dopingflow

Release 0.1.0

Kazem Zhour

Feb 13, 2026

USER GUIDE

Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

CHAPTER

ONE

WELCOME TO DOPINGFLOW’S DOCUMENTATION

1.1 Workflow Overview

1.1.1 Conceptual Pipeline

The ML Doping Workflow implements a fully automated, multi-stage surrogate pipeline for the exploration of doped crystalline materials.

1.1.2 Pipeline Structure

Enumeration $\rightarrow$ Ranking $\rightarrow$ Relaxation $\rightarrow$ Filtering $\rightarrow$ Band Gap $\rightarrow$ Formation Energy $\rightarrow$ Database

1.1.3 Stages

1. Symmetry-reduced dopant enumeration
2. ML-based energy ranking (M3GNet)
3. Full cell relaxation (M3GNet + FIRE)
4. Energy-window filtering
5. Band gap prediction (ALIGNN)
6. Formation energy evaluation using ML chemical potentials
7. Database assembly

1.2 Installation, Usage, and Outputs

This page explains how to install **dopingflow** and how to run the workflow either step-by-step or using the single orchestration command.

1.2.1 Installation

Clone the repository and install in editable mode:

```
git clone KazemZh/dopingflow
cd dopingflow
python -m venv .venv
source .venv/bin/activate
pip install -U pip
pip install -e .
```

Verify the CLI is available:

```
dopingflow --help
```

1.2.2 Required Inputs

Refer to *Required Input Files* page.

1.2.3 Running the Workflow

All commands accept `-c/--config` to specify the TOML file. If omitted, `input.toml` in the current directory is used.

Run the full pipeline with one command

To run the complete workflow in order:

```
dopingflow run-all -c input.toml
```

This executes:

`refs -> generate -> scan -> relax -> filter -> bandgap -> formation -> collect`

Resuming and partial runs (run-all)

You can resume from a given stage:

```
dopingflow run-all -c input.toml --from relax
```

You can stop at a stage (inclusive). This is useful if you do not want to run bandgap yet:

```
dopingflow run-all -c input.toml --until filter
```

You can print the planned steps without running them:

```
dopingflow run-all -c input.toml --dry-run
```

You can run only a subset of steps inside a selected range:

```
dopingflow run-all -c input.toml --from refs --until collect --only refs,generate,scan
```

Filtering controls inside run-all

The filter stage supports optional overrides (passed through by `run-all`):

- Restrict filtering to a single composition folder:

  ```
  dopingflow run-all -c input.toml --from relax --until filter --filter-only Sb5_Zr5
  ```

- Force re-filtering even if outputs exist:

  ```
  dopingflow run-all -c input.toml --from filter --until filter --force
  ```

- Override filtering mode by specifying one of:

  ```
  dopingflow run-all -c input.toml --from filter --until filter --window-mev 50
  dopingflow run-all -c input.toml --from filter --until filter --topn 12
  ```

Step-by-step execution

Step 00: reference energies:

```
dopingflow refs-build -c input.toml
```

Step 01: structure generation:

```
dopingflow generate -c input.toml
```

Step 02: scan (symmetry-unique enumeration + M3GNet single-point energies):

```
dopingflow scan -c input.toml
```

Step 03: relax scanned candidates (M3GNet Relaxer):

```
dopingflow relax -c input.toml
```

Step 04: filter relaxed candidates:

```
dopingflow filter -c input.toml
```

Optional Step 05: predict bandgap (ALIGNN):

Before running bandgap, set the model path:

```
export ALIGNN_MODEL_DIR=/path/to/your/alignn/model_root
dopingflow bandgap -c input.toml
```

Step 06: formation energies:

```
dopingflow formation -c input.toml
```

Step 07: collect results into one CSV database:

```
dopingflow collect -c input.toml
```

1.2.4 Outputs Overview

This section summarizes the main outputs created by each stage.

