DPGEN2

DeepModeling

Oct 04, 2022

USER GUIDE

DPGEN2 is the 2nd generation of the Deep Potential GENerator.

Important: The project DeePMD-kit is licensed under GNU LGPLv3.0.

CHAPTER

ONE

COMMAND LINE INTERFACE

DPGEN2: concurrent learning workflow generating the machine learning potential energy models.

```
usage: dpgen2 [-h] [--version] {submit,resubmit,status} ...
```

1.1 Named Arguments

--version show program's version number and exit

1.2 Valid subcommands

command Possible choices: submit, resubmit, status

1.3 Sub-commands

1.3.1 submit

Submit DPGEN2 workflow

```
dpgen2 submit [-h] CONFIG
```

Positional Arguments

CONFIG the config file in json format defining the workflow.

1.3.2 resubmit

Submit DPGEN2 workflow resuing steps from an existing workflow

```
dpgen2 resubmit [-h] [--list] [--reuse REUSE [REUSE ...]] CONFIG ID
```

Positional Arguments

CONFIG the config file in json format defining the workflow.

ID the ID of the existing workflow.

Named Arguments

--list list the Steps of the existing workflow.

Default: False

--reuse specify which Steps to reuse.

1.3.3 status

Print the status (stage, iteration, convergence) of the DPGEN2 workflow

```
dpgen2 status [-h] CONFIG ID
```

Positional Arguments

CONFIG the config file in json format.

ID the ID of the existing workflow.

CHAPTER

TWO

OP CONFIGS

2.1 RunDPTrain

init_model_start_pref_v:

type: `float`, optional, default: `0.0`
argument path: `init_model_start_pref_v`

The start virial prefactor in loss when init-model

init_model_start_pref_f:

type: `int` | `float`, optional, default: `100`
argument path: `init_model_start_pref_f`

The start force prefactor in loss when init-model

init_model_start_pref_e:

type: `float`, optional, default: `0.1`
argument path: `init_model_start_pref_e`

The start energy prefactor in loss when init-model

init_model_start_lr:

type: `float`, optional, default: `0.0001`
argument path: `init_model_start_lr`

The start learning rate when init-model

init_model_numb_steps:

type: `int`, optional, default: `400000`, alias: *init_model_stop_batch*
argument path: `init_model_numb_steps`

The number of training steps when init-model

init_model_old_ratio:

type: `float`, optional, default: `0.9`
argument path: `init_model_old_ratio`

The frequency ratio of old data over new data

init_model_policy:

type: str, optional, default: no
argument path: init_model_policy

The policy of init-model training. It can be

- 'no': No init-model training. Traing from scratch.
- 'yes': Do init-model training.
- 'old_data_larger_than:XXX': Do init-model if the training data size of the previous model is larger than XXX. XXX is an int number.

2.2 RunLmp

command:

type: str, optional, default: lmp
argument path: command

The command of LAMMPS

2.3 RunVasp

out:

type: str, optional, default: data
argument path: out

The output dir name of labeled data. In *deepmd/npy* format provided by *dpdata*.

log:

type: str, optional, default: vasp.log
argument path: log

The log file name of VASP

command:

type: str, optional, default: vasp
argument path: command

The command of VASP

CHAPTER

THREE

DEVELOPERS' GUIDE

- The concurrent learning algorithm
- Overview of the DPGEN2 implementation
- The DPGEN2 workflow
- How to contribute

3.1 The concurrent learning algorithm

DPGEN2 implements the concurrent learning algorithm named DP-GEN, described in this paper. It is noted that other types of workflows, like active learning, should be easily implemented within the infrastructure of DPGEN2.

The DP-GEN algorithm is iterative. In each iteration, four steps are consecutively executed: training, exploration, selection, and labeling.

1. **Training**. A set of DP models are trained with the same dataset and the same hyperparameters. The only difference is the random seed initializing the model parameters.
2. **Exploration**. One of the DP models is used to explore the configuration space. The strategy of exploration highly depends on the purpose of the application case of the model. The simulation technique for exploration can be molecular dynamics, Monte Carlo, structure search/optimization, enhanced sampling, or any combination of them. Current DPGEN2 only supports exploration based on molecular simulation platform LAMMPS.
3. **Selection**. Not all the explored configurations are labeled, rather, the model prediction errors on the configurations are estimated by the ***model deviation***, which is defined as the standard deviation in predictions of the set of the models. The critical configurations with large and not-that-large errors are selected for labeling. The configurations with very large errors are not selected because the large error is usually caused by non-physical configurations, e.g. overlapping atoms.
4. **Labeling**. The selected configurations are labeled with energy, forces, and virial calculated by a method of first-principles accuracy. The usually used method is the density functional theory implemented in VASP, Quantum Expresso, CP2K, and etc.. The labeled data are finally added to the training dataset to start the next iteration.

In each iteration, the quality of the model is improved by selecting and labeling more critical data and adding them to the training dataset. The DP-GEN iteration is converged when no more critical data can be selected.

3.2 Overview of the DPGEN2 Implementation

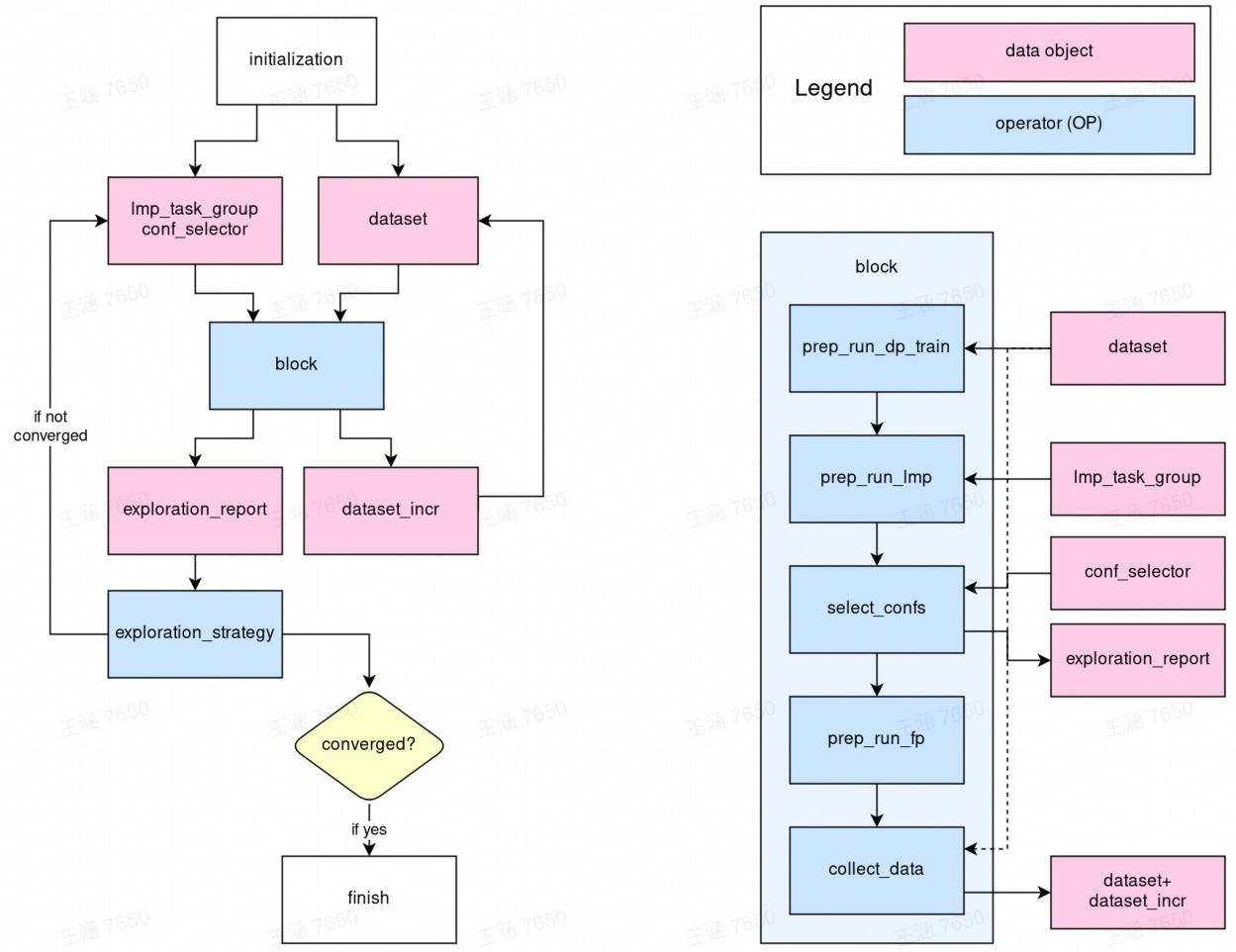
The implementation DPGEN2 is based on the workflow platform dflow, which is a python wrapper of the Argo Workflows, an open-source container-native workflow engine on Kubernetes.

The DP-GEN algorithm is conceptually modeled as a computational graph. The implementation is then considered as two lines: the operators and the workflow.

