deepmd.json* will be filled automatically by **gdp**. training.training_data and training.validation_data will be the folder paths generated by **gdp**. Moreover, deepmd uses numb_steps instead of epochs. **gdp** will compute the number of batches based on the input dataset and multiply it with *train_epochs* to give the value of *numb_steps*.

See DEEPMD doc for more info about configuration parameters. Example Configuration:

```
dataset:
  name: xyz
  dataset_path: ./dataset
  train_ratio: 0.9
  batchsize: 16
  random_seed: 1112
trainer:
  name: deepmd
  config: ./dpconfig.json
  type_list: ["H", "O"]
  train_epochs: 10
  random_seed: 1112
init_model: ../model.ckpt
```

Note: Deepmd Trainer in **gdp** supports a *init_model* keyword that allows one to initialise model parameters from a previous checkpoint. This is useful when training models iteratively in an active learning loop.

mace

gdp writes *./_train.xyz* and *./_test.xyz* into the training directory based on *dataset* and generates a command line based on *trainer*.

Notice some parameters are override by **gdp** based on the *dataset* and the *trainer* parameters. The *trainer.config* section will be converted to a command line as *python ./run_train.py –name='MACE_model' ...*, which is the current training command supported by MACE.

- seed: Override by *trainer.seed*
- max_num_epochs: Override by *trainer.train_epochs*.
- batch_size: Override by *dataset*.
- train_file: Override as *./_train.xyz*
- valid_file: Override as *./_test.xyz*

- valid_fraction: Always 0.
- device: Automatically detected (either cpu or cuda). No Apple Silicon!
- config_type_weights: Must be a string instead of a dictionary.

Note: Train set are data used to optimise model parameters. Validation set are data that helps us monitor the training progress and decide to save the model at which epoch. Test set are data that neither are trained nor affect our decision on the model. Some training simplifies these complex concepts and just use one *test* set for both the validation and the test purposes.

See MACE doc for more info about configuration parameters. Example Configuration:

```yaml
dataset:
  name: xyz
  dataset_path: ./dataset
  train_ratio: 0.9
  batchsize: 16
  random_seed: 1112
trainer:
  name: mace
  command: python ./run_train.py
  config: # This section can be put into a separate file e.g. `./config.yaml`
    name: MACE_model
    valid_fraction: 0.05
    config_type_weights: '{"Default": 1.0}'
    E0s: {1: -12.6261, 8: -428.5812}
    model: MACE
    default_dtype: float32
    hidden_irreps: "128x0e + 128x1o"
    r_max: 4.0
    swa: true
    start_swa: 10
    ema: true
    ema_decay: 0.99
    amsgrad: true
    restart_latest: true
  type_list: ["H", "O"]
  train_epochs: 10
  random_seed: 1112
```

Warning: If one uses *swa*, **gdp** will not check if *start_swa* is smaller than *max_num_epochs*. If *start_swa* is larger than *max_num_epochs*, there will be an error when saving the model.

2.5.4 Use Scheduler

If the training is too time-consuming, one can use *gdp session* to access a workflow that defines a training operation. See instructions in the *Session* section.

2.6 Computations

This section gives more details how to run basic simulations with different potentials using a unified input file, which is generally made up of three components. We need to define what potential to use in the **potential** section, what simulation to run in the **driver** section, and finally what **scheduler** to delegate if necessary. See Worker_Examples in the GDPy repository for prepared input files.

The related commands are

```
# gdp -h for more info
$ gdp -h

# --- run simulations on local nodes or submitted to job queues
$ gdp -p ./worker.yaml compute ./structures.xyz

# - if -d option is used, results would be written to the folder `./results`
$ gdp -d ./results -p ./worker.yaml compute ./structures.xyz
```

An example input file (*worker.yaml*) is organised as follows:

```
potential:
    ... # define the backend, the model path and the specific parameters
driver:
    ... # define the init and the run parameters of a simulation
scheduler:
    ... # define a scheduler
```

2.6.1 Potential

Potential is the engine to drive any simulation. See *Potentials* section for more details on how to define a potential.

2.6.2 Driver

The driver supports **minisation**, **molecular dynamics**, and **transition state search**. For the **driver** section, *backend*, *task*, *init*, and *run* should be specified for each simulation. If an *external* backend is used, the minimisation would use the same backend defined in the **potential** section if it is valid.

An example input file (*worker.yaml*) is

```
driver:
    backend: external # options are external, ase, lammps, and lasp
    task: min         # options are min, md, and ts
    init:
        ...           # initialisation parameters
    run:
        ...           # running parameters
```

Constraints

Constraints are of great help when simulating some systems, for instance, surfaces. There are two ways to fix atoms in structures. The constraints could be either stored in the structure file (e.g. move_mask of xyz and FractionalXYZ of xsd) or specified in *run: constraints*. If the latter one is used, the file-attached constraints would be overridden.

Constraints can be specified as:

```
# 1. similiar to lammps group definition by atom ids
run:
    # fix atoms with indices 1,2,3,4, 6,7,8, starting from 1
    constraints: "1:4 6:8"

# 2. useful for surface systems
run:
    # fix 8 atoms with the smallest z-coordinates
    # NOTE: this does not consider PBC...
    constraints: "lowest 8"
```

Minimisation

To drive a minisation, the minimal parameetrs are *steps* and *fmax*. Specific minisation algorithm can be defined in *init: min_style: ...*. The default *min_style* is *BFGS* for the *ase* backend while *fire* for the *lammps* backend.

```
driver:
    backend: external
    task: min
    init:
        min_style: bfgs
    run:
        steps: 200 # number of steps
        fmax: 0.05 # unit eV/AA, convergence criteria for atomic forces
```

Molecular Dynamics

To driver a molecular dynamics, thermostat and related parameters need to set in *init: ...*. Three thermostats are supported both by *ase* and *lammps*, which are nve, nvt and npt.

```
driver:
    backend: external
    task: md
    init:
        # 1. NVE
        md_style: nve # options are nve, nvt, and npt
        timestep: 2.0 # fs, verlet integration timestep
        # 2. NVT
        #md_style: nvt # options are nve, nvt, and npt
        #timestep: 2.0 # fs, verlet integration timestep
        #temp: 300     # Kelvin, temperature
        #Tdamp: 100    # fs, temperature control frequency
        # 3. NPT
        #md_style: nvt # options are nve, nvt, and npt
```

(continues on next page)

(continued from previous page)

```
        #timestep: 2.0 # fs, verlet integration timestep
        #temp: 300     # Kelvin, temperature
        #Tdamp: 100    # fs, Heatbath frequency
        #pres: 1.0     # atm, equilibrium pressure
        #Pdamp: 100    # fs, pressure control frequency
    run:
        steps: 200 # number of steps
```

Transition-State Search

We are working on the interface to methods of Sella using the *ase* backend and NEB using the *lammps* backend.

2.6.3 Worker

If the **scheduler** section is defined in the input file (*worker.yaml*), a worker would be created to delegate simulations to the queue. Instead of using server database, we implement a light-weight file-based database using TinyDB to manage jobs.

Currently, we only support the **slurm** scheduler. The definition is

```
scheduler:
    backend: slurm
    ...
    # SLURM-PARAMETERS
    ntasks: ...
    time: ...
    ...
    environs: "conda activte py37" # working environment setting
```

An additional keyword **batchsize** can be set in the input file as

```
batchsize: 3
potential:
    ...
driver:
    ...
scheduler:
    ...
```

which would split the input structures into groups that run as separate jobs. For example, two jobs would be submitted if we set a **batchsize** of 3 and have 5 input structures. The first job would have 3 structures and the second one would have 2 structures. The default **batchsize** is 1 that one structure would occupy one job.

2.7 Builders

Builders are several classes that generate structures. They can be defined in two categories as Builder and Modifier.

2.7.1 Related Commands

```
# - build structures based on `config.yaml`
#   results would be written to the `results` directory
$ gdp -d ./results build ./config.yaml

# - build structures based on `config.yaml`
#   some builders (modifiers) require substrates as input
#   it can be set in `config.yaml` directly or as a command argument
$ gdp -d ./results build ./config.yaml --substrates ./sub.xyz

# - build 10 structures based on `config.yaml`
#   `number` can be used for some random-based builders (modifiers)
#   otherwise, only **1** structure is randomly built.
$ gdp -d ./results build ./config.yaml --substrates ./sub.xyz --number 10
```

2.7.2 List of Builders

DimerBuilder

RandomBuilders

This section is about builders that generate random structures under certain geometric restraints.

The builder parameters:

- composition:

  A dictionary of the chemical composition to insert. This can be atoms, molecules, or a mixture of them. For example, only atoms as *{"Cu": 13}*, only molecules as *{"H2O": 3}* and mixed atoms and molecules *{"Cu": 13, "H2O": 3}*. Moreover, if the exact number of molecules is unknwon, it can be automatically determined by the density as *{"H2O": "density 0.998"}* with the unit of `g/cm^3`.

- region:

  Define the region where random atoms/molecules are put. See *Regions* for more details.

- covalent_ratio:

  The geometric restraints. The minimum and maximum multipliers for the covalent distance between atoms.

- random_seed:

  Random seed for the random generator. Use the same seed to reproduce results.