Step 00 (refs-build)

Writes:

- `reference_structures/reference_energies.json`

This file contains:

- the relaxed pristine supercell energy
- per-atom chemical potentials for host and dopant species
- metadata about reference structures and relaxation settings

Step 01 (generate)

Writes a structure folder per composition under `[structure].outdir` (default: `random_structures`):

- `<outdir>/<composition_tag>/POSCAR`
- `<outdir>/<composition_tag>/metadata.json`

Step 02 (scan)

Inside each `<composition_tag>/` folder, writes:

- `ranking_scan.csv` (top-k single-point energies)
- `scan_summary.txt` (human-readable summary)
- candidate structures:

```
<composition_tag>/candidate_###/01_scan/POSCAR
<composition_tag>/candidate_###/01_scan/meta.json
```

Step 03 (relax)

For each candidate:

- `candidate_###/02_relax/POSCAR`
- `candidate_###/02_relax/meta.json`

Also writes per composition folder:

- `ranking_relax.csv`

Step 04 (filter)

Writes per composition folder:

- `ranking_relax_filtered.csv` (filtered candidate table)
- `selected_candidates.txt` (names of kept candidates)

Step 05 (bandgap)

Writes per composition folder:

- `bandgap_alignn_summary.csv`

Writes per candidate:

- `candidate_###/03_band/meta.json`

Step 06 (formation)

Writes per composition folder:

- `formation_energies.csv`

Writes per candidate:

- `candidate_###/04_formation/meta.json`

Step 07 (collect)

Writes one flat CSV in the workflow root:

- `results_database.csv`

This file is a compact "database view" across compositions and selected candidates, combining scan/relax/filter/bandgap/formation results where available.

1.2.5 Tips

- Use `--verbose` with any command for more detailed logs:

```
dopingflow run-all -c input.toml --verbose
```

- If bandgap is not configured yet (no `ALIGNN_MODEL_DIR`), stop before bandgap:

```
dopingflow run-all -c input.toml --until filter
```

1.3 Required Input Files

This page summarizes **all external files required to run the workflow**.

The workflow itself is code-driven, but it depends on a small number of user-provided structure files and a configuration file.

1.3.1 Overview

At minimum, the following files are required:

1. *input.toml* (workflow configuration).
2. Pristine unit-cell structure of the crystal to be doped (POSCAR format)
3. Optional: host and dopant reference bulk structures (POSCAR format)

All file paths are interpreted relative to the directory containing `input.toml` unless absolute paths are used.

1.3.2 Directory Layout Example

A clean minimal directory structure may look like:

```
project_root/
    input.toml
    reference_structures/
        base.POSCAR         # pristine unit cell
        host.POSCAR         # host elemental bulk (for formation energies)
        dopant1.POSCAR      # dopant bulk reference
        dopant2.POSCAR      # dopant bulk reference
        dopant3.POSCAR      # dopant bulk reference
        ...
```

Notes:

- All structure files may be placed inside `reference_structures/`.
- `base.POSCAR` is the pristine crystal structure used for supercell generation.

- The elemental POSCAR files (host.POSCAR, dopant1.POSCAR, dopant2.POSCAR …) are only required if formation energies are computed using local bulk references.
- The exact filenames are user-defined, but must match what is specified in `input.toml`.

1.3.3 ALIGNN Model Directory (Environment Variable)

For bandgap prediction (Step 05), a trained ALIGNN model must be available.

The path must be set via environment variable:

```
export ALIGNN_MODEL_DIR=/path/to/alignn/model
```

This directory must contain:

- `config.json`
- `checkpoint_*.pt`

Without this variable, Step 05 will fail.

1.3.4 Important Notes

- Dopant unit-cell POSCAR files are **not required for structure generation**. Doping is substitutional and uses the pristine structure only.
- Dopant bulk POSCAR files are only needed if: - You compute formation energies using local references.
- The workflow does **not** require separate dopant unit-cell structures for substitution.

1.3.5 Summary

Minimum to start:

- `input.toml`
- Pristine POSCAR

To enable full workflow including formation energies and bandgaps:

- Reference bulk structures (or database access)
- ALIGNN model directory

1.4 Input File Specification (input.toml)

The workflow is fully controlled through a single TOML configuration file.

Each stage reads only the parameters relevant to it. All paths are interpreted relative to the directory containing `input.toml`.

The following sections are supported:

- `[structure]`
- `[doping]`
- `[generate]`
- `[scan]`
- `[relax]`
- `[filter]`

- `[bandgap]`
- `[formation]`
- `[database]` (optional)

Not all sections are required for every stage. Each stage reads only what it needs.

1.4.1 [structure]

Defines the base structure and global output directory.

base_poscar (string)

Path to the pristine structure file (POSCAR format).

Used in: - Step 00 (reference construction) - Step 01 (structure generation)

supercell (array of 3 integers)

Supercell expansion applied to the pristine structure:

```
supercell = [nx, ny, nz]
```

Used in: - Step 00 - Step 01

outdir (string, default: "random_structures")

Directory where all generated composition folders are written.