1. **Operators**. Operators are implemented in Python v3. The operators should be implemented and tested ***without*** the workflow.
2. **Workflow**. Workflow is implemented on dflow. Ideally, the workflow is implemented and tested with all operators mocked.

3.3 The DPGEN2 workflow

The workflow of DPGEN2 is illustrated in the following figure



In the center is the `block` operator, which is a super-OP (an OP composed by several OPs) for one DP-GEN iteration, i.e. the super-OP of the training, exploration, selection, and labeling steps. The inputs of the `block` OP are `lmp_task_group`, `conf_selector` and `dataset`.

- `lmp_task_group`: definition of a group of LAMMPS tasks that explore the configuration space.

- `conf_selector`: defines the rule by which the configurations are selected for labeling.
- `dataset`: the training dataset.

The outputs of the `block` OP are

- `exploration_report`: a report recording the result of the exploration. For example, home many configurations are accurate enough and how many are selected as candidates for labeling.
- `dataset_incr`: the increment of the training dataset.

The `dataset_incr` is added to the training `dataset`.

The `exploration_report` is passed to the `exploration_strategy` OP. The `exploration_strategy` implements the strategy of exploration. It reads the `exploration_report` generated by each iteration (`block`), then tells if the iteration is converged. If not, it generates a group of LAMMPS tasks (`lmp_task_group`) and the criteria of selecting configurations (`conf_selector`). The `lmp_task_group` and `conf_selector` are then used by `block` of the next iteration. The iteration closes.

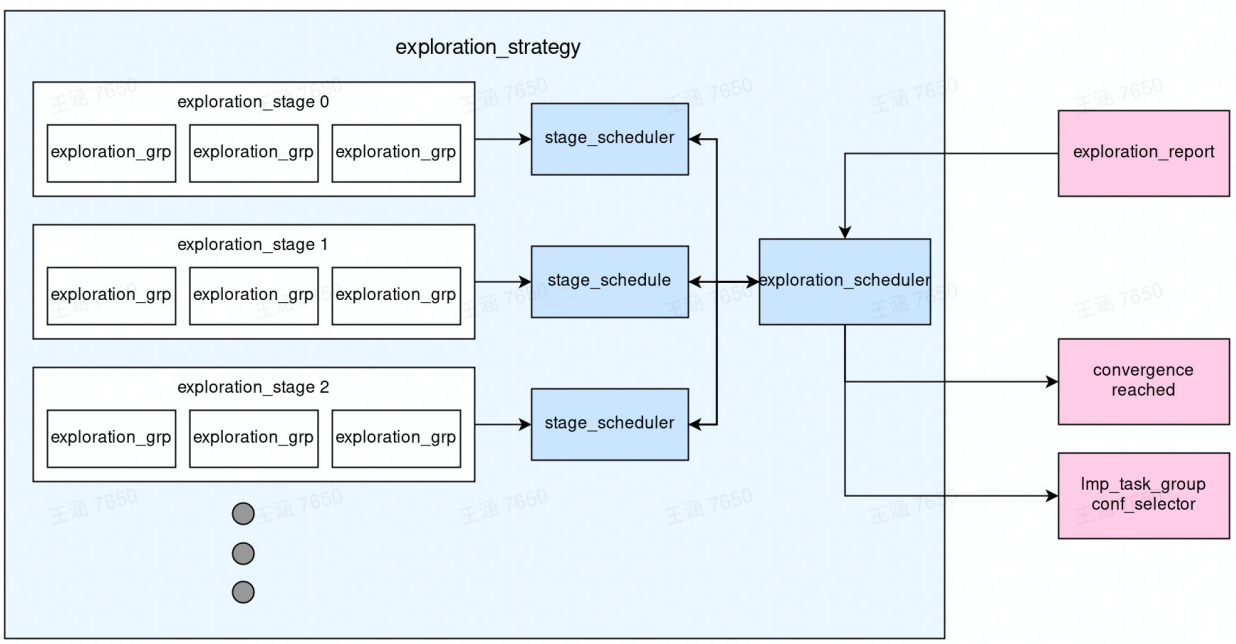
3.3.1 Inside the `block` operator

The inside of the super-OP `block` is displayed on the right-hand side of the figure. It contains the following steps to finish one DPGEN iteration

- `prep_run_dp_train`: prepares training tasks of DP models and runs them.
- `prep_run_lmp`: prepares the LAMMPS exploration tasks and runs them.
- `select_confs`: selects configurations for labeling from the explored configurations.
- `prep_run_fp`: prepares and runs first-principles tasks.
- `collect_data`: collects the `dataset_incr` and adds it to the `dataset`.

3.3.2 The exploration strategy

The exploration strategy defines how the configuration space is explored by the concurrent learning algorithm. The design of the exploration strategy is graphically illustrated in the following figure. The exploration is composed of stages. Only the DP-GEN exploration is converged at one stage (no configuration with a large error is explored), the exploration goes to the next iteration. The whole procedure is controlled by `exploration_scheduler`. Each stage has its schedule, which talks to the `exploration_scheduler` to generate the schedule for the DP-GEN algorithm.



Some concepts are explained below:

- **Exploration group**. A group of LAMMPS tasks shares similar settings. For example, a group of NPT MD simulations in a certain thermodynamic space.
- **Exploration stage**. The `exploration_stage` contains a list of exploration groups. It contains all information needed to define the `lmp_task_group` used by the `block` in the DP-GEN iteration.
- **Stage scheduler**. It guarantees the convergence of the DP-GEN algorithm in each `exploration_stage`. If the exploration is not converged, the `stage_scheduler` generates `lmp_task_group` and `conf_selector` from the `exploration_stage` for the next iteration (probably with a different initial condition, i.e. different initial configurations and randomly generated initial velocity).
- **Exploration scheduler**. The scheduler for the DP-GEN algorithm. When DP-GEN is converged in one of the stages, it goes to the next stage until all planned stages are used.

3.4 How to contribute

Anyone interested in the DPGEN2 project may contribute OPs, workflows, and exploration strategies.

- To contribute OPs, one may check the *guide on writing operators*
- To contribute workflows, one may take the DP-GEN workflow as an example. It is implemented in dpgen2/flow/dpgen_loop.py and tested with all operators mocked in test/test_dpgen_loop.py
- To contribute the exploration strategy, one may check the *guide on writing exploration strategies*

CHAPTER

FOUR

OPERATORS

There are two types of OPs in DPGEN2

- OP. An execution unit the the workflow. It can be roughly viewed as a piece of Python script taking some input and gives some outputs. An OP cannot be used in the `dflow` until it is embedded in a super-OP.
- Super-OP. An execution unite that is composed by one or more OP and/or super-OPs.

Techinically, OP is a Python class derived from `dflow.python.OP`. It serves as the `PythonOPTemplate` of `dflow.Step`.

The super-OP is a Python class derived from `dflow.Steps`. It contains `dflow.Steps` as building blocks, and can be used as OP template to generate a `dflow.Step`. The explanation of the concepts `dflow.Step` and `dflow.Steps`, one may refer to the manual of dflow.

4.1 The super-OP `PrepRunDPTrain`

In the following we will take the `PrepRunDPTrain` super-OP as an example to illustrate how to write OPs in DPGEN2.

`PrepRunDPTrain` is a super-OP that prepares several DeePMD-kit training tasks, and submit all of them. This super-OP is composed by two `dflow.Steps` building from `dflow.python.OPs` `PrepDPTrain` and `RunDPTrain`.

```
from dflow import (
    Step,
    Steps,
)
from dflow.python import(
    PythonOPTemplate,
    OP,
    Slices,
)

class PrepRunDPTrain(Steps):
    def __init__(
            self,
            name : str,
            prep_train_op : OP,
            run_train_op : OP,
            prep_train_image : str = "dflow:v1.0",
            run_train_image : str = "dflow:v1.0",
    ):
                ...
```

(continues on next page)

(continued from previous page)

```
        self = _prep_run_dp_train(
            self,
            self.step_keys,
            prep_train_op,
            run_train_op,
            prep_train_image = prep_train_image,
            run_train_image = run_train_image,
        )
```

The construction of the PrepRunDPTrain takes prepare-training OP and run-training OP and their docker images as input, and implemented in internal method _prep_run_dp_train.

```
def _prep_run_dp_train(
        train_steps,
        step_keys,
        prep_train_op : OP = PrepDPTrain,
        run_train_op : OP = RunDPTrain,
        prep_train_image : str = "dflow:v1.0",
        run_train_image : str = "dflow:v1.0",
):
    prep_train = Step(
        ...
        template=PythonOPTemplate(
            prep_train_op,
            image=prep_train_image,
            ...
        ),
        ...
    )
    train_steps.add(prep_train)

    run_train = Step(
        ...
        template=PythonOPTemplate(
            run_train_op,
            image=run_train_image,
            ...
        ),
        ...
    )
    train_steps.add(run_train)

    train_steps.outputs.artifacts["scripts"]._from = run_train.outputs.artifacts["script
↪"]
    train_steps.outputs.artifacts["models"]._from = run_train.outputs.artifacts["model"]
    train_steps.outputs.artifacts["logs"]._from = run_train.outputs.artifacts["log"]
    train_steps.outputs.artifacts["lcurves"]._from = run_train.outputs.artifacts["lcurve
↪"]

    return train_steps
```

In _prep_run_dp_train, two instances of dflow.Step, i.e. prep_train and run_train, generated from prep_train_op and run_train_op, respectively, are added to train_steps. Both of prep_train_op and

`run_train_op` are OPs (python classes derived from `dflow.python.OP`s) that will be illustrated later. `train_steps` is an instance of `dflow.Steps`. The outputs of the second OP `run_train` are assigned to the outputs of the `train_steps`.