Note: Sometimes the builder will fail to generate new structures due to geometric restraints. Simply reduce the first element of *covalent_ratio* that allows structures with very close atomic distances. If this still does not work, set *test_too_far* to false, which allows sparse atomic positions to generate.

The YAML input file has the format of

Cluster

Use *cell* to define the box where the generated cluster is in even though it is not periodic. Here, a smaller region is set to put random atoms, which makes atoms more concentrated.

```yaml
# - Genreate a cluster with 13 Cu and 3 H2O.
method: random_cluster
composition:
  Cu: 13
  H2O: 3
cell: [30., 0., 0., 0., 30., 0., 0., 0., 30.]
region:
  method: lattice
  origin: [10., 10., 10.,]
  cell: [10., 0., 0., 0., 10., 0., 0., 0., 10.]
covalent_ratio: [0.6, 2.0]
test_too_far: false
random_seed: 1112
```

Surface

Generate random atoms on a substrate. This is useful to explore reconstructed (amorphous) surfaces or supported nanoparticles. The region defines the lattice vector in the x-axis and the y-axis but a cut in z-axis that has a range from 7.5 to 13.5 (7.5+6.0).

```yaml
# - Genreate a surface with 8 Cu and 3 O.
method: random_surface
substrate: ./assets/slab.xyz
composition:
  Cu: 8
  O: 3
region:
  method: surface_lattice
  origin: [0., 0., 7.5]
  cell: [5.85, 0.0, 0.0, 0.0, 4.40, 0.0, 0.0, 0.0, 6.0]
covalent_ratio: [0.4, 2.0]
test_dist_to_slab: false
test_too_far: false
random_seed: 1112
```

Bulk

Bulks have random lattice parameters. Use *cell_bounds* to set the range of angles and lengths.

```
# - Genreate a bulk with 4 Cu and 2 O.
method: random_bulk
composition:
  Cu: 4
  O: 2
cell_bounds:
  phi: [35, 145]
  chi: [35, 145]
  psi: [35, 145]
  a: [3, 50]
  b: [3, 50]
  c: [3, 50]
```

Graph

insert

```
# config.yaml
method: graph_insert
species: CO
spectators: [C, O]
sites:
    - cn: 1
      group:
          - "symbol Cu"
          - "region cube 0. 0. 0. -100. -100. 6. 100. 100. 8."
      radius: 3
      ads:
          mode: "atop"
          distance: 2.0
    - cn: 2
      group:
          - "symbol Cu"
          - "region cube 0. 0. 0. -100. -100. 6. 100. 100. 8."
      radius: 3
      ads:
          mode: "atop"
          distance: 2.0
    - cn: 3
      group:
          - "symbol Cu"
          - "region cube 0. 0. 0. -100. -100. 6. 100. 100. 8."
      radius: 3
      ads:
          mode: "atop"
          distance: 2.0
graph:
```

(continues on next page)

(continued from previous page)

```
    pbc_grid: [2, 2, 0]
    graph_radius: 2
    neigh_params:
        covalent_ratio: 1.1
        skin: 0.25
```

remove

```
# config.yaml
method: graph_remove
species: O
graph:
    pbc_grid: [2, 2, 0]
    graph_radius: 2
    neigh_params:
        covalent_ratio: 1.1
        skin: 0.25
spectators: [O]
target_group:
    - "symbol O"
    - "region surface_lattice 0.0 0.0 8.0 9.8431 0.0 0.0 0.0 10.5534 0.0 0.0 0.0 8.0"
```

exchange

```
# config.yaml
method: graph_exchange
species: Zn
target: Cr
graph:
    pbc_grid: [2, 2, 0]
    graph_radius: 2
    neigh_params:
        # AssertionError: Single atoms group into one adsorbate.
        # Try reducing the covalent radii. if it sets 1.1.
        covalent_ratio: 1.0
        skin: 0.25
spectators: [Zn, Cr]
target_group:
    - "symbol Zn Cr"
    - "region surface_lattice 0.0 0.0 8.0 9.8431 0.0 0.0 0.0 10.5534 0.0 0.0 0.0 8.0"
```

2.7.3 Related Components

Regions

A region is a space defined in the Cartesian coordinate system, which help operators access and modify atoms in a much easier way. See its application in *Monte Carlo (MC)*.

Define a region in the *yaml* input file as follows (units: Ang):

- *auto*.

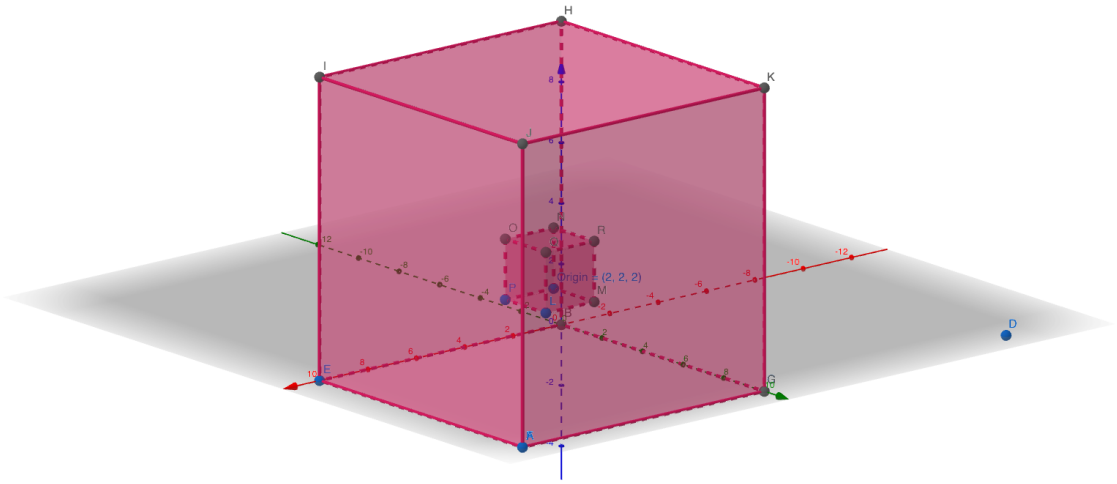
  This region takes the simulation box of the input atoms.

```
region:
  method: auto
```

- *cube*.

```
# Create a cube as ox+xl <= x <= ox+xh, oy+yl <= y <= oy+yh, oz+zl <= z <= oz+zh
region:
  method: cube
  origin: [50, 50, 50] # ox, oy, oz
  boundary: [0, 0, 0, 10, 10, 10] # xl, yl, zl, xh, yh, zh
```

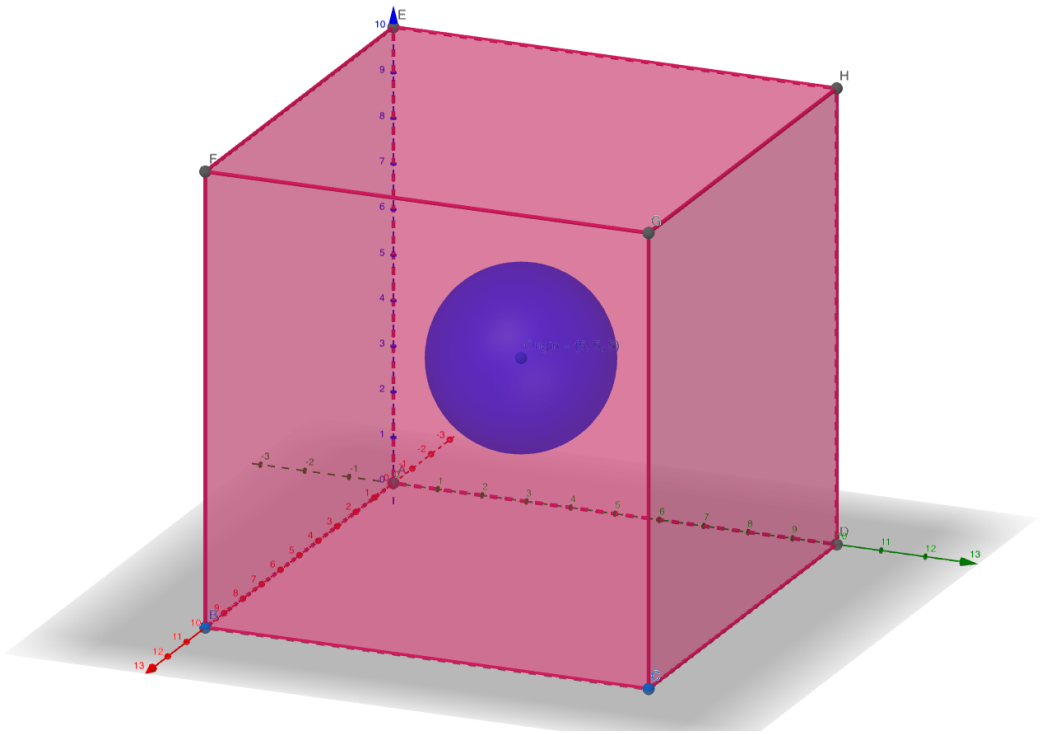
The figure shows a cubic region in a (10x10x10) simulation box. The origin is (2,2,2) and the boundary is [0,0,0,2,2,2].