This directory becomes the root for: - Step 01 outputs - Step 02–06 per-composition subfolders

1.4.2 [doping]

Defines substitutional doping behavior.

host_species (string, required)

Element symbol of the host species to be substituted.

Used in: - Step 01 (generation) - Step 02 (scan) - Step 06 (formation)

mode (string)

Defines doping mode:

- `"explicit"` — user provides exact compositions.
- `"enumerate"` — workflow constructs compositions combinatorially.

max_dopants_total (integer)

Maximum number of distinct dopant species allowed per structure.

Only enforced in enumerate mode.

dopant_elements (array of strings)

List of possible dopant species.

required_elements (array of strings, optional)

Subset of dopants that must appear in enumerated compositions.

total_levels (array of floats)

Allowed total dopant percentages (relative to host sites).

per_dopant_levels (array of floats)

Discrete allowed concentrations per dopant.

1.4.3 [generate]

Controls structure writing and reproducibility (Step 01).

seed_base (integer)

Base seed for deterministic random substitution.

Ensures reproducible structure generation.

poscar_order (array of strings, required)

Defines element ordering in written POSCAR files.

Must be non-empty.

Used consistently in: - Step 01 - Step 02 - Step 03

1.4.4 [scan]

Controls symmetry enumeration and single-point prescreening (Step 02).

poscar_in (string, default: "POSCAR")

Filename inside each composition folder used as enumeration input.

topk (integer)

Number of lowest-energy configurations retained.

symprec (float)

Tolerance passed to `SpacegroupAnalyzer` for symmetry detection.

max_enum (integer)

Maximum allowed number of raw combinatorial configurations. Prevents runaway combinatorics.

max_unique (integer)

Maximum allowed number of symmetry-unique configurations.

nproc (integer)

Number of parallel worker processes for energy evaluation.

chunksize (integer)

Chunk size used in multiprocessing pool.

anion_species (array of strings)

Species excluded from substitutional enumeration (e.g., oxygen).

1.4.5 [relax]

Controls structural relaxation (Step 03).

fmax (float)

Maximum force convergence criterion (eV/Å).

n_workers (integer)

Number of parallel relaxation workers.

tf_threads (integer)

TensorFlow intra/inter-op thread count per worker.

omp_threads (integer)

OpenMP thread count per worker.

skip_if_done (boolean)

If true, skip composition folder if `ranking_relax.csv` exists.

skip_candidate_if_done (boolean)

If true, skip candidate if `02_relax/meta.json` exists.

1.4.6 [filter]

Controls candidate selection after relaxation (Step 04).

mode (string)

Either:

- `"window"` — keep structures within an energy window
- `"topn"` — keep lowest N structures

window_meV (float)

Energy window (meV above minimum) used when mode = `"window"`.

max_candidates (integer)

Number of candidates kept when mode = `"topn"`.

skip_if_done (boolean)

Skip filtering if output files already exist.

1.4.7 [bandgap]

Controls ALIGNN bandgap prediction (Step 05).

skip_if_done (boolean)

Skip bandgap calculation if summary file exists.

cutoff (float)

Radial cutoff for graph construction (Å).

max_neighbors (integer)

Maximum neighbors per atom in graph.

Note: The ALIGNN model directory must be defined via environment variable:

```
ALIGNN_MODEL_DIR=/path/to/model
```

1.4.8 [formation]

Controls formation energy calculation (Step 06).

skip_if_done (boolean)

Skip formation calculation if output exists.

normalize (string)

Defines reported energy normalization:

- `"total"` — total formation energy (eV)
- `"per_dopant"` — eV per substituted dopant atom
- `"per_host"` — eV per atom in supercell

Formation energies require:

- Step 00 reference construction completed
- `reference_structures/reference_energies.json`

1.4.9 [database] (optional)

Controls final database collection (Step 07).

skip_if_done (boolean, default: true)

If true, do not overwrite existing `results_database.csv`.

If this section is omitted, default behavior applies.

1.4.10 Design Principles

- All stages are deterministic given fixed input.
- All randomness is seed-controlled.
- Each stage can be skipped independently via `skip_if_done`.
- Each stage writes explicit metadata for full reproducibility.
- All outputs are composition-folder scoped except final database export.