The `prep_train` prepares a list of paths, each of which contains all necessary files to start a DeePMD-kit training tasks.

The `run_train` slices the list of paths, and assign each item in the list to a DeePMD-kit task. The task is executed by `run_train_op`. This is a very nice feature of `dflow`, because the developer only needs to implement how one DeePMD-kit task is executed, and then all the items in the task list will be executed in parallel. See the following code to see how it works

```
    run_train = Step(
        'run-train',
        template=PythonOPTemplate(
            run_train_op,
            image=run_train_image,
            slices = Slices(
                "int('{{item}}')",
                input_parameter = ["task_name"],
                input_artifact = ["task_path", "init_model"],
                output_artifact = ["model", "lcurve", "log", "script"],
            ),
        ),
        parameters={
            "config" : train_steps.inputs.parameters["train_config"],
            "task_name" : prep_train.outputs.parameters["task_names"],
        },
        artifacts={
            'task_path' : prep_train.outputs.artifacts['task_paths'],
            "init_model" : train_steps.inputs.artifacts['init_models'],
            "init_data": train_steps.inputs.artifacts['init_data'],
            "iter_data": train_steps.inputs.artifacts['iter_data'],
        },
        with_sequence=argo_sequence(argo_len(prep_train.outputs.parameters["task_names
↪"]), format=train_index_pattern),
        key = step_keys['run-train'],
    )
```

The input parameter `"task_names"` and artifacts `"task_paths"` and `"init_model"` are sliced and supplied to each DeePMD-kit task. The output artifacts of the tasks (`"model"`, `"lcurve"`, `"log"` and `"script"`) are stacked in the same order as the input lists. These lists are assigned as the outputs of `train_steps` by

```
    train_steps.outputs.artifacts["scripts"]._from = run_train.outputs.artifacts["script
↪"]
    train_steps.outputs.artifacts["models"]._from = run_train.outputs.artifacts["model"]
    train_steps.outputs.artifacts["logs"]._from = run_train.outputs.artifacts["log"]
    train_steps.outputs.artifacts["lcurves"]._from = run_train.outputs.artifacts["lcurve
↪"]
```

4.2 The OP RunDPTrain

We will take RunDPTrain as an example to illustrate how to implement an OP in DPGEN2. The source code of this OP is found here

Firstly of all, an OP should be implemented as a derived class of dflow.python.OP.

The dflow.python.OP requires static type define for the input and output variables, i.e. the signatures of an OP. The input and output signatures of the dflow.python.OP are given by classmethods get_input_sign and get_output_sign.

```
from dflow.python import (
    OP,
    OPIO,
    OPIOSign,
    Artifact,
)
class RunDPTrain(OP):
    @classmethod
    def get_input_sign(cls):
        return OPIOSign({
            "config" : dict,
            "task_name" : str,
            "task_path" : Artifact(Path),
            "init_model" : Artifact(Path),
            "init_data" : Artifact(List[Path]),
            "iter_data" : Artifact(List[Path]),
        })

    @classmethod
    def get_output_sign(cls):
        return OPIOSign({
            "script" : Artifact(Path),
            "model" : Artifact(Path),
            "lcurve" : Artifact(Path),
            "log" : Artifact(Path),
        })
```

All items not defined as Artifact are treated as parameters of the OP. The concept of parameter and artifact are explained in the dflow document. To be short, the artifacts can be pathlib.Path or a list of pathlib.Path. The artifacts are passed by the file system. Other data structures are treated as parameters, they are passed as variables encoded in str. Therefore, a large amout of information should be stored in artifacts, otherwise they can be considered as parameters.

The operation of the OP is implemented in method execute, and are run in docker containers. Again taking the execute method of RunDPTrain as an example

```
    @OP.exec_sign_check
    def execute(
            self,
            ip : OPIO,
    ) -> OPIO:
        ...
        task_name = ip['task_name']
```

(continues on next page)

(continued from previous page)

```
        task_path = ip['task_path']
        init_model = ip['init_model']
        init_data = ip['init_data']
        iter_data = ip['iter_data']
        ...
        work_dir = Path(task_name)
        ...
        # here copy all files in task_path to work_dir
        ...
        with set_directory(work_dir):
            fplog = open('train.log', 'w')
            def clean_before_quit():
                fplog.close()
            # train model
            command = ['dp', 'train', train_script_name]
            ret, out, err = run_command(command)
            if ret != 0:
                clean_before_quit()
                raise FatalError('dp train failed')
            fplog.write(out)
            # freeze model
            ret, out, err = run_command(['dp', 'freeze', '-o', 'frozen_model.pb'])
            if ret != 0:
                clean_before_quit()
                raise FatalError('dp freeze failed')
            fplog.write(out)
            clean_before_quit()

        return OPIO({
            "script" : work_dir / train_script_name,
            "model" : work_dir / "frozen_model.pb",
            "lcurve" : work_dir / "lcurve.out",
            "log" : work_dir / "train.log",
        })
```

The inputs and outputs variables are recorded in data structure `dflow.python.OPIO`, which is initialized by a Python dict. The keys in the input/output `dict`, and the types of the input/output variables will be checked against their signatures by decorator `OP.exec_sign_check`. If any key or type does not match, an exception will be raised.

It is noted that all input artifacts of the `OP` are read-only, therefore, the first step of the `RunDPTrain.execute` is to copy all necessary input files from the directory `task_path` prepared by `PrepDPTrain` to the working directory `work_dir`.

`with_directory` method creates the `work_dir` and swithes to the directory before the execution, and then exits the directoy when the task finishes or an error is raised.

In what follows, the training and model frozen bash commands are executed consecutively. The return code is check and a `FatalError` is raised if a non-zero code is detected.

Finally the trained model file, input script, learning curve file and the log file are recored in a `dflow.python.OPIO` and returned.

CHAPTER

FIVE

EXPLORATION

DPGEN2 allows developers to contribute exploration strategies. The exploration strategy defines how the configuration space is explored by molecular simulations in each DPGEN iteration. Notice that we are not restricted to molecular dynamics, any molecular simulation is, in priciple, allowed. For example, Monte Carlo, enhanced sampling, structure optimization, and so on.

An exploration strategy takes the history of exploration as input, and gives back DPGEN the exploration tasks (we call it **task group**) and the rule to select configurations from the trajectories generated by the tasks (we call it **configuration selector**).

One can contribute from three aspects:

- The stage scheduler
- The exploration task groups
- Configuration selector

5.1 Stage scheduler

The stage scheduler takes an exploration report passed from the exploration scheduler as input, and tells the exploration scheduler if the exploration in the stage is converged, if not, returns a group of exploration tasks and a configuration selector that are used in the next DPGEN iteration.

Detailed explanation of the concepts are found here.

All the stage schedulers are derived from the abstract base class `StageScheduler`. The only interface to be implemented is `StageScheduler.plan_next_iteration`. One may check the doc string for the explanation of the interface.

```
class StageScheduler(ABC):
    """
    The scheduler for an exploration stage.
    """

    @abstractmethod
    def plan_next_iteration(
            self,
            hist_reports : List[ExplorationReport],
            report : ExplorationReport,
            confs : List[Path],
    ) -> Tuple[bool, ExplorationTaskGroup, ConfSelector] :
        """
```

(continues on next page)

(continued from previous page)

```
        Make the plan for the next iteration of the stage.

        It checks the report of the current and all historical iterations of the stage,
        and tells if the iterations are converged.
        If not converged, it will plan the next ieration for the stage.

        Parameters
        ----------
        hist_reports: List[ExplorationReport]
            The historical exploration report of the stage. If this is the first␣
↪iteration of the stage, this list is empty.
        report : ExplorationReport
            The exploration report of this iteration.
        confs: List[Path]
            A list of configurations generated during the exploration. May be used to␣
↪generate new configurations for the next iteration.

        Returns
        -------
        converged: bool
            If the stage converged.
        task: ExplorationTaskGroup
            A `ExplorationTaskGroup` defining the exploration of the next iteration.␣
↪Should be `None` if the stage is converged.
        conf_selector: ConfSelector
            The configuration selector for the next iteration. Should be `None` if the␣
↪stage is converged.

        """
```

One may check more details on the exploratin task group and the configuration selector.