- *sphere*.

```
# Create a sphere centre at [50, 50, 50] with the radius 50.
region:
  method: sphere
  origin: [50, 50, 50]
  radius: 50
```

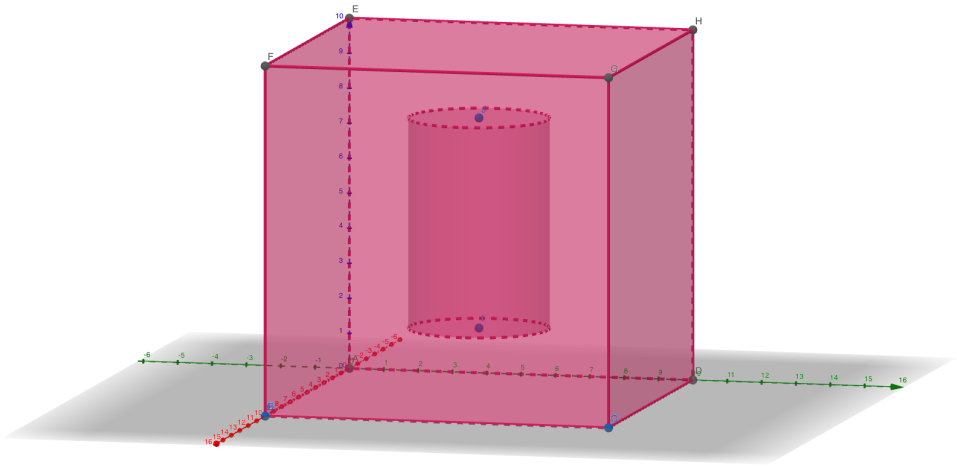
The figure shows a spherical region in a (10x10x10) simulation box. The origin is (5,5,5) and the radius is 2.



- *cylinder*.

```
# Create a vertical cylinder.
region:
  method: cylinder
  origin: [50, 50, 50]
  radius: 50
  height: 20
```

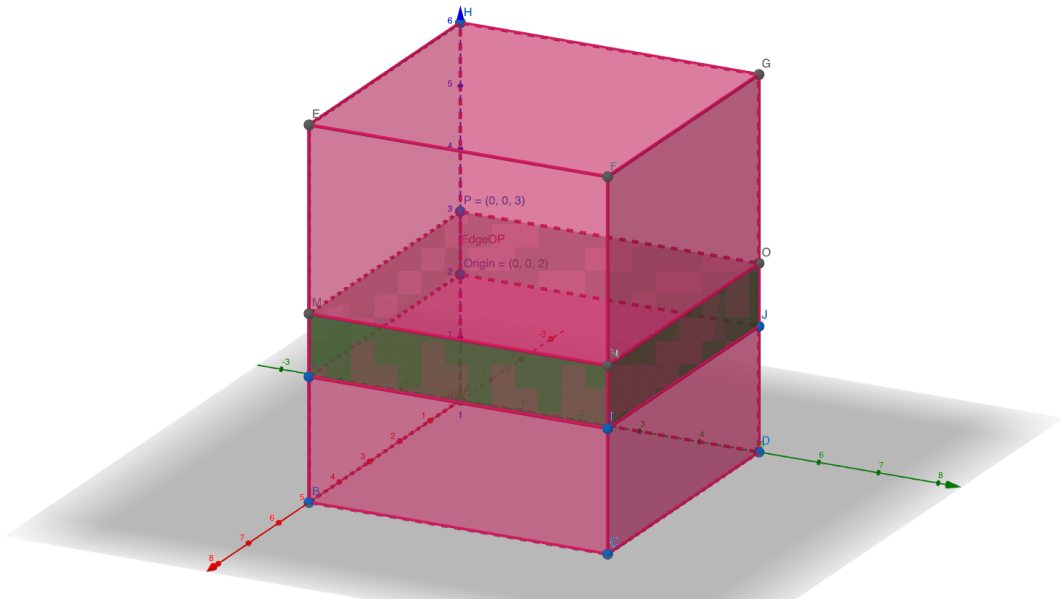
The figure shows a vertical cylinderal region in a (10x10x10) simulation box. The origin is (5,5,2), the radius is 2, and the height is 6.



- *lattice*.

```
# Create a periodic cubic lattice that is centred at [50, 50, 50].
region:
  method: lattice
  origin: [0, 0, 2]
  cell: [10, 0, 0, 0, 10, 0, 0, 0, 1]
```

The figure shows a lattice region in a (10x10x10) simulation box. The origin is (0,0,2) and the cell is [10,0,0,0,10,0,0,0,1]. The surface thickness is 1 and atoms with z-coordinate within [2, 3) will be considered as in the region. Periodic boundary condition is used for this region.



- *surface_lattice*

This has the same definition as *lattice*. However, periodic boundary condition is only applied in x- and y-axis. This is useful when only considering surface atoms.

2.8 Selections

This section gives more details how to run basic selections with different selectors using a unified input file.

The related commands are

```
# gdp -h for more info
$ gdp -h

# - results would be written to current directory
$ gdp select ./selection.yaml -s structures.xyz

# - if -d option is used, results would be written to it
$ gdp -d results select ./selection.yaml -s structures.xyz

# - accelerate the selection using 8 parallel processes,
#   which is useful for descriptor-based selection as it requires
```

(continues on next page)

(continued from previous page)

```
#   massive computations
$ gdp -nj 8 -d results select ./selection.yaml -s structures.xyz
```

The selection configuration (*./selection.yaml*) is organised as

```
selection:
  - method: property
    ... # define property and sparsification
    number: [512, 0.5]
  - method: descriptor
    ... # define descriptor and sparsification
    number: [128, 1.0]
```

This *selection* defines a sequential produces that consists of two selectors. Input structures will be first selected based on the property and the selected structures will be further selected based on the descriptor.

For the most selections, a parameter *number* is required as it determines the number of selected structures. The first value is a fixed number and the second value is a percentage. If the input dataset have 500 structures, with *number: [512, 0.5]*, 250 structures will be selected (500*0.5=250 as 500 < 512). Then, with *number: [128, 1.0]*, 128 structures will be selected (250 > 128).

After a successful selection, there are several output files. The *selected_frames.xyz* contains the final structures. Output files start with a number that indicates their oder in the list of selections. Some files, which end with *-info*, stores basic information of selected structures. For example,

```
#  index    confid      step    natoms         ene         aene        maxfrc          ␣
↪score
      0        -1         0        43    -196.7322      -4.5752       16.1094          ␣
↪0.0994
      1        -1       100        43    -242.8428      -5.6475       48.5978          ␣
↪0.0203
     87        -1      8700        43    -271.3238      -6.3099       30.7878          ␣
↪0.0164
     88        -1      8800        43    -271.2264      -6.3076       47.6111          ␣
↪0.0175
     89        -1      8900        43    -284.6631      -6.6201       64.4184          ␣
↪0.0143
     90        -1      9000        43    -303.0153      -7.0469       60.4111          ␣
↪0.0147
     91        -1      9100        43    -311.1232      -7.2354       66.2150          ␣
↪0.0120
     92        -1      9200        43    -309.4916      -7.1975       60.4200          ␣
↪0.0091
     93        -1      9300        43    -312.9583      -7.2781      149.9330          ␣
↪0.0097
     94        -1      9400        43    -314.2778      -7.3088       29.8337          ␣
↪0.0089
     95        -1      9500        43    -315.8645      -7.3457       34.8396          ␣
↪0.0114
     96        -1      9600        43    -310.2994      -7.2163       24.1396          ␣
↪0.0073
     97        -1      9700        43    -313.9329      -7.3008       29.1520          ␣
↪0.0062
```

(continues on next page)

(continued from previous page)

```
      98         -1       9800         43      -327.4579        -7.6153        14.7447        ␣
↪0.0074
      99         -1       9900         43      -330.4879        -7.6858        20.1336        ␣
↪0.0083
     100         -1      10000         43      -329.5945        -7.6650        34.6097        ␣
↪0.0055
# random_seed None
```

The first columns are structure identifiers that come from explorations, for instance, the candidate ID (confid) and the dynamics step (step) in MD or minimisation. Other notations are *natoms* - number of atoms, *ene* - total energy, *aene* - average atomic energy (ene/natoms), *maxfrc* - maximum atomic force, *score* - selection score whose meaning depends on the sparsification method. Units are in *eV* nad *eV/Ang*.

There have some other output files by specific selection method. Find details in the following subsections.

Warning: When run the same selection again, *gdp* will read the cached results (*-info.txt* files). However, it will not check whether the input structures are different from the last time. Remove output files before selection if necessary.

2.8.1 List of Selectors

Descriptor

Select structures based on descriptors.

Two sparsification methods are supported.

- cur:

  Run CUR decomposition to select the most representative structures. This method computes a CUR score for every structure and *strategy* defines the selection either performs a deterministic selection (*descent*), structures with the *number* largest scores, or a random one (*stochastic*), structures with higher scores that have higher probability. If *zeta* is larger than 0., the input descripters will be transformed as *MATMUL(descriptors.T, descriptors)^zeta*.