1.5 0. Reference Energy Construction

1.5.1 Implementation

This stage is implemented in:

```
src/dopingflow/refs.py
```

The public entry point is:

```
run_refs_build(...)
```

1.5.2 Purpose

This stage constructs the reference energies required for formation energy evaluation of doped structures.

The following quantities are computed:

- The relaxed total energy of the pristine supercell
- The elemental chemical potentials of all species involved (host + dopants)

All reference data are written to:

```
reference_structures/reference_energies.json
```

These references are later used for formation energy evaluation.

1.5.3 Inputs

This stage uses settings from:

- `[structure]`: provides the pristine structure and supercell.
- `[doping]`: defines the host species and dopant set.
- `[references]`: controls reference source, relaxation settings, and caching behavior.

1.5.4 Method Summary

1. Read the pristine unit-cell structure.
2. Apply the workflow supercell expansion.
3. Relax the pristine supercell.
4. Identify all elements involved (host + dopants).
5. For each element: a. Obtain a bulk structure (local or external source). b. Relax the bulk structure. c. Compute per-atom chemical potential.
6. Store all reference quantities in a JSON file.

1.5.5 Formation Energy Framework

The workflow assumes substitutional doping.

The formation energy is defined as:

$$E_{\mathrm{form}} = E_{\mathrm{doped}} - E_{\mathrm{pristine}} + \sum_i n_i \left(\mu_{\mathrm{host}} - \mu_i\right)$$

where:

- E_{doped} is the relaxed total energy of the doped supercell
- E_{pristine} is the relaxed total energy of the pristine supercell
- μ_i is the chemical potential of dopant species i
- μ_{host} is the chemical potential of the substituted host species
- n_i is the number of substituted atoms of species i

This corresponds to removing host atoms and inserting dopant atoms while preserving total lattice size.

1.5.6 Pristine Supercell Energy

The pristine reference energy is computed by:

1. Reading the unit-cell structure defined in the input.
2. Applying the supercell defined in `[structure].supercell`.
3. Performing full structural relaxation.
4. Extracting the final total energy.

Both atomic positions and lattice vectors are allowed to relax.

1.5.7 Elemental Chemical Potentials

For each relevant element, a bulk structure is:

- Obtained either from a local file or an external database.
- Fully relaxed.
- Used to compute a per-atom chemical potential:

$$\mu_i = \frac{E_{\mathrm{bulk}}}{N_{\mathrm{atoms}}}$$

where:

- E_{bulk} is the relaxed total energy of the bulk structure.
- N_{atoms} is the number of atoms in the bulk cell.

1.5.8 Reference Sources

Two reference sources are supported.

Local bulk structures

Bulk structures are read from a user-specified directory. Each element must have a corresponding structure file.

External database

Bulk structures may be retrieved programmatically from an external materials database using provided identifiers and API credentials.

Downloaded structures are cached locally to ensure reproducibility.

1.5.9 Relaxation Method

All reference relaxations use:

- Interatomic potential: M3GNet
- Optimizer: FIRE
- Convergence criterion: maximum force below `fmax`

The relaxations are fully unconstrained (cell parameters and atomic positions).

1.5.10 Caching Strategy

If:

```
skip_if_done = true
```

and the reference JSON already exists, this stage is skipped.

This ensures deterministic behavior and avoids unnecessary recomputation.

1.5.11 Outputs

The file `reference_energies.json` contains:

- Timestamp
- Host species
- Supercell definition
- Relaxed pristine supercell energy
- Chemical potentials per element
- Metadata describing bulk relaxation conditions
- Reference source information

1.5.12 Notes and Limitations

- This stage does not evaluate doped structures.
- Energies are ML-predicted (not DFT-level total energies).
- Reference phase selection affects chemical potentials.
- No finite-size corrections are applied.
- No charge-state corrections are included.
- No entropy or temperature effects are considered.
- No competing phase stability analysis is performed.

1.6 1. Structure Generation

1.6.1 Implementation

This stage is implemented in:

```
src/dopingflow/generate.py
```

The public entry point is:

```
run_generate(...)
```

1.6.2 Purpose

This stage generates an initial set of doped structures starting from a pristine unit cell. The output is a directory of subfolders, each containing a VASP `POSCAR` and a small metadata file describing the effective composition and the random seed used for site selection.

The generation step supports two workflows:

- **Explicit compositions**: the user provides exact dopant percentages.
- **Enumerated compositions**: the workflow constructs a systematic set of dopant combinations and doping levels under user-defined constraints.