5.2 Exploration task groups

DPGEN2 defines a python class `ExplorationTask` to manage all necessry files needed to run a exploration task. It can be used as the example provided in the doc string.

```
class ExplorationTask():
    """Define the files needed by an exploration task.

    Examples
    --------
    >>> # this example dumps all files needed by the task.
    >>> files = exploration_task.files()
    ... for file_name, file_content in files.items():
    ...     with open(file_name, 'w') as fp:
    ...         fp.write(file_content)

    """
```

A collection of the exploration tasks is called exploration task group. All tasks groups are derived from the base class

ExplorationTaskGroup. The exploration task group can be viewd as a list of ExplorationTasks, one may get the list by using property ExplorationTaskGroup.task_list. One may add tasks, or ExplorationTaskGroup to the group by methods ExplorationTaskGroup.add_task and ExplorationTaskGroup.add_group, respectively.

```
class ExplorationTaskGroup(Sequence):
    @property
    def task_list(self) -> List[ExplorationTask]:
        """Get the `list` of `ExplorationTask`"""
        ...

    def add_task(self, task: ExplorationTask):
        """Add one task to the group."""
        ...

    def add_group(
            self,
            group : 'ExplorationTaskGroup',
    ):
        """Add another group to the group."""
        ...
```

An example of generating a group of NPT MD simulations may illustrate how to implement the ExplorationTaskGroups.

5.3 Configuration selector

The configuration selectors are derived from the abstract base class ConfSelector

```
class ConfSelector(ABC):
    """Select configurations from trajectory and model deviation files.
    """
    @abstractmethod
    def select (
            self,
            trajs : List[Path],
            model_devis : List[Path],
            traj_fmt : str = 'deepmd/npy',
            type_map : List[str] = None,
    ) -> Tuple[List[ Path ], ExplorationReport]:
```

The abstractmethod to implement is ConfSelector.select. trajs and model_devis are lists of files that recording the simulations trajectories and model deviations respectively. traj_fmt and type_map are parameters that may be needed for loading the trajectories by dpdata.

The ConfSelector.select returns a Path, each of which can be treated as a dpdata.MultiSystems, and a ExplorationReport.

An example of selecting configurations from LAMMPS trajectories may illustrate how to implement the ConfSelectors.

CHAPTER

SIX

DPGEN2 API

6.1 dpgen2 package

6.1.1 Subpackages

dpgen2.entrypoint package

Submodules

dpgen2.entrypoint.main module

dpgen2.entrypoint.main.**main**()

dpgen2.entrypoint.main.**main_parser**() → ArgumentParser

DPGEN2 commandline options argument parser.

Returns

argparse.ArgumentParser
the argument parser

Notes

This function is used by documentation.

dpgen2.entrypoint.main.**parse_args**(*args: Optional[List[str]] = None*)

DPGEN2 commandline options argument parsing.

Parameters

args: List[str]
list of command line arguments, main purpose is testing default option None takes arguments from sys.argv

dpgen2.entrypoint.status module

dpgen2.entrypoint.status.**status**(*workflow_id*, *wf_config: Optional[Dict] = {}*)

dpgen2.entrypoint.submit module

dpgen2.entrypoint.submit.**expand_idx**(*in_list*)

dpgen2.entrypoint.submit.**expand_sys_str**(*root_dir: Union[str, Path]*) → List[str]

dpgen2.entrypoint.submit.**get_kspacing_kgamma_from_incar**(*fname*)

dpgen2.entrypoint.submit.**make_concurrent_learning_op**(*train_style: str = 'dp'*, *explore_style: str = 'lmp'*, *fp_style: str = 'vasp'*, *prep_train_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *run_train_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *prep_explore_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *run_explore_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *prep_fp_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *run_fp_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *select_confs_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *collect_data_config: str = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest',*

dpgen2.entrypoint.submit.**make_conf_list**(*conf_list*, *type_map*, *fmt='vasp/poscar'*)

dpgen2.entrypoint.submit.**make_naive_exploration_scheduler**(*config*)

dpgen2.entrypoint.submit.**print_list_steps**(*steps*)

dpgen2.entrypoint.submit.**resubmit_concurrent_learning**(*wf_config*, *wfid*, *list_steps=False*, *reuse=None*)

dpgen2.entrypoint.submit.**submit_concurrent_learning**(*wf_config*, *reuse_step=None*)

dpgen2.entrypoint.submit.**successful_step_keys**(*wf*)

dpgen2.entrypoint.submit.**wf_global_workflow**(*wf_config*)

dpgen2.entrypoint.submit.**workflow_concurrent_learning**(*config*)

dpgen2.exploration package

Subpackages

dpgen2.exploration.report package

Submodules

dpgen2.exploration.report.naive_report module

class dpgen2.exploration.report.naive_report.**NaiveExplorationReport**(*counter_f*, *counter_v*)

Bases: *ExplorationReport*

Methods

accurate_ratio	
calculate_ratio	
candidate_ratio	
failed_ratio	
ratio	

accurate_ratio(*tag=None*) → float

static calculate_ratio(*cc*, *ca*, *cf*)

candidate_ratio(*tag=None*) → float

failed_ratio(*tag=None*) → float

ratio(*quantity: str*, *item: str*) → float

dpgen2.exploration.report.report module

class dpgen2.exploration.report.report.**ExplorationReport**

Bases: ABC

Methods

accurate_ratio	
candidate_ratio	
failed_ratio	

abstract accurate_ratio(*tag=None*) → float

abstract candidate_ratio(*tag=None*) → float

abstract failed_ratio(*tag=None*) → float

dpgen2.exploration.report.trajs_report module

class dpgen2.exploration.report.trajs_report.**TrajsExplorationReport**

Bases: *ExplorationReport*

Methods

get_candidates([max_nframes])	Get candidates.
record_traj(id_f_accu, id_f_cand, id_f_fail, ...)	Record one trajctory.

accurate_ratio	
candidate_ratio	
clear	
failed_ratio	

accurate_ratio(*tag=None*)

candidate_ratio(*tag=None*)

clear()

failed_ratio(*tag=None*)

get_candidates(*max_nframes: Optional[int] = None*) → List[Tuple[int, int]]

Get candidates. If number of candidates is larger than *max_nframes*, then randomly pick *max_nframes* frames from the candidates.

Parameters

max_nframes int
The maximal number of frames of candidates.

Returns

cand_frames List[Tuple[int,int]]
Candidate frames. A list of tuples: [(traj_idx, frame_idx), ...]

record_traj(*id_f_accu*, *id_f_cand*, *id_f_fail*, *id_v_accu*, *id_v_cand*, *id_v_fail*)
Record one trajctory. inputs are the indexes of candidate, accurate and failed frames.

dpgen2.exploration.scheduler package

Submodules

dpgen2.exploration.scheduler.convergence_check_stage_scheduler module

class dpgen2.exploration.scheduler.convergence_check_stage_scheduler.**ConvergenceCheckStageScheduler**(*stage*, *exploration_selector: ConfSelector*, *conv_tolerance: float = 0.9*, *max_numb_iter: Optional = None*, *fatal_at_max: bool = True*)

Bases: *StageScheduler*

Methods

converged()	Tell if the stage is converged
plan_next_iteration([report, trajs])	Make the plan for the next iteration of the stage.

complete	
reached_max_iteration	

`complete()`

`converged()`

Tell if the stage is converged

Returns

converged bool
the convergence

`plan_next_iteration`(*report: Optional[*ExplorationReport*] = None*, *trajs: Optional[List[Path]] = None*) → Tuple[bool, *ExplorationTaskGroup*, *ConfSelector*]

Make the plan for the next iteration of the stage.

It checks the report of the current and all historical iterations of the stage, and tells if the iterations are converged. If not converged, it will plan the next ieration for the stage.

Parameters

hist_reports: List[ExplorationReport]
The historical exploration report of the stage. If this is the first iteration of the stage, this list is empty.

report
[ExplorationReport] The exploration report of this iteration.

confs: List[Path]
A list of configurations generated during the exploration. May be used to generate new configurations for the next iteration.

Returns

stg_complete: bool
If the stage completed. Two cases may happen: 1. converged. 2. when not fatal_at_max, not converged but reached max number of iterations.

task: ExplorationTaskGroup
A *ExplorationTaskGroup* defining the exploration of the next iteration. Should be *None* if the stage is converged.

conf_selector: ConfSelector
The configuration selector for the next iteration. Should be *None* if the stage is converged.

`reached_max_iteration()`

dpgen2.exploration.scheduler.scheduler module

`class dpgen2.exploration.scheduler.scheduler.ExplorationScheduler`

Bases: `object`

The exploration scheduler.

Methods

add_stage_scheduler(stage_scheduler)	Add stage scheduler.
complete()	Tell if all stages are converged.
get_convergence_ratio()	Get the accurate, candidate and failed ratios of the iterations
get_iteration()	Get the index of the current iteration.
get_stage()	Get the index of current stage.
get_stage_of_iterations()	Get the stage index and the index in the stage of iterations.
plan_next_iteration([report, trajs])	Make the plan for the next DPGEN iteration.

print_convergence	

add_stage_scheduler(*stage_scheduler:* StageScheduler)

Add stage scheduler.