- fps:

  The farthest point sampling strategy. *min_distance* can be set to adjust the sparsity of selected structures in the feature (descriptor) space.

```
selection:
    - method: descriptor
    descriptor:
        name: soap
        species: ["H", "O", "Pt"]
        rcut : 6.0
        nmax : 12
        lmax : 8
        sigma : 0.3
        average : inner
        periodic : true
    sparsify:
        method: cur # fps
        zeta: -1
```

(continues on next page)

(continued from previous page)

```
        strategy: descent
    number: [16, 1.0]
```

This selection will produce a picture to visualise the distribution of structures.

Note: This requires the python package *dscribe* to be installed. Use *pip install* or *conda install dscribe -c conda-forge*.

2.8.2 Property

Select structures based on properties. The property can be total energy, atomic forces, or any properties that can be stored in the **ase** *atoms.info*. The example below demonstrates that the selection based on *max_devi_f* that is the maximum deviation of force prediction by a committee of MLIPs.

After chosing the property, there are several sparsification methods to select structures.

- filter:

  Select structures that have property within *range*. All valid structures will be selected, which is not affected by the parameter *number*.

- sort:

  Sort structures by property and select the first *number* of them. Set *reverse: True* if structures with larger property values are of interest.

- hist:

  Randomly select *number* structures based on probabilities by the histogram. For example, if 10 structures will be selected, dataset has 100 structures in bin 1 and 25 in bin 2, then roughly 8 will be from bin 1 and 2 from bin 2.

- boltz:

Randomly select *number* structures based on probabilities by the Boltzmann distribution. This is useful when selecting structures based on energy-related properties. The probabilty is computed as *exp(-p/kBT)* where *p* is the property value and *kBT* is the custom parameter in eV.

```
selection:
  - method: property
    properties:
      max_devi_f:
        range: [0.05, null]
        nbins: 20
        sparsify: filter
  - method: property
    properties:
      max_devi_f:
        range: [0.05, 0.25]
        nbins: 20
        sparsify: hist
    number: [256, 1.0]
```

The first selection on property *max_devi_f* with *filter* will give an output file below

```
#Property max_devi_f
# min 0.0304       max 17.9258
# avg 0.7199       std 0.4960
# histogram of 4914 points in the range (npoints: 5005)
      0.0500          3344
      0.9438          1547
      1.8376            11
      2.7314             2
      3.6252             4
      4.5189             3
      5.4127             1
      6.3065             0
      7.2003             0
      8.0941             0
      8.9879             0
      9.8817             0
     10.7755             0
     11.6693             0
     12.5631             0
     13.4568             1
     14.3506             0
     15.2444             0
     16.1382             0
     17.0320             1
```

There 4914 structure from 5005 have *max_devi_f* within [0.05,inf]. The rest 91 structures have a *max_devi_f* smaller than 0.05.

2.8.3 Graph

...

2.9 Expeditions

This section demonstrates several advanced methods to explore the configuration space, which are made up of basic computations. In general, an expedition is made up of three components, namely, **builder**, **worker**, and some specific parameters.

Also, an expedition progresses iteratively. The figure below demonstrates how we decouple the working ingredients and manage the communication between the expedition and the job queue (e.g. SLURM) if minimisations were not directly performed in the command line.

In every iteration, the expedition will build several new structures (either from the scratch or based on previously explored structures), and then evolves these structures into more physically reasonable ones by minimisation or molecular dynamics. This procedure produces a large number of trajectories. Applying some selections, we can extract local minima of interest and some structures from trajectories, which help the MLIP learns a comprehensive configuration space.

2.9.1 Related Commands

```
# - explore configuration space defined by `config.yaml`
#   results will be written to the `results` folder
#   a log file will be written to `results/gdp.out` as well
$ gdp -d exp explore ./config.yaml

# - or use the below command if there is `worker` section defined in `config.yaml`
$ gdp -d exp -p worker.yaml explore ./config.yaml
```

The *config.yaml* defines the specific expedition. See the page of each documentation for more information.

The *worker.yaml* defines the potential and the minimisation used through the expedition. (See *Computations* for more details.) For example, the below configuration indicates a minisation by *deepmd* using both *lammps* backends. Note set *ignore_convergence* to **true** will ignore the convergence check of the minimisation. Since the structures from the first few iterations are far away from minima i.e. they have very high potential energies, there is no need to minimise them to the full convergence. In most cases, 400 *steps* is more than enough. This setting help us reduce computation costs.

Besides, a *slurm* scheduler is set with a *batchsize* of 5. When the expedition comes across any minimisation, it will automatically submit jobs to the queue and each job will contain 5 structures as a group. When run the *gdp ... explore ...* command again, the expedition will try to retrieve the minimisation results if they are finished, and continues to run the rest procedure.

If no scheduler is set, all minimisation will run in the command line. Therefore, it is practical to write a job script with the *gdp ... explore ...* command and submit it to the queue if a large number of structures are to explore.

```
batchsize: 5
driver:
  backend: lammps
  ignore_convergence: true
```

(continues on next page)

(continued from previous page)

```
  task: min
  run:
    fmax: 0.05 # eV/Ang
    steps: 400
    constraint: lowest 120
potential:
  name: deepmd
  params:
    backend: lammps
    command: lmp -in in.lammps 2>&1 > lmp.out
    type_list: [Al, Cu, O]
    model:
      - ./graph-0.pb
      - ./graph-1.pb
      - ./graph-2.pb
      - ./graph-3.pb
scheduler:
  backend: slurm
  ntasks: 1
  cpus-per-task: 4
  time: "00:10:00"
  environs: "conda activate deepmd\n"
```

2.9.2 List of Expeditions

Monte Carlo (MC)

Overview

MC is a conventional method to explore the configuration space.

Example

The related commands are

```
# - explore configuration space defined by `config.yaml`
#   results will be written to the `results` folder
#   a log file will be written to `results/gdp.out` as well
$ gdp -d exp -p worker.yaml explore ./config.yaml

# - after MC is converged i.e. reaches the maximum number of steps,
#   the MC trajectory is stored at `results/mc.xyz`
```

In the *operators* section,

Every MC operator has parameters of *temperature*, *pressure*, and *region*. In general, these three parameters should be consistent among different operators used in the simulation. Otherwise, the simulation may not converge the structure to the phyiscal equilibrium.

To increase the acceptance, *convalent_ratio* is often set to check if the new structure has too small or too large distances. The two values are the minimum and the maximum coefficients, which will be multipied by the covalent bond distance.

See *Regions* for more information about defining a region.

- move:

  Move a particle to a random position with maximum *max_disp* displacement.

- swap:

  Swap the positions of two particles from two different types.

- exchange:

  Exchange particles with an imaginary reservoir by inserting or removing. This changes the number of atoms in the system as it samples the grand canonical ensemble.

Note: In general, operators should have the same region. Otherwise, the simulation is not converged to an equilibrium.

In the *convergence* section,

- steps: Number of MC steps.

Since MC usually takes ~5000 steps, the *dump_period* determines what MC step will be saved. For example, if *dump_period* = *2*, step 0, 2, 4 ... will be saved. These saved structures and trajectories can be used for MLIP training.

The input file shown below explores the oxidation of Cu(111) surface. The MC operators only apply to atoms in the surface region including Cu and O.

```
method: monte_carlo
random_seed: 1112
builder:
  method: reader
  fname: ./fcc-s111p44.xyz
operators:
  - method: exchange
    region:
      method: lattice
      origin: [0, 0, 8.0]
      cell: [10.17, 0, 0, 0, 8.81, 0, 0, 0, 6.0]
    covalent_ratio: [0.8, 2.0]
    reservoir:
      mu: -5.75
      species: O
    temperature: 800
    prob: 0.5
  - method: move
    particles: [Cu, O]
    region:
      method: lattice
      origin: [0, 0, 8.0]
      cell: [10.17, 0, 0, 0, 8.81, 0, 0, 0, 6.0]
    covalent_ratio: [0.8, 2.0]
    max_disp: 2.0
    temperature: 800
    prob: 0.5
convergence:
  steps: 5
dump_period: 1
```

For the *worker.yaml*, the parameter **`use_single`** must be **true** as

```
use_single: true
potential:
  name: deepmd
  params:
    backend: lammps
    command: lmp -in in.lammps 2>&1 > lmp.out
    type_list: [Cu, O]
    model:
      - ./graph.pb
driver:
  backend: lammps
  task: min
  ignore_convergence: false
  run:
    fmax: 0.05
    steps: 400
```

Application

1. **Xu, J.**; Xie, W.; Han, Y.; Hu, P. Atomistic Insights into the Oxidation of Flat and Stepped Platinum Surfaces Using Large-Scale Machine Learning Potential-Based Grand-Canonical Monte Carlo. **ACS Catal.** 2022, 12, 14812-14824.