1.6.3 Inputs

This stage uses settings from three sections of `input.toml`:

- `[structure]`: provides the pristine structure file and the supercell size.
- `[doping]`: defines the doping mode and composition rules.
- `[generate]`: controls structure-writing details and reproducible randomness.

1.6.4 Method Summary

1. Read the pristine structure from `[structure].base_poscar`.
2. Build the supercell using `[structure].supercell`.
3. Identify all sites matching the substitution host species (`[doping].host_species`).
4. For each requested composition:
    a. Convert requested dopant percentages to integer substitution counts by rounding.

b. Randomly choose host sites to substitute, using a deterministic seed.

c. Optionally reorder species in the written POSCAR.

d. Write `POSCAR` and `metadata.json` to an output subdirectory.

The output directory is `[structure].outdir` (default: `random_structures`).

1.6.5 Composition Handling

Requested vs effective composition

Doping levels are provided as percentages *relative to the number of host sites* in the generated supercell. Since the number of sites is discrete, requested percentages are converted to integer substitution counts by rounding.

The workflow therefore distinguishes:

- **requested composition**: the percentages from the input
- **effective composition**: the percentages implied by the rounded integer counts

If rounding changes any dopant level, warnings are reported and the **effective** composition is stored in the metadata.

A basic consistency check is applied:

- the total number of substituted atoms may not exceed the number of host sites
- the total requested dopant percentage may not exceed 100%

Explicit mode

In explicit mode, each composition is provided directly by the user as a mapping `element -> percent`. Each composition produces exactly one structure.

This mode is recommended when:

- specific compositions are already known or desired
- only a small number of target compositions are needed

Enumerate mode

In enumerate mode, the workflow generates composition dictionaries automatically from:

- a list of possible dopant species
- an optional list of dopants that must appear
- a set of allowed total dopant levels
- a set of discrete per-dopant levels

The workflow enumerates combinations of distinct dopants of size:

$$k \in \{1, \ldots, k_{\max}\}$$

where $k_{\max}$ is controlled by the input parameter `[doping].max_dopants_total`.

Interpretation:

- you may provide many candidate dopant elements in the input
- but each generated structure contains at most `max_dopants_total` distinct dopant species *when using enumerate mode*

(Explicit mode is not inherently limited unless the user adopts the same constraint.)

1.6.6 Reproducible Random Substitution

The actual substitutional sites are selected randomly, but deterministically:

- a **composition tag** is constructed from the effective composition
- a stable seed is derived from the tag and a base seed (`[generate].seed_base`)

This ensures that rerunning the workflow with unchanged input produces identical structures, while still providing randomized site selection.

1.6.7 Directory Naming and Collision Handling

Each generated structure is written to:

```
<outdir>/<composition_tag>/POSCAR
<outdir>/<composition_tag>/metadata.json
```

The folder tag is constructed from the **effective** composition. If two different requested compositions round to the same effective composition, a suffix is added:

```
<tag>__v2, <tag>__v3, ...
```

This guarantees that all generated structures are preserved.

1.6.8 POSCAR Species Ordering

To control the species order in the written POSCAR, you may provide:

```
[generate]
poscar_order = [...]
```

If this list is empty, the structure is written using pymatgen's default ordering. If non-empty, sites are reordered to match the given preference (and any remaining species are appended).

1.6.9 Outputs

For each generated structure:

- `POSCAR`: doped supercell structure
- `metadata.json`: provenance information, including:
  - host species and number of host sites
  - requested and effective compositions
  - rounded substitution counts
  - seed used for deterministic substitution
  - composition tag and input file name

1.6.10 Notes and Limitations

- This stage performs **structure generation only** and does not evaluate stability.
- Rounding is unavoidable for small supercells; larger supercells reduce rounding error.
- Enumerated composition counts can grow combinatorially with the number of dopants and levels; users should choose constraints accordingly.

1.7 2. Symmetry-Reduced Energy Pre-screening (Scanning)

1.7.1 Implementation

This stage is implemented in:

```
src/dopingflow/scan.py
```

The public entry point is:

```
run_scan(...)
```

1.7.2 Purpose

This stage performs an efficient **energy pre-screening** of dopant arrangements by enumerating **symmetry-unique** configurations on a selected sublattice and evaluating their **single-point energies** using a machine-learned interatomic potential.