All added schedulers can be treated as a *list* (order matters). Only one stage is converged, the iteration goes to the next iteration.

Parameters

stage_scheduler: StageScheduler
The added stage scheduler

complete()

Tell if all stages are converged.

get_convergence_ratio()

Get the accurate, candidate and failed ratios of the iterations

Returns

accu np.ndarray
The accurate ratio. length of array the same as # iterations.

cand np.ndarray
The candidate ratio. length of array the same as # iterations.

fail np.ndarray
The failed ration. length of array the same as # iterations.

get_iteration()

Get the index of the current iteration.

Iteration index increase when *self.plan_next_iteration* returns valid *lmp_task_grp* and *conf_selector* for the next iteration.

get_stage()

Get the index of current stage.

Stage index increases when the previous stage converges. Usually called after *self.plan_next_iteration*.

get_stage_of_iterations()

Get the stage index and the index in the stage of iterations.

plan_next_iteration(*report: Optional[*ExplorationReport*] = None*, *trajs: Optional[List[Path]] = None*) → Tuple[bool, *ExplorationTaskGroup*, *ConfSelector*]

Make the plan for the next DPGEN iteration.

Parameters

report
[ExplorationReport] The exploration report of this iteration.

confs: List[Path]
A list of configurations generated during the exploration. May be used to generate new configurations for the next iteration.

Returns

complete: bool
If all the DPGEN stages complete.

task: ExplorationTaskGroup
A *ExplorationTaskGroup* defining the exploration of the next iteration. Should be *None* if converged.

conf_selector: ConfSelector
The configuration selector for the next iteration. Should be *None* if converged.

print_convergence()

dpgen2.exploration.scheduler.stage_scheduler module

class dpgen2.exploration.scheduler.stage_scheduler.**StageScheduler**

Bases: ABC

The scheduler for an exploration stage.

Methods

converged()	Tell if the stage is converged
plan_next_iteration(report, trajs)	Make the plan for the next iteration of the stage.

abstract converged()

Tell if the stage is converged

Returns

converged bool
the convergence

abstract plan_next_iteration(*report:* ExplorationReport, *trajs: List[Path]*) → Tuple[bool, *ExplorationTaskGroup*, *ConfSelector*]

Make the plan for the next iteration of the stage.

It checks the report of the current and all historical iterations of the stage, and tells if the iterations are converged. If not converged, it will plan the next ieration for the stage.

Parameters

hist_reports: List[ExplorationReport]
The historical exploration report of the stage. If this is the first iteration of the stage, this list is empty.

report
[ExplorationReport] The exploration report of this iteration.

confs: List[Path]
A list of configurations generated during the exploration. May be used to generate new configurations for the next iteration.

Returns

stg_complete: bool
If the stage completed. Two cases may happen: 1. converged. 2. when not fatal_at_max, not converged but reached max number of iterations.

task: ExplorationTaskGroup
A *ExplorationTaskGroup* defining the exploration of the next iteration. Should be *None* if the stage is converged.

conf_selector: ConfSelector
The configuration selector for the next iteration. Should be *None* if the stage is converged.

dpgen2.exploration.selector package

Submodules

dpgen2.exploration.selector.conf_filter module

`class dpgen2.exploration.selector.conf_filter.ConfFilter`

Bases: ABC

Methods

check(coords, cell, atom_types, nopbc)	Check if the configuration is valid.

`abstract check(coords: array, cell: array, atom_types: array, nopbc: bool) → bool`

Check if the configuration is valid.

Parameters

coords
[numpy.array] The coordinates, numpy array of shape natoms x 3

cell
[numpy.array] The cell tensor. numpy array of shape 3 x 3

atom_types
[numpy.array] The atom types. numpy array of shape natoms

nopbc
[bool] If no periodic boundary condition.

Returns

valid
[bool] *True* if the configuration is a valid configuration, else *False*.

class dpgen2.exploration.selector.conf_filter.**ConfFilters**

Bases: object

Methods

add	
check	

add(*conf_filter:* ConfFilter) → *ConfFilters*

check(*conf:* *System*) → bool

dpgen2.exploration.selector.conf_selector module

class dpgen2.exploration.selector.conf_selector.**ConfSelector**

Bases: ABC

Select configurations from trajectory and model deviation files.

Methods

select	

abstract select(*trajs: List[Path]*, *model_devis: List[Path]*, *traj_fmt: str = 'deepmd/npy'*, *type_map: Optional[List[str]] = None*) → Tuple[List[Path], *ExplorationReport*]

dpgen2.exploration.selector.conf_selector_frame module

class dpgen2.exploration.selector.conf_selector_frame.**ConfSelectorLammpsFrames**(*trust_level*, *max_numb_sel: Optional[int] = None*, *conf_filters: Optional[*ConfFilters*] = None*)

Bases: *ConfSelector*

Select frames from trajectories as confs.

Parameters: trust_level: TrustLevel

The trust level

conf_filter: ConfFilters
The configuration filter

Methods

select(trajs, model_devis[, traj_fmt, type_map])	Select configurations

record_one_traj	

record_one_traj(*traj*, *model_devi*, *traj_fmt*, *type_map*) → None

select(*trajs: List[Path]*, *model_devis: List[Path]*, *traj_fmt: str = 'lammps/dump'*, *type_map: Optional[List[str]] = None*) → Tuple[List[Path], *ExplorationReport*]

Select configurations

Parameters

trajs
[List[Path]] A *list* of *Path* to trajectory files generated by LAMMPS

model_devis
[List[Path]] A *list* of *Path* to model deviation files generated by LAMMPS. Format: each line has 7 numbers they are used as # frame_id md_v_max md_v_min md_v_mean md_f_max md_f_min md_f_mean where *md* stands for model deviation, v for virial and f for force

traj_fmt
[str] Format of the trajectory, by default it is the dump file of LAMMPS

type_map
[List[str]] The *type_map* of the systems

Returns

confs
[List[Path]] The selected confgurations, stored in a folder in deepmd/npy format, can be parsed as dpdata.MultiSystems. The *list* only has one item.

report
[ExplorationReport] The exploration report recoding the status of the exploration.

dpgen2.exploration.selector.trust_level module

class dpgen2.exploration.selector.trust_level.**TrustLevel**(*level_f_lo*, *level_f_hi*, *level_v_lo=None*, *level_v_hi=None*)

Bases: object

Attributes

level_f_hi
level_f_lo
level_v_hi
level_v_lo

property level_f_hi

property level_f_lo

property level_v_hi

property level_v_lo

dpgen2.exploration.task package

Subpackages

dpgen2.exploration.task.lmp package

Submodules

dpgen2.exploration.task.lmp.lmp_input module

dpgen2.exploration.task.lmp.lmp_input.**make_lmp_input**(*conf_file: str*, *ensemble: str*, *graphs: List[str]*, *nsteps: int*, *dt: float*, *neidelay: int*, *trj_freq: int*, *mass_map: List[float]*, *temp: float*, *tau_t: float = 0.1*, *pres: Optional[float] = None*, *tau_p: float = 0.5*, *use_clusters: bool = False*, *relative_f_epsilon: Optional[float] = None*, *relative_v_epsilon: Optional[float] = None*, *pka_e: Optional[float] = None*, *ele_temp_f: Optional[float] = None*, *ele_temp_a: Optional[float] = None*, *nopbc: bool = False*, *max_seed: int = 1000000*, *deepmd_version='2.0'*, *trj_seperate_files=True*)

Submodules

dpgen2.exploration.task.npt_task_group module

class dpgen2.exploration.task.npt_task_group.**NPTTaskGroup**

Bases: *ExplorationTaskGroup*

Attributes

task_list
Get the *list* of *ExplorationTask*

Methods

add_group(group)	Add another group to the group.
add_task(task)	Add one task to the group.
count(value)	
index(value, [start, [stop]])	Raises ValueError if the value is not present.
make_task()	Make the LAMMPS task group.
set_conf(conf_list[, n_sample, random_sample])	Set the configurations of exploration
set_md(numb_models, mass_map, temps[, ...])	Set MD parameters

clear

make_task() → *ExplorationTaskGroup*

Make the LAMMPS task group.