Genetic Algorithm (GA)

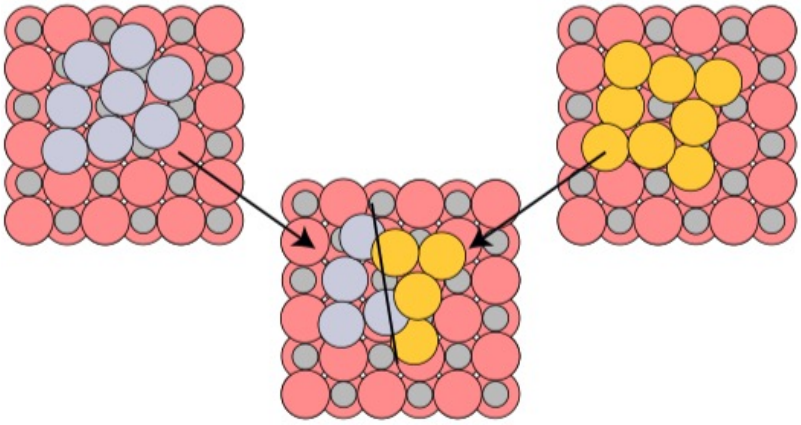
Overview

Genetic algorithm is a popular global optimisation method to find stable structures. GA in gdpy makes use of functionalities in `ase` package and provides a user-friendly interface by YAML.

The workflow of a GA-based global optimisation is

The steps are

1. Initial Population
   - Generate an initial population of structures.
   - Relax initial structures.
2. Iterate population until convergence.
   - **Selection**: Choose the best-*N* structures to form a new population.
   - **Crossover**: Use CutAndSplicePairing method to generate a new structure from two structures. This is critical to the success of GA. See the schema below. (Phys. Rev. Lett. 2012, 108, 126101.)



- **Mutation**: Each new structure (offspring) has a possibility to mutate that part of structures are modified.
- **Minimisation**: Relax new structures.
- **Convergence**: If the maximum number of generations (iterations) reaches.

3. Anaylse structures.
   - Find structures with target properties in all explored structures.

Example

To use GA, the related commands are

```
# - explore configuration space defined by `config.yaml`
#   results will be written to the `results` folder
#   a log file will be written to `results/gdp.out` as well
$ gdp -d exp -p worker.yaml explore ./config.yaml

# - after GA is converged i.e. reaches the maximum generation,
#   all found minima will be saved to `./resuslts/results/all_candidates.xyz`
```

The GA input file *./config.yaml* contains several sections:

- method: This must be *genetic_algorithm*.
- builder:

  Define the builder that generates random structures. This is used to generate an initial population of structures. See *RandomBuilders* for more details. The example builder will put 4 Cu atoms on the substrate stored in the file *./sub.xyz*. Cu atoms are randomly created in a *lattice region*, details of which can be found in *Regions*. More specific, Cu atoms will have arbitrary x- and y-coordiantes but z-coordinate within the range [7,7+6].

In the *params* section,

- database: All explored structures are stored in this file with suffix *.db*.
- property: Target property to minimise. (Only *energy* for now.)
- convergence: Convergence criteria, e.g., the maximum number of generation.
- population: Define how to organise a popultion.
  - init:

Number of structures (*size*) in the initial generation. These structures will be created by the method defined in the *builder* section.

- gen:

  Number of structures in the following generation. *size* is the total number. *random* is the number of structures generated by the initial builder while the rest is generated by the crossover operator.

- pmut:

  The probability for each reproduced structure to mutate.

- operators:
  - comparator:

    Explored structures are compared with each other. The probability of a structure to be selected as a parent is inverse of the number of its similiar structures.

  - crossover:

    This is the most critical operator in GA.

  - mutation: List of mutation operators.

    Each operator can be selected based on relative probabilities.

```
method: genetic_algorithm
builder:
  method: random_surface
  composition:
    Cu: 4
  region:
    method: lattice
    origin: [0., 0., 7.]
    cell: [11.174, 0., 0., 0., 8.413, 0., 0., 0., 6.]
  substrates: ./sub.xyz
  covalent_ratio: [0.8, 2.0]
  random_seed: 127
params:
  database: mydb.db
  population:
    init:
      size: 5
    gen:
      size: 5
      random: 2
    pmut: 0.8
  operators:
    comparator:
      dE: 0.015
      method: interatomic_distance
    crossover:
      method: cut_and_splice
    mutation:
    - method: rattle
      prob: 1.0
    - method: mirror
      prob: 1.0
```

(continues on next page)

(continued from previous page)

```
  property:
    target: energy
  convergence:
    generation: 2
```

Application

1. Lee, M.-H.; **Xu, J.**; Xie, W. Exploring the Stability of Single-Atom Catalysts Using the Density Functional Theory-Based Global Optimization Method: H2 Formation on VOx/-Al2O3(100). **J. Phys. Chem. C** 2022, 126, 6973-6981.
2. **Xu, J.**; Xie, W.; Han, Y.; Hu, P. Atomistic Insights into the Oxidation of Flat and Stepped Platinum Surfaces Using Large-Scale Machine Learning Potential-Based Grand-Canonical Monte Carlo. **ACS Catal.** 2022, 12, 14812-14824.
3. Han, Y.; **Xu, J.**; Xie, W.; Wang, Z.; Hu, P. Comprehensive Study of Oxygen Vacancies on the Catalytic Performance of Zno for CO/H(2) Activation Using Machine Learning-Accelerated First-Principles Simulations. **ACS Catal.** 2023, 13, 5104-5113.

2.10 Tutorials

We have listed several tutorials to demonstrate how to build a potential for a very specific chemical system.

2.10.1 List of Tutorials

Build a Potential for Cu Bulk with Global Search

Here, we use EMT-GA to explore structures of Cu bulks.

Build a Potential for Pt/H2O with On-the-Fly Molecular Dynamics

Here, we use deepmd to explore structures of platinum-water interface.

2.11 Sessions

To make the structure exploration and the model training much more flexible, **gdp** organises several steps of a workflow into a *session*, which is a graph flow that is made up of *variables* and *operations*. In general, *variables* are some simple components that execute for little time while *operations* require a lot of time to get the final results. One can organise a custom workflow using the combination of *variables* and *operations* through a simple YAML configuration file.

Each variable uses some parameters and sometimes other variables as inputs and creates a working component. Each *operation* accepts *variables*, *operations* and custom parameters as inputs and forwards calculation results.

2.11.1 Related Commands

```
# - run a session defined in the file `./config.yaml`
$ gdp session ./config.yaml

# - run a session defined in the file `./config.yaml`
#   fill in the placeholders in config by `--feed`
$ gdp session ./config.yaml --feed temperatures=50,150,300
```

2.11.2 Minimal Configuration

Every *session* configuration needs three sections, namely, *variables*, *operations*, and *sessions*. In the configuration below, one defines a workflow that runs molecular dynamics simulations of some structures.

For each variable, one must first set a *type* parameter. *driver*, *potter* (potential), and *scheduler* are simple variables that can be defined by several parameters, which are similiar to the definition in *Potentials*. The *computer* variable requires three variables as the input. One can **${vx:potter}** to point to the required variable. **vx** means the input is in the *variables* section and *potter* is just the variable name. At the first glance, this way of definition is a little complicated than ones uses in *gdp compute*. However, if several different *computer* are required in one workflow, *driver* and *scheduler* variables can be reusable.

The definition for *operations* is similar. One can further use **${op:read}** to point to a defined operation. **op** means the input is in the *operations* section and **read** is the operation name. Here, **read** operation reads structures from a file, which is a wrapper of the *ase.io.read* function. Then **scan** operation takes the output of **read** (structures) to run simulations defined in the **computer** variable.

sessions sets the entry point of the workflow. Here, **scan** is the name of the operation, and ALL results by variables and operations will be saved the directory *_scan*. When starting this worfklow, **gdp** checks what inputs the **scan** operation needs and executes those inputs, which forms a flow of operations as

```
**scan** <- **read**
         <- **computer** <- **potter**
                         <- **driver**
                         <- **scheduler_loc**
```

```
variables:
  computer:
    type: computer
    potter: ${vx:potter}
    driver: ${vx:driver}
    scheduler: ${vx:scheduler_loc}
  potter:
    type: potter
    name: deepmd
    params:
      backend: lammps
      command: lmp -in in.lammps 2>&1 > lmp.out
      type_list: [H, O]
      model:
        - ./graph-0.pb
        - ./graph-1.pb
  driver:
    type: driver
```

(continues on next page)

(continued from previous page)

```
    task: md
    init:
      md_style: nvt
      timestep: 0.5
      temp: [150, 300, 600] # ${placeholders.temperatures}
      dump_period: 50
    run:
      steps: 10000
  scheduler_loc:
    type: scheduler
    backend: local
operations:
  read:
    type: read_stru
    fname: ./candidates.xyz # ${placeholders.structure}
  scan:
    type: compute
    builder: ${op:read}
    worker: ${vx:_computer}
    batchsize: 256
sessions:
  _scan: scan
```

2.11.3 Variables

2.11.4 Operations

See operations.

2.12 Workflows

This section includes several oft-used *sessions* (workflows).

2.12.1 List of Workflows

Compute+Select

We can run basic computations with the operation *compute*. This operation accepts two input variables and one extra parameter and forwards *a List of Workers* that have computation results (several MD/MIN trajectories).

Two input variables,

- builder: A node (variable or operation) that forwards structures.
- worker: Any *computer* variable.