For each generated structure folder, the method:

- infers the substitutional sublattice and dopant counts from the input structure
- enumerates symmetry-unique dopant permutations on that sublattice
- predicts a single-point energy for each unique configuration
- keeps the **top-k lowest-energy** candidates for downstream relaxation

1.7.3 Inputs

This stage uses settings from the following sections of `input.toml`:

- `[structure]`: provides the output directory containing generated structures.
- `[doping]`: provides the host species definition.
- `[generate]`: provides the species ordering used when writing POSCAR files.
- `[scan]`: controls enumeration limits, symmetry tolerance, parallelism, and selection.

1.7.4 Method Summary

For each structure subdirectory in `[structure].outdir`:

1. Read the structure file specified by `[scan].poscar_in`.
2. Identify the **enumeration sublattice** (all non-anion sites) and infer: - host species count on the sublattice - dopant species counts on the sublattice
3. Estimate the total number of raw (non-symmetry-reduced) configurations.
4. Construct a parent structure and compute symmetry operations acting on the sublattice.
5. Enumerate dopant labelings and reduce them to **symmetry-unique** configurations.
6. Evaluate a single-point energy for each unique configuration in parallel (multiprocessing).
7. Select the lowest-energy `topk` candidates and write them to `candidate_*/01_scan`.
8. Write a CSV ranking file and a human-readable summary.

1.7.5 Enumeration Sublattice Definition

The enumerated sublattice is inferred from the input structure using the rule:

- **anion sites** are excluded (defined by `[scan].anion_species`)
- **all remaining sites** form the enumeration sublattice

This design makes the method general across crystals where doping occurs on a cation (or non-anion) sublattice.

The dopant counts are inferred directly from the input POSCAR by counting species on the enumeration sublattice. The host species is provided by `[doping].host_species`.

1.7.6 Combinatorics and Safety Limits

Raw configuration count

Given a sublattice with N sites and fixed dopant counts, the number of raw configurations (before symmetry reduction) grows combinatorially.

To avoid runaway enumeration, the workflow:

- estimates the raw count and enforces `[scan].max_enum`
- enforces a hard limit on symmetry-unique configurations via `[scan].max_unique`

If either limit is exceeded, the scan stops for that structure and reports an error. This protects against infeasible compositions and/or large supercells.

Maximum number of dopant species per scan

The current symmetry-unique enumerator is implemented explicitly for up to **three distinct dopant species** on the enumerated sublattice.

If the inferred dopant set contains more than three species, the scan raises an error.

This limit applies to the scanning enumeration procedure (not necessarily to other stages).

1.7.7 Symmetry Reduction

To avoid evaluating symmetry-equivalent configurations, the scan:

1. Constructs a **parent** structure where all sites of the enumeration sublattice are assigned to the host species.
2. Uses a space-group analysis (with tolerance `[scan].symprec`) to obtain symmetry operations.
3. Converts symmetry operations into **permutations of sublattice indices**.
4. For each dopant labeling, computes a canonical representation under these permutations.
5. Keeps only labelings with unique canonical keys.

This yields the set of symmetry-unique dopant configurations.

1.7.8 Energy Model and Parallel Evaluation

Each symmetry-unique configuration is evaluated using a **single-point** energy prediction with M3GNet.

Key points:

- the structure is not relaxed in this stage
- only the energy is predicted (fast screening)
- energies are computed in parallel using multiprocessing

To improve robustness when using TensorFlow-based models, worker processes are created using the `spawn` start method.

1.7.9 Selection of Top-k Candidates

The workflow keeps the `[scan].topk` configurations with the **lowest predicted** single-point energies and writes each to:

```
<structure_dir>/candidate_###/01_scan/POSCAR
<structure_dir>/candidate_###/01_scan/meta.json
```

A CSV ranking file is written to the structure directory:

```
<structure_dir>/ranking_scan.csv
```

The ranking CSV contains:

- candidate name
- rank (single-point)
- predicted single-point energy
- a short dopant-position signature

1.7.10 Reproducibility

For a fixed input structure and scan settings, the result is deterministic with respect to:

- inferred dopant counts
- symmetry analysis (controlled by `symprec`)
- enumeration ordering
- the ML model used for prediction

Parallel evaluation does not affect the final ranking, since candidates are selected based on energy values.

1.7.11 Outputs

For each processed structure folder:

- `candidate_*/01_scan/POSCAR`: POSCAR files of selected low-energy candidates
- `candidate_*/01_scan/meta.json`: metadata describing scan settings and counts
- `ranking_scan.csv`: ranked list of candidates with