Returns

task_grp: ExplorationTaskGroup
The returned lammps task group. The number of tasks is nconf*nT*nP. nconf is set by *n_sample* parameter of *set_conf*. nT and nP are lengths of the *temps* and *press* parameters of *set_md*.

set_conf(*conf_list: List[str]*, *n_sample: Optional[int] = None*, *random_sample: bool = False*)

Set the configurations of exploration

Parameters

conf_list str
A list of file contents

n_sample int
Number of samples drawn from the conf list each time *make_task* is called. If set to *None*, *n_sample* is set to length of the conf_list.

random_sample bool
If true the confs are randomly sampled, otherwise are consecutively sampled from the conf_list

set_md(*numb_models*, *mass_map*, *temps: List[float]*, *press: Optional[List[float]] = None*, *ens: str = 'npt'*, *dt: float = 0.001*, *nsteps: int = 1000*, *trj_freq: int = 10*, *tau_t: float = 0.1*, *tau_p: float = 0.5*, *pka_e: Optional[float] = None*, *neidelay: Optional[int] = None*, *no_pbc: bool = False*, *use_clusters: bool = False*, *relative_f_epsilon: Optional[float] = None*, *relative_v_epsilon: Optional[float] = None*, *ele_temp_f: Optional[float] = None*, *ele_temp_a: Optional[float] = None*)

Set MD parameters

dpgen2.exploration.task.stage module

class dpgen2.exploration.task.stage.**ExplorationStage**

Bases: object

The exploration stage.

Methods

add_task_group(grp)	Add an exploration group
clear()	Clear all exploration group.
make_task()	Make the LAMMPS task group.

add_task_group(*grp:* ExplorationTaskGroup)

Add an exploration group

Parameters

grp: ExplorationTaskGroup
The added exploration task group

clear()

Clear all exploration group.

make_task() → *ExplorationTaskGroup*

Make the LAMMPS task group.

Returns

task_grp: ExplorationTaskGroup
The returned lammps task group. The number of tasks is equal to the summation of task groups defined by all the exploration groups added to the stage.

dpgen2.exploration.task.task module

class dpgen2.exploration.task.task.**ExplorationTask**

Bases: object

Define the files needed by an exploration task.

Examples

```
>>> # this example dumps all files needed by the task.
>>> files = exploration_task.files()
... for file_name, file_content in files.items():
...     with open(file_name, 'w') as fp:
...         fp.write(file_content)
```

Methods

add_file(fname, fcont)	Add file to the task
files()	Get all files for the task.

add_file(*fname: str*, *fcont: str*)

Add file to the task

Parameters

fname
[str] The name of the file

fcont
[str] The content of the file.

files() → Dict

Get all files for the task.

Returns

files
[dict] The dict storing all files for the task. The file name is a key of the dict, and the file content is the corresponding value.

class dpgen2.exploration.task.task.**ExplorationTaskGroup**

Bases: Sequence

A group of exploration tasks. Implemented as a *list* of *ExplorationTask*.

Attributes

task_list
Get the *list* of *ExplorationTask*

Methods

add_group(group)	Add another group to the group.
add_task(task)	Add one task to the group.
count(value)	
index(value, [start, [stop]])	Raises ValueError if the value is not present.

clear

add_group(*group:* ExplorationTaskGroup)

Add another group to the group.

add_task(*task:* ExplorationTask)

Add one task to the group.

clear() → None

property task_list: List[*ExplorationTask*]

Get the *list* of *ExplorationTask*

class dpgen2.exploration.task.task.**FooTask**(*conf_name='conf.lmp'*, *conf_cont=''*, *inpu_name='in.lammps'*, *inpu_cont=''*)

Bases: *ExplorationTask*

Methods

add_file(fname, fcont)	Add file to the task
files()	Get all files for the task.

class dpgen2.exploration.task.task.**FooTaskGroup**(*numb_task*)

Bases: *ExplorationTaskGroup*

Attributes

task_list
Get the *list* of *ExplorationTask*

Methods

add_group(group)	Add another group to the group.
add_task(task)	Add one task to the group.
count(value)	
index(value, [start, [stop]])	Raises ValueError if the value is not present.

clear	

property task_list

Get the *list* of *ExplorationTask*

dpgen2.flow package

Submodules

dpgen2.flow.dpgen_loop module

class dpgen2.flow.dpgen_loop.**ConcurrentLearning**(*name: str*, *block_op: Steps*, *step_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *upload_python_package: Optional[str] = None*)

Bases: Steps

Attributes

init_keys
input_artifacts
input_parameters
loop_keys
output_artifacts
output_parameters

Methods

add(step)	Add a step or a list of parallel steps to the steps

convert_to_argo	
handle_key	
run	

property init_keys

property input_artifacts

property input_parameters

property loop_keys

property output_artifacts

property output_parameters

class dpgen2.flow.dpgen_loop.**ConcurrentLearningLoop**(*name: str*, *block_op: Steps*, *step_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *upload_python_package: Optional[str] = None*)

Bases: Steps

Attributes

input_artifacts
input_parameters
keys
output_artifacts
output_parameters

Methods

add(step)	Add a step or a list of parallel steps to the steps

convert_to_argo	
handle_key	
run	

property input_artifacts

property input_parameters

property keys

property output_artifacts

property output_parameters

class dpgen2.flow.dpgen_loop.**MakeBlockId**(**args*, ***kwargs*)

Bases: OP

Methods

execute(ip)	Run the OP
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	

execute(*ip: OPIO*) → OPIO

Run the OP

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

class dpgen2.flow.dpgen_loop.**SchedulerWrapper**(**args*, ***kwargs*)

Bases: OP

Methods

execute(ip)	Run the OP
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	

execute(*ip: OPIO*) → OPIO

Run the OP

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

dpgen2.fp package

Submodules

dpgen2.fp.vasp module

class dpgen2.fp.vasp.**VaspInputs**(*kspacing: Union[float, List[float]]*, *kgamma: bool = True*, *incar_template_name: Optional[str] = None*, *potcar_names: Optional[Dict[str, str]] = None*)

Bases: object

Attributes

incar_template
potcars

Methods

incar_from_file	
make_kpoints	
make_potcar	
potcars_from_file	

incar_from_file(*fname: str*)

property incar_template

make_kpoints(*box: array*) → str

make_potcar(*atom_names*) → str

property potcars

potcars_from_file(*dict_fnames: Dict[str, str]*)

dpgen2.fp.vasp.**make_kspacing_kpoints**(*box*, *kspacing*, *kgamma*)

dpgen2.op package

Submodules

dpgen2.op.collect_data module

class dpgen2.op.collect_data.**CollectData**(**args*, ***kwargs*)

Bases: OP

Collect labeled data and add to the iteration dataset.

After running FP tasks, the labeled data are scattered in task directories. This OP collect the labeled data in one data directory and add it to the iteration data. The data generated by this iteration will be place in *ip["name"]* subdirectory of the iteration data directory.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	

execute(*ip: OPIO*) → OPIO

Execute the OP. This OP collect data scattered in directories given by *ip['labeled_data']* in to one *dpdata.Multisystems* and store it in a directory named *name*. This directory is appended to the list *iter_data*.

Parameters

ip

[dict] Input dict with components:

- *name*: (*str*) The name of this iteration. The data generated by this iteration will be place in a sub-directory of *name*.
- *labeled_data*: (*Artifact(List[Path])*) The paths of labeled data generated by FP tasks of the current iteration.
- *iter_data*: (*Artifact(List[Path])*) The data paths previous iterations.

Returns

Output dict with components:

- ***iter_data*: (*Artifact(List[Path])*) The data paths of previous and the current iteration data.**

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

dpgen2.op.md_settings module

class dpgen2.op.md_settings.**MDSettings**(*ens: str*, *dt: float*, *nsteps: int*, *trj_freq: int*, *temps: Optional[List[float]] = None*, *press: Optional[List[float]] = None*, *tau_t: float = 0.1*, *tau_p: float = 0.5*, *pka_e: Optional[float] = None*, *neidelay: Optional[int] = None*, *no_pbc: bool = False*, *use_clusters: bool = False*, *relative_epsilon: Optional[float] = None*, *relative_v_epsilon: Optional[float] = None*, *ele_temp_f: Optional[float] = None*, *ele_temp_a: Optional[float] = None*)

Bases: object

Methods

to_str	

to_str() → str

dpgen2.op.prep_dp_train module

class dpgen2.op.prep_dp_train.PrepDPTrain(*args, **kwargs)

Bases: OP

Prepares the working directories for DP training tasks.

A list of (*numb_models*) working directories containing all files needed to start training tasks will be created. The paths of the directories will be returned as *op["task_paths"]*. The identities of the tasks are returned as *op["task_names"]*.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	

execute(*ip: OPIO*) → OPIO

Execute the OP.

Parameters

ip

[dict] Input dict with components:

- *template_script*: (*str* or *List[str]*) A template of the training script. Can be a *str* or *List[str]*. In the case of *str*, all training tasks share the same training input template, the only difference is the random number used to initialize the network parameters. In the case of *List[str]*, one training task uses one template from the list. The random numbers used to initialize the network parameters are differnt. The length of the list should be the same as *numb_models*.
- *numb_models*: (*int*) Number of DP models to train.

Returns

op

[dict] Output dict with components:

- *task_names*: (*List[str]*) The name of tasks. Will be used as the identities of the tasks. The names of different tasks are different.
- *task_paths*: (*Artifact(List[Path])*) The parepared working paths of the tasks. The order fo the Paths should be consistent with *op["task_names"]*

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

dpgen2.op.prep_lmp module

dpgen2.op.prep_lmp.**PrepExplorationTaskGroup**

alias of *PrepLmp*

class dpgen2.op.prep_lmp.**PrepLmp**(**args*, ***kwargs*)

Bases: OP

Prepare the working directories for LAMMPS tasks.