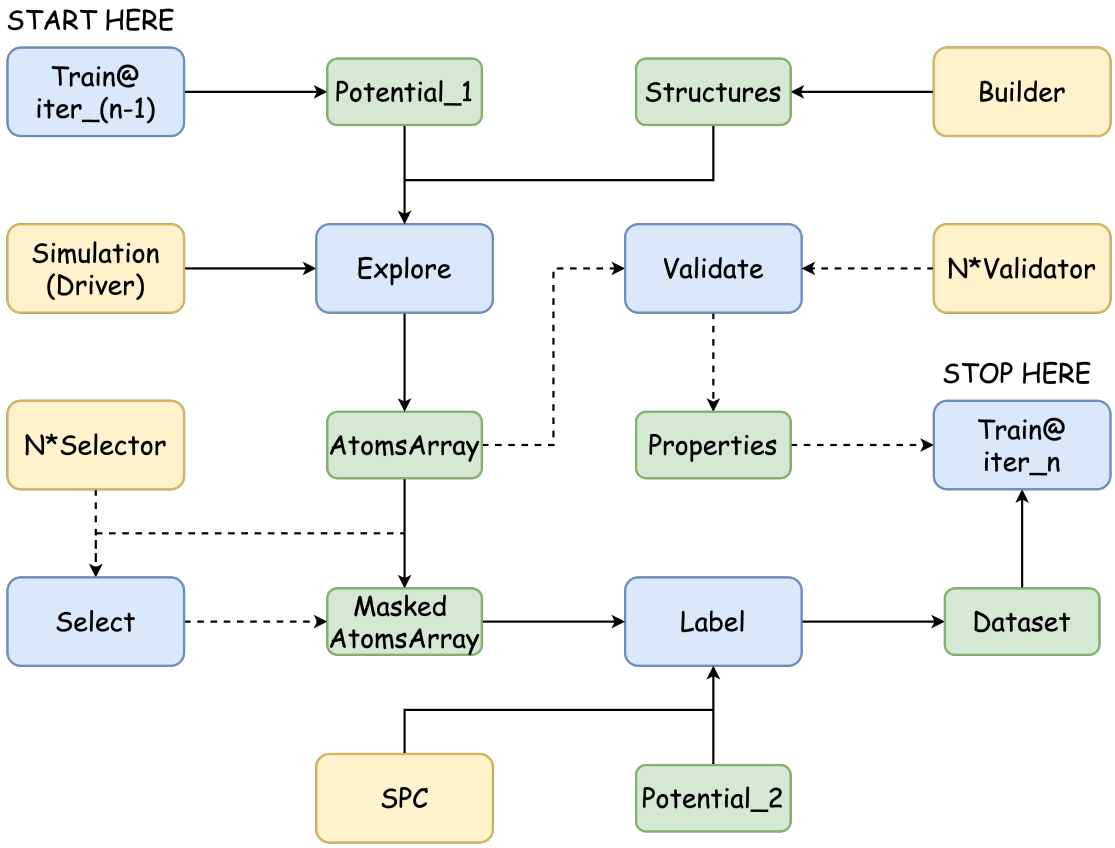
One parameter,

- batchsize: How to allocate simulations into jobs.

The *computer* variable is almost the same as what we define in *worker.yaml* shown in *Computations*. (Just change *potential* to *potter*...)

Taking *compute*'s output, we can use the *extract* operation to get the trajectories and use the *select* operation to select certain structures.

The workflow defined in the configuration below looks like



Sesscion Configuration

This configuration runs MD simulations, select some structures for DFT single-point calculations, and transfer them to the dataset.

read , the *read_stru* operation reads structures from the file, which is basically a wrapper of the *ase.io.read* function. The *./candidates.xyz* contains five structures.

scan, the *compute* operation, accepts **${op:read}** as input structures and runs simulations defined in **${vx:dpmd_computation}**. In fact, *builder* can be any variable or operation that forwards structures, which, for instance, are from builders in Section *Builders* or the extract/select by other explorations. Meanwhile, *temp* in the *driver* variable is *[150, 300, 600, 1200, 1500]*. There will be FIVE workers that run MD simulations at different temperatures.

extract, the *extract* operation, reads the trajectories by *scan* and forwards an **AtomsArray** with a shape of (5, 5, 1000). The dimensions are *number of workers*, *number of input structures*, *the length of trajectory*.

select_devi, the *select* operation, uses a *property* selector to select structures with *max_devi_f* in the range of [0.08, 0.64] eV/Ang (NOTE: The potential used should support uncertainty quantification.)

select_desc, the *select* operation, usese a *descriptor* selector to select structures using *fps* (Farthest-Point Sampling) in the *soap*-based feature space. The selection is performed on the dimension (axis) 0, which means structures from

different temperatures will be selected separately. Each group gets 64 structures and 320 (64*5) structures are selected in total.

run_vasp, another *compute* operation, takes the output of *select_desc* and perform the single-point DFT calculations.

transfer, the *transfer* operation, transfers structures calculated by DFT to a file *./dataset/${SESSION_NAME}-${COMPOSITION}-${SYSTEM}/${VERSION}.xyz*. If the input structures from *./candidates.xyz* all have a composition of Cu16. The stored xyz-file should be *./dataset/md-Cu16-surf/dpmd.xyz*

```
variables:
  dataset:
    type: dataset
    name: xyz
    dataset_path: ./dataset
  # --- computers (workers)
  dpmd_computation:
    type: computer
    potter: ${vx:dpmd}
    driver: ${vx:nvtmd}
    scheduler: ${vx:scheduler_gpu1_dpmd}
  dpmd:
    type: potter
    name: deepmd
    params:
      backend: lammps
      command: lmp -in in.lammps 2>&1 > lmp.out
      type_list: ["Al", "Cu", "O"]
      model:
        - ./graph-0.pb
        - ./graph-1.pb
  nvtmd:
    type: driver
    task: md
    init:
      md_style: nvt
      timestep: 2.0
      temp: [150, 300, 600, 1200, 1500]
      dump_period: 10
      neighbor: "2.0 bin"
      neigh_modify: "every 10 check yes"
    run:
      steps: 10000
      constraint: "lowest 120"
  scheduler_gpu1_dpmd:
    type: scheduler
    backend: slurm
    ntasks: 1
    cpus-per-task: 1
    gres: gpu:1
    mem-per-cpu: 8G
    time: "0:30:00"
    environs: "export OMP_NUM_THREADS=1\nexport KMP_WARNINGS=0\nconda activate deepmd\n"
  vasp_computation:
    type: computer
```

(continues on next page)

(continued from previous page)

```
    potter: ${vx:vasp_gam}
    driver: ${vx:driver_spc}
    scheduler: ${vx:scheduler_cpu64_vasp}
  vasp_gam:
    type: potter
    name: vasp
    params:
      backend: vasp
      command: srun vasp_gam 2>&1 > vasp.out
      incar: ./INCAR_LABEL_NoMAG
      kpts: 25
      pp_path: /home/apps/vasp/potpaw/recommend
      vdw_path: /home/apps/vasp/potpaw
  driver_spc:
    type: driver
    ignore_convergence: true
  scheduler_cpu64_vasp:
    type: scheduler
    backend: slurm
    ntasks: 64
    cpus-per-task: 1
    mem-per-cpu: 256M
    time: "24:00:00"
    environs: "export OMP_NUM_THREADS=1\nmodule purge\nmodule load intel/2021.1.2 intel-
↪mpi/intel/2021.1.1\nconda activate deepmd\n"
  # --- selectors
  sift_desc:
    type: selector
    selection:
      - method: descriptor
        axis: 0
        descriptor:
          name: soap
          species: ["Al", "Cu", "O"]
          r_cut : 6.0
          n_max : 12
          l_max : 8
          sigma : 0.2
          average : inner
          periodic : true
        sparsify:
          method: fps
          min_distance: 0.1
        number: [64, 1.0]
  sift_devi:
    type: selector
    selection:
      - method: property
        properties:
          max_devi_f:
            range: [0.08, 0.64]
            nbins: 20
```

(continues on next page)

(continued from previous page)

```
            sparsify: filter
operations:
  read:
    type: read_stru
    fname: ./candidates.xyz
  scan:
    type: compute
    builder: ${op:read}
    worker: ${vx:dpmd_computation}
    batchsize: 256
  extract:
    type: extract
    compute: ${op:scan}
  select_devi:
    type: select
    structures: ${op:extract}
    selector: ${vx:sift_devi}
  select_soap:
    type: select
    structures: ${op:select_devi}
    selector: ${vx:sift_desc}
  run_vasp:
    type: compute
    builder: ${op:select_soap}
    worker: ${vx:vasp_computation}
    batchsize: 512
  extract_dft:
    type: extract
    compute: ${op:run_vasp}
  transfer:
    type: transfer
    structures: ${op:extract_dft}
    dataset: ${vx:dataset}
    version: dpmd
    system: surf
sessions:
  md: transfer
```

Warning: If the installed **dscribe** version is < 2.0.0, you need to change the parameters *r_cut*, *n_max*, and *l_max* to *rcut*, *nmax*, and *lmax*.

Run Massive NEB Calculations

We can run reaction calculations with the operation *react* and the variable *reactor*. In the workflow below, we use Nudged Elastic Band (NEB) to calculate the reaction pathway between several structure pairs.

For a typical NEB calculation, it has the following steps:

1. Minimise the initial state (IS) and the final state (FS).
2. Align atoms in IS and FS and make sure the atom order is consistent.
3. Run NEB to converge the minimum energy path (MEP).

To run several NEB calculations at the same time, we need

1. Minimse all intermediates of interest (operations: read_stru to select_converged).
2. Run NEB calculations on selected structure pairs. (operations: pair -> react)

Node Definitions

The *reactor* variable is very similiar to *computer* that we use for minimisations and molecular dynamics. It requires a *potter* and a *driver* as well. The *potter* can be any potential interfaced with **gdp**. Currently, the *driver* only suports NEB.

The *init* section defines some parameters related to NEB.

- mic:

  Whether use the minimum image convention. If used, the smallest displacement between the IS and the FS will be found according to the periodic boundary.

- nimages:

  Number of images that define the pathway. This includes the IS and the FS that are two fixed points.

- optimiser:

  Algorithm to optimise the structures. *mdmin* is recommanded. Sometimes *bfgs* is more effective but less efficient as it needs to update the Hessian matrix. Matrix diagonalisation for Hessian is very time-consuming, O(N^3), where N is the matrix size, and it takes much more time to solve the Hessian than evaluate the forces for a large structure or a large number of images.

The *run* section is the same as the one in *computer*. *fmax* is the force convergence tolerance and *steps* is the maximum optimisation steps. Sometimes one would like NOT to minimise the IS and the FS that are pre-minimised by DFT, provided that the MLIP is not good enough. *steps* can be set to -1 that means a single-point calculation will be performed.

```
reactor:
  type: reactor
  potter: ${vx:dpmd}
  driver:
    backend: ase
    init:
      optimiser: mdmin
      mic: true
      nimages: 11
    run:
      fmax: 0.05
```

(continues on next page)

(continued from previous page)

```
      steps: 50
      constraint: "lowest 16"
```

The *locate* variable is used to get the last frame of the minimisation. If there are 3 intermediates minimised with 50 steps, the input for the selection is an AtomsArray with a shape of (3, 50). The *axis* defines on what dimension the selection is performed. The *indices* defines the indices to select on the selected dimension. In the example below, the selection is performed on the second dimension (axis: 1, the shape 50) and the last structure is selected (indices: -1).

```
locator:
  type: selector
  selection:
    - method: locate
      axis: 1
      indices: "-1"
```

The *pair* operation constructs the structure pairs for NEB. Here, the *custom* method is used. If the input *structures* has 3 structures, it will prepare two pairs, one between the first and the second structure, and another is between the second and the third structure. Therefore, two pathways will be calculated by NEB later.

```
pair:
  type: pair_stru
  structures: ${op:select_converged}
  method: custom
  pairs: # NOTE: index starts from 0!!!
    - [0, 1]
    - [1, 2]
```

Session Configuration

```
variables:
  reactor:
    type: reactor
    potter: ${vx:dpmd}
    driver:
      backend: ase
      init:
        optimiser: mdmin
        mic: true
        nimages: 11
      run:
        fmax: 0.05
        steps: 50
        constraint: "lowest 16"
  dpmd_min:
    type: computer
    potter: ${vx:dpmd}
    driver:
      task: min
      run:
        fmax: 0.05 # eV/Ang
        steps: -1 # steps: 400
```

(continues on next page)

(continued from previous page)

```
        constraint: "lowest 16"
  dpmd:
    type: potter
    name: deepmd
    params:
      backend: ase
      type_list: ["Al", "Cu", "O"]
      model:
        - ./graph.pb
  scheduler_loc:
    type: scheduler
  locator:
    type: selector
    selection:
      - method: locate
        axis: 1
        indices: "-1"
operations:
  read_stru:
    type: read_stru
    fname: ./intermediates.xyz
  run_dpmin:
    type: compute
    builder: ${op:read_stru}
    worker: ${vx:dpmd_min}
    batchsize: 512
  extract_min:
    type: extract
    compute: ${op:run_dpmin}
  select_converged:
    type: select
    structures: ${op:extract_min}
    selector: ${vx:locator}
  pair:
    type: pair_stru
    structures: ${op:select_converged}
    method: custom
    pairs: # NOTE: index starts from 0!!!
      - [0, 1]
      - [1, 2]
  react:
    type: react
    structures: ${op:pair}
    reactor: ${vx:reactor}
sessions:
  _rxn: react
```

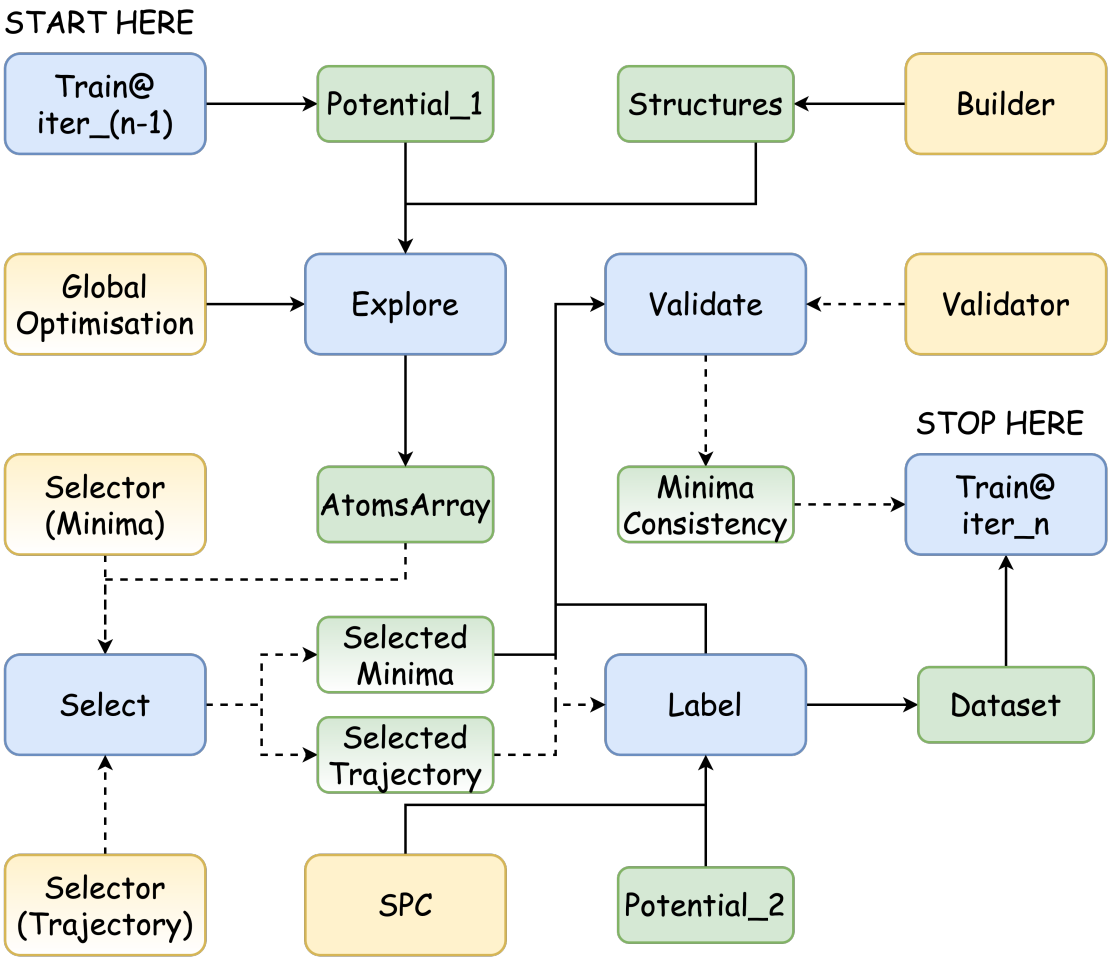
Explore with GA

We can access the *expeditions* introduced in by *explore* operation. This operation accepts two input variables and forwards *a List of Workers* (several MD/MIN trajectories). Taking its output, we can further use *selectors* to select structures of interest and use *validators* to check if the potential gave us satisfactory results,

For the input variables,

- expedition: A variable with all expedition-related parameters.
- scheduler: Where to run the expedition.

The workflow defined in the configuration below looks like



Session Confiugration

The definition of GA is similar to what we have in the *Genetic Algorithm (GA)*. The difference is that we use **{vx:builder}** and **{vx:computer}** in the *variables* section. The *worker* is the minimisation the same as the one we defined in *worker.yaml* that is no longer needed here.

After the exploration finishes, the *select* operation selects some structures from the output minimisation trajectories. In the configuration below, there are 20 trajectories in total, which is 5 structures in each generation times 4 generations (the initial generation plus 3 generations). Also, the *extract* operation sets *merge_worker* to true and thus forwards an AtomsArray with the shape of (20, ?), otherwise it is (4, 5, ?). The ? is the largest length of the minimisation trajectory.

locate selectors require two parameters:

- axis: The axis of the AtomsArray.
- indices: A string similar to python *slice*.

select_minima uses a *locate_minima* selector to first select the last structures of each trajectory and then takes 8 structures with the lowest total energies.

select_mintraj uses a *locate_mintraj* selector to select the structures of each trajectory with a step of *10*. If a trajectory has 27 structures, the structure at 0, 10, 20 steps will be selected.