A list of working directories (defined by *ip["task"]*) containing all files needed to start LAMMPS tasks will be created. The paths of the directories will be returned as *op["task_paths"]*. The identities of the tasks are returned as *op["task_names"]*.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	

execute(*ip: OPIO*) → OPIO

Execute the OP.

Parameters

ip

[dict] Input dict with components: - *lmp_task_grp* : (*Artifact(Path)*) Can be pickle loaded as a ExplorationTaskGroup. Definitions for LAMMPS tasks

Returns

op

[dict] Output dict with components:

- *task_names*: (*List[str]*) The name of tasks. Will be used as the identities of the tasks. The names of different tasks are different.

- *task_paths*: (*Artifact(List[Path])*) The parepared working paths of the tasks. Contains all input files needed to start the LAMMPS simulation. The order fo the Paths should be consistent with *op["task_names"]*

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

dpgen2.op.prep_vasp module

class dpgen2.op.prep_vasp.**PrepVasp**(**args*, ***kwargs*)

Bases: OP

Prepares the working directories for VASP tasks.

A list of (same length as ip["confs"]) working directories containing all files needed to start VASP tasks will be created. The paths of the directories will be returned as *op["task_paths"]*. The identities of the tasks are returned as *op["task_names"]*.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	

execute(*ip: OPIO*) → OPIO

Execute the OP.

Parameters

ip

[dict] Input dict with components:

- *inputs* : (*VaspInputs*) Definitions for the VASP inputs
- *confs* : (*Artifact(List[Path])*) Configurations for the VASP tasks. Stored in folders as deepmd/npy format. Can be parsed as dpdata.MultiSystems.

Returns

op

[dict] Output dict with components:

- *task_names*: (*List[str]*) The name of tasks. Will be used as the identities of the tasks. The names of different tasks are different.

- *task_paths*: (*Artifact(List[Path])*) The parepared working paths of the tasks. Contains all input files needed to start the VASP. The order fo the Paths should be consistent with *op["task_names"]*

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

dpgen2.op.run_dp_train module

class dpgen2.op.run_dp_train.**RunDPTrain**(**args*, ***kwargs*)

Bases: OP

Execute a DP training task. Train and freeze a DP model.

A working directory named *task_name* is created. All input files are copied or symbol linked to directory *task_name*. The DeePMD-kit training and freezing commands are exectuted from directory *task_name*.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

decide_init_model	
exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	
normalize_config	
training_args	
write_data_to_input_script	
write_other_to_input_script	

static decide_init_model(*config*, *init_model*, *init_data*, *iter_data*)

execute(*ip: OPIO*) → OPIO

Execute the OP.

Parameters

ip

[dict] Input dict with components:

- *config*: (*dict*) The config of training task. Check *RunDPTrain.training_args* for definitions.
- *task_name*: (*str*) The name of training task.

- *task_path*: (*Artifact(Path)*) The path that contains all input files prepareed by *PrepDP-Train*.
- *init_model*: (*Artifact(Path)*) A frozen model to initialize the training.
- *init_data*: (*Artifact(List[Path])*) Initial training data.
- *iter_data*: (*Artifact(List[Path])*) Training data generated in the DPGEN iterations.

Returns

Output dict with components:

- ***script*: (*Artifact(Path)*) The training script.**
- ***model*: (*Artifact(Path)*) The trained frozen model.**
- ***lcurve*: (*Artifact(Path)*) The learning curve file.**
- ***log*: (*Artifact(Path)*) The log file of training.**

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

static normalize_config(*data={}*)

static training_args()

static write_data_to_input_script(*idict: dict*, *init_data: List[Path]*, *iter_data: List[Path]*, *auto_prob_str: str = 'prob_sys_size'*, *major_version: str = '1'*)

static write_other_to_input_script(*idict*, *config*, *do_init_model*, *major_version: str = '1'*)

dpgen2.op.run_dp_train.**config_args**()

dpgen2.op.run_lmp module

class dpgen2.op.run_lmp.**RunLmp**(**args*, ***kwargs*)

Bases: OP

Execute a LAMMPS task.

A working directory named *task_name* is created. All input files are copied or symbol linked to directory *task_name*. The LAMMPS command is exectuted from directory *task_name*. The trajectory and the model deviation will be stored in files *op["traj"]* and *op["model_devi"]*, respectively.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	
lmp_args	
normalize_config	

execute(*ip: OPIO*) → OPIO

Execute the OP.

Parameters

ip

[dict] Input dict with components:

- *config*: (*dict*) The config of lmp task. Check *RunLmp.lmp_args* for definitions.
- *task_name*: (*str*) The name of the task.
- *task_path*: (*Artifact(Path)*) The path that contains all input files prepareed by *PrepLmp*.
- *models*: (*Artifact(List[Path])*) The frozen model to estimate the model deviation. The first model with be used to drive molecular dynamics simulation.

Returns

Output dict with components:

- ***log*: (*Artifact(Path)*) The log file of LAMMPS.**
- ***traj*: (*Artifact(Path)*) The output trajectory.**
- ***model_devi*: (*Artifact(Path)*) The model deviation. The order of recorded model deviations should be consistent with the order of frames in *traj*.**

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

static lmp_args()

static normalize_config(*data={}*)

dpgen2.op.run_lmp.**config_args**()

dpgen2.op.run_vasp module

class dpgen2.op.run_vasp.**RunVasp**(**args*, ***kwargs*)

Bases: OP

Execute a VASP task.

A working directory named *task_name* is created. All input files are copied or symbol linked to directory *task_name*. The VASP command is exectuted from directory *task_name*. The *op["labeled_data"]* in *"deepmd/npy"* format (HF5 in the future) provided by *dpdata* will be created.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check	
function	
get_input_artifact_link	
get_input_artifact_storage_key	
get_output_artifact_link	
get_output_artifact_storage_key	
normalize_config	
vasp_args	

execute(*ip: OPIO*) → OPIO

Execute the OP.

Parameters

ip

[dict] Input dict with components:

- *config*: (*dict*) The config of vasp task. Check *RunVasp.vasp_args* for definitions.
- *task_name*: (*str*) The name of task.
- *task_path*: (*Artifact(Path)*) The path that contains all input files prepareed by *PrepVasp*.

Returns

Output dict with components:

- ***log*: (*Artifact(Path)*) The log file of VASP.**
- ***labeled_data*: (*Artifact(Path)*) The path to the labeled data in *"deepmd/npy"* format provided by *dpdata*.**

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

static normalize_config(*data={}*)

static vasp_args()

dpgen2.op.run_vasp.**config_args**()

dpgen2.op.select_confs module

class dpgen2.op.select_confs.**SelectConfs**(**args*, ***kwargs*)

Bases: OP

Select configurations from exploration trajectories for labeling.

Methods

execute(ip)	Execute the OP.
get_input_sign()	Get the signature of the inputs
get_output_sign()	Get the signature of the outputs

exec_sign_check
function
get_input_artifact_link
get_input_artifact_storage_key
get_output_artifact_link
get_output_artifact_storage_key

execute(*ip: OPIO*) → OPIO

Execute the OP.

Parameters

ip

[dict] Input dict with components:

- *conf_selector*: (*ConfSelector*) Configuration selector.
- *traj_fmt*: (*str*) The format of trajectory.
- *type_map*: (*List[str]*) The type map.
- *trajs*: (*Artifact(List[Path])*) The trajectories generated in the exploration.
- *model_devis*: (*Artifact(List[Path])*) The file storing the model deviation of the trajectory. The order of model deviation storage is consistent with that of the trajectories. The order of frames of one model deviation storage is also consistent with tat of the corresponding trajectory.