With selected structures, one can further add the *compute* operation that label them with DFT or *validate* operation that analyses them.

The *seqrun* operation serves as an entry point of the worflow. It takes a list of nodes and will execute them sequentially.

```
variables:
  # --- Define the system to explore
  builder:
    type: builder
    method: random_surface
    composition:
      Cu: 19
    region:
      method: lattice
      origin: [0., 0., 7.]
      cell: [16.761, 0., 0., 0., 16.826, 0., 0., 0., 6.]
    substrates: ./surface.xyz
    # covalent_ratio: [0.8, 2.0]
    # test_too_far: false
    random_seed: 127 # Change this!
                     # Otherwise, it will generate the same initial structures.
  # --- Define a potential to use
  computer:
    type: computer
    potter: ${vx:potter}
    driver: ${vx:driver_min}
    scheduler: ${vx:scheduler_loc}
  potter:
    type: potter
    name: deepmd
    params:
      backend: lammps
      command: lmp -in in.lammps 2>&1 > lmp.out
      type_list: ["Al", "Cu", "O"]
      model:
        - ./graph-0.pb
        - ./graph-1.pb
  driver_min:
    type: driver
    task: min
    backend: lammps
    ignore_convergence: true # Allow unconvgerd trajectory
    run:
      fmax: 0.05 # 0.1
```

(continues on next page)

(continued from previous page)

```
      steps: 400
      constraint: "lowest 120"
  scheduler_loc:
    type: scheduler
  # --- Define the expedition (exploration)...
  ga:
    type: genetic_algorithm
    builder: ${vx:builder}
    worker: ${vx:computer}
    params:
      database: mydb.db # ase convert ./mydb.db mydb.xyz
      property:
        target: energy
      convergence:
        generation: 3
      population:
        init:
          size: 5
        gen:
          size: 5 # = init.size
          random: 1
        pmut: 0.8 # prob of mutation
      operators: # here, define a bunch of operators
        comparator:
          method: interatomic_distance
          dE: 0.015
        crossover: # reproduce
          method: cut_and_splice
        mutation:
          - method: rattle
            prob: 1.0
          - method: mirror
            prob: 1.0
  # --- Define some selectors...
  locate_minima:
    type: selector
    selection:
      - method: locate
        axis: 1
        indices: "-1"
      - method: property
        properties:
          energy:
            sparsify: sort
        number: [8, 1.0]
  locate_mintraj:
    type: selector
    selection:
      - method: locate
        axis: 1
        indices: ":-1:10"
operations:
```

(continues on next page)

(continued from previous page)

```
  # --- run exploration
  explore: # -> a List of workers
    type: explore # forward a list of workers
    expedition: ${vx:ga}
    scheduler: ${vx:scheduler_loc}
  # --- get minima
  extract: # -> trajectories with a shape of (20, ?)
    type: extract
    compute: ${op:explore}
    merge_workers: true
  select_minima: # -> minima
    type: select
    structures: ${op:extract}
    selector: ${vx:locate_minima}
  # --- get mintraj
  extract_mintraj: # -> trajectories with a shape of (20, ?)
    type: extract
    compute: ${op:explore}
    merge_workers: true
  select_mintraj: # -> minima
    type: select
    structures: ${op:extract_mintraj}
    selector: ${vx:locate_mintraj}
  seqrun:
    type: seqrun
    nodes:
      - ${op:select_minima}
      - ${op:select_mintraj}
sessions:
  _ga: seqrun
```

Note: Since initial structures are randomly created, they may have very small atomic distances. The minimisation may not converged even after hundreds of steps. Set **ignore_convergence** in the **driver** to allow the unconverged trajectory. The LAST structure in the trajectory will be accepted by GA to reproduce structures in the following generations.

Train

We can access the training by a *train* operation. This operation accepst four input variables and forwards a *potter* (AbstractPotentialManager) object.

For the input variables,

- potter:

  The potential manager. See *Potentials* for more details.

- dataset:

  The dataset. See *Trainers* for more details.

- trainer:

  The trainer configuration that defines the commands and the model configuration.

- scheduler:

  Any scheduler. In general, the training needs a GPU-scheduler.

Note: The name in *potter* and *trainer* should be the same.

Extra parameters,

- size:

  Number of models trained at the same time. This is useful when a committee needs later for uncertainty estimation.

- init_models:

  A List of model checkpoints to initialise model parameters. The number should be the same as size.

Session Configuration

```
variables:
  dataset:
    type: dataset
    name: xyz
    dataset_path: ./dataset
    train_ratio: 0.9
    batchsize: 16
    # random_seed: 1112 # Set this if one wants to reproduce results
  potter:
    type: potter
    name: deepmd
    params:
      backend: lammps
      command: "lmp -in in.lammps 2>&1 > lmp.out"
      type_list: ["H", "O"]
  trainer:
    type: trainer
    name: deepmd
    command: dp
    config: ${json:./config.json}
    train_epochs: 500
    # random_seed: 1112 # Set this if one wants to reproduce results
  scheduler_gpu:
    type: scheduler
    backend: slurm
    partition: k2-gpu
    time: "6:00:00"
    ntasks: 1
    cpus-per-task: 4
    mem-per-cpu: 4G
    gres: gpu:1
    environs: "conda activate deepmd\n"
operations:
  train:
```

(continues on next page)

(continued from previous page)

```
    type: train
    potter: ${vx:potter}
    dataset: ${vx:dataset}
    trainer: ${vx:trainer}
    scheduler: ${vx:scheduler_gpu}
    size: 4
    init_models:
      - ./model.ckpt
sessions:
  _train: train
```

Validate

**Validate* operations requires *validator*, *structures* (dataset), and *worker* (optional) as inputs.

```
variables:
    ...
operations:
    ...
sessions:
    ...
```

Add Correction to Computed Structures

Use *gdp session* to run a worflow add energy/forces correction to computed structures. The *correct* operation needs two input variables and forwards a *Tempdata* variable (See Dataset section for more details).

For the input variables,

- structures: A *Tempdata* variable.
- computer: A *computer* variable.

The example below adds *DFT-D3* correction to a dataset of a H2O molecule. The output cache is saved *./_corr/0002.corr_dftd3/merged.xyz*. The structures have energy/forces equal *origin+dftd3*.

Example Configuration

```
variables:
  dataset:
    type: tempdata
    system_dirs:
      - ./min-H2O-molecule
  # ---
  spc_dftd3:
    type: computer
    potter:
      name: dftd3
      params:
        backend: ase
```

(continues on next page)

(continued from previous page)

```
        method: PBE # xc
        damping: d3bj
operations:
  corr_dftd3:
    type: correct
    structures: ${vx:dataset}
    computer: ${vx:spc_dftd3}
sessions:
  _corr: corr_dftd3
```

Note: *DFT-D3* computation requires python packge *dftd3-python*. Use *conda install dftd3-python -c conda-forge* if one does not have it.

2.13 Extensions

This section is about how to extend GDPy with custom python files.

2.13.1 Custom Potential

First we define a class named `EmtManager` that is a subclass of `AbstractPotentialManager` in `emt.py`. We need to implement two attributes (`implemented_backends` and `valid_combinations`) and one method (`register_calculator`). Here, we only implement one backend that uses built-in EMT calculator in **ase**.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*

from ase.calculators.emt import EMT

from GDPy.potential.manager import AbstractPotentialManager

class EmtManager(AbstractPotentialManager):

    name = "emt"
    implemented_backends = ["ase"]

    valid_combinations = [
        ["ase", "ase"]
    ]


    def register_calculator(self, calc_params, *args, **kwargs):
        super().register_calculator(calc_params)

        if self.calc_backend == "ase":
            calc = EMT()

        self.calc = calc
```

(continues on next page)

(continued from previous page)

```
        return

if __name__ == "__main__":
    pass
```

Then we can use EMT through `pot.yaml`.

```
potential:
    name: ./emt.py # lowercase
    params:
        backend: ase
driver:
    backend: external
    task: min
    run:
        fmax: 0.05
        steps: 10
```

At last, we optimise a **H2O** molecule with **EMT**. The results are stored in the directory **cand0**.

```
$ gdp driver ./pot.yaml -s H2O
nframes:  1
potter:  emt
*** run-driver time:   0.1517 ***
[1.8792752663147125]
```

2.14 Applications

1. Lee, M.-H.; **Xu, J.**; Xie, W. Exploring the Stability of Single-Atom Catalysts Using the Density Functional Theory-Based Global Optimization Method: H2 Formation on VOx/-Al2O3(100). **J. Phys. Chem. C** 2022, 126, 6973-6981.
2. **Xu, J.**; Xie, W.; Han, Y.; Hu, P. Atomistic Insights into the Oxidation of Flat and Stepped Platinum Surfaces Using Large-Scale Machine Learning Potential-Based Grand-Canonical Monte Carlo. **ACS Catal.** 2022, 12, 14812-14824.
3. Han, Y.; **Xu, J.**; Xie, W.; Wang, Z.; Hu, P. Comprehensive Study of Oxygen Vacancies on the Catalytic Performance of Zno for CO/H(2) Activation Using Machine Learning-Accelerated First-Principles Simulations. **ACS Catal.** 2023, 13, 5104-5113.