Returns

Output dict with components:

- ***report*: (*ExplorationReport*) The report on the exploration.**
- ***conf*: (*Artifact(List[Path])*) The selected configurations.**

classmethod get_input_sign()

Get the signature of the inputs

classmethod get_output_sign()

Get the signature of the outputs

dpgen2.superop package

Submodules

dpgen2.superop.block module

class dpgen2.superop.block.**ConcurrentLearningBlock**(*name: str*, *prep_run_dp_train_op: OP*, *prep_run_lmp_op: OP*, *select_confs_op: OP*, *prep_run_fp_op: OP*, *collect_data_op: OP*, *select_confs_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *collect_data_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *upload_python_package: Optional[str] = None*)

Bases: Steps

Attributes

input_artifacts
input_parameters
keys
output_artifacts
output_parameters

Methods

add(step)	Add a step or a list of parallel steps to the steps

convert_to_argo	
handle_key	
run	

property input_artifacts

property input_parameters

property keys

property output_artifacts

property output_parameters

dpgen2.superop.prep_run_dp_train module

class dpgen2.superop.prep_run_dp_train.**PrepRunDPTrain**(*name: str*, *prep_train_op: OP*, *run_train_op: OP*, *prep_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *run_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *upload_python_package: Optional[str] = None*)

Bases: Steps

Attributes

input_artifacts
input_parameters
keys
output_artifacts
output_parameters

Methods

add(step)	Add a step or a list of parallel steps to the steps

convert_to_argo	
handle_key	
run	

property input_artifacts

property input_parameters

property keys

property output_artifacts

property output_parameters

dpgen2.superop.prep_run_fp module

class dpgen2.superop.prep_run_fp.**PrepRunFp**(*name: str*, *prep_op: OP*, *run_op: OP*, *prep_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *run_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *upload_python_package: Optional[str] = None*)

Bases: Steps

Attributes

input_artifacts
input_parameters
keys
output_artifacts
output_parameters

Methods

add(step)	Add a step or a list of parallel steps to the steps

convert_to_argo	
handle_key	
run	

property input_artifacts

property input_parameters

property keys

property output_artifacts

property output_parameters

dpgen2.superop.prep_run_lmp module

class dpgen2.superop.prep_run_lmp.**PrepRunLmp**(*name: str*, *prep_op: OP*, *run_op: OP*, *prep_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *run_config: dict = {'continue_on_failed': False, 'continue_on_num_success': None, 'continue_on_success_ratio': None, 'executor': None, 'template_config': {'envs': None, 'image': 'dptechnology/dpgen2:latest', 'retry_on_transient_error': None, 'timeout': None, 'timeout_as_transient_error': False}}*, *upload_python_package: Optional[str] = None*)

Bases: Steps

Attributes

input_artifacts
input_parameters
keys
output_artifacts
output_parameters

Methods

add(step)	Add a step or a list of parallel steps to the steps

convert_to_argo	
handle_key	
run	

property input_artifacts

property input_parameters

property keys

property output_artifacts

property output_parameters

dpgen2.utils package

Submodules

dpgen2.utils.alloy_conf module

class dpgen2.utils.alloy_conf.**AlloyConf**(*lattice: Union[System, Tuple[str, float]]*, *type_map: List[str]*, *replicate: Optional[Union[List[int], Tuple[int], int]] = None*)

Bases: object

Parameters

lattice Union[dpdata.System, Tuple[str,float]]
Lattice of the alloy confs. can be *dpdata.System*: lattice in *dpdata.System Tuple[str, float]*: pair of lattice type and lattice constant. lattice type can be "bcc", "fcc", "hcp", "sc" or "diamond"

replicate Union[List[int], Tuple[int], int]
replicate of the lattice

type_map List[str]
The type map

Methods

generate_file_content(numb_confs[, ...])	**Parameters**
generate_systems(numb_confs[, ...])	**Parameters**

generate_file_content(*numb_confs*, *concentration: Optional[Union[List[List[float]], List[float]]] = None*, *cell_pert_frac: float = 0.0*, *atom_pert_dist: float = 0.0*, *fmt: str = 'lammps/lmp'*) → List[str]

Parameters

numb_confs int
Number of configurations to generate

concentration List[List[float]] or List[float] or None
If *List[float]*, the concentrations of each element. The length of the list should be the same as the *type_map*. If *List[List[float]]*, a list of concentrations (*List[float]*) is randomly picked from the List. If *None*, the elements are assumed to be of equal concentration.

cell_pert_frac float
fraction of cell perturbation

atom_pert_dist float
the atom perturbation distance (unit angstrom).

fmt str
the format of the returned conf strings. Should be one of the formats supported by *dpdata*

Returns

conf_list List[str]
A list of file content of configurations.

generate_systems(*numb_confs*, *concentration: Optional[Union[List[List[float]], List[float]]] = None*, *cell_pert_frac: float = 0.0*, *atom_pert_dist: float = 0.0*) → List[str]

Parameters

numb_confs int
Number of configurations to generate

concentration List[List[float]] or List[float] or None
If *List[float]*, the concentrations of each element. The length of the list should be the same as the *type_map*. If *List[List[float]]*, a list of concentrations (*List[float]*) is randomly picked from the List. If *None*, the elements are assumed to be of equal concentration.

cell_pert_frac float
fraction of cell perturbation

atom_pert_dist float
the atom perturbation distance (unit angstrom).

Returns

conf_list List[dpdata.System]
A list of generated confs in *dpdata.System*.

dpgen2.utils.alloy_conf.**gen_doc**(**, make_anchor=True, make_link=True, **kwargs*)

dpgen2.utils.alloy_conf.**generate_alloy_conf_args**()

dpgen2.utils.alloy_conf.**generate_alloy_conf_file_content**(*lattice: Union[System, Tuple[str, float]]*, *type_map: List[str]*, *numb_confs*, *replicate: Optional[Union[List[int], Tuple[int], int]] = None*, *concentration: Optional[Union[List[List[float]], List[float]]] = None*, *cell_pert_frac: float = 0.0*, *atom_pert_dist: float = 0.0*, *fmt: str = 'lammps/lmp'*)

dpgen2.utils.alloy_conf.**normalize**(*data*)

dpgen2.utils.chdir module

dpgen2.utils.chdir.**chdir**(*path_key: str*)
Returns a decorator that can change the current working path.

Parameters

path_key
[str] key to OPIO

Examples

```
>>> class SomeOP(OP):
...     @chdir("path")
...     def execute(self, ip: OPIO):
...         do_something()
```

dpgen2.utils.chdir.**set_directory**(*path: Path*)

Sets the current working path within the context.

Parameters

path
[Path] The path to the cwd

Yields

None

Examples

```
>>> with set_directory("some_path"):
...     do_something()
```

dpgen2.utils.dflow_config module

dpgen2.utils.dflow_config.**dflow_config**(*config_data*)

dpgen2.utils.dflow_query module

dpgen2.utils.dflow_query.**find_slice_ranges**(*keys: List[str]*, *sliced_subkey: str*)

find range of sliced OPs that matches the pattern 'iter-[0-9]*–{sliced_subkey}-[0-9]*'

dpgen2.utils.dflow_query.**get_last_iteration**(*keys: List[str]*)

get the index of the last iteraction from a list of step keys.

dpgen2.utils.dflow_query.**get_last_scheduler**(*wf: Any*, *keys: List[str]*)

get the output Scheduler of the last successful iteration

dpgen2.utils.dflow_query.**get_subkey**(*key: str*, *idx: Optional[int] = -1*)

dpgen2.utils.dflow_query.**print_keys_in_nice_format**(*keys: List[str]*, *sliced_subkey: List[str]*, *idx_fmt_len: int = 8*)

dpgen2.utils.dflow_query.**sort_slice_ops**(*keys: List[str]*, *sliced_subkey: List[str]*)

sort the keys of the sliced ops. the keys of the sliced ops contains sliced_subkey

dpgen2.utils.obj_artifact module

dpgen2.utils.obj_artifact.**dump_object_to_file**(*obj*, *fname*)
pickle dump object to a file

dpgen2.utils.obj_artifact.**load_object_from_file**(*fname*)
pickle load object from a file

dpgen2.utils.run_command module

dpgen2.utils.run_command.**run_command**(*cmd*, *shell=None*)

dpgen2.utils.step_config module

dpgen2.utils.step_config.**gen_doc**(*, *make_anchor=True*, *make_link=True*, ***kwargs*)

dpgen2.utils.step_config.**init_executor**(*executor_dict*)

dpgen2.utils.step_config.**lebesgue_executor_args**()

dpgen2.utils.step_config.**lebesgue_extra_args**()

dpgen2.utils.step_config.**normalize**(*data*)

dpgen2.utils.step_config.**step_conf_args**()

dpgen2.utils.step_config.**template_conf_args**()

dpgen2.utils.step_config.**variant_executor**()

dpgen2.utils.unit_cells module

class dpgen2.utils.unit_cells.**BCC**
Bases: object

Methods

gen_box	
numb_atoms	
poscar_unit	

gen_box()

numb_atoms()

poscar_unit(*latt*)

class dpgen2.utils.unit_cells.**DIAMOND**
Bases: object

Methods

gen_box	
numb_atoms	
poscar_unit	

gen_box()

numb_atoms()

poscar_unit(*latt*)

class dpgen2.utils.unit_cells.**FCC**

Bases: object

Methods

gen_box	
numb_atoms	
poscar_unit	

gen_box()

numb_atoms()

poscar_unit(*latt*)

class dpgen2.utils.unit_cells.**HCP**

Bases: object

Methods

gen_box	
numb_atoms	
poscar_unit	

gen_box()

numb_atoms()

poscar_unit(*latt*)

class dpgen2.utils.unit_cells.**SC**

Bases: object

Methods

gen_box	
numb_atoms	
poscar_unit	

gen_box()

numb_atoms()

poscar_unit(*latt*)

dpgen2.utils.unit_cells.**generate_unit_cell**(*crystal: str*, *latt: float = 1.0*) → System

6.1.2 Submodules

6.1.3 dpgen2.constants module

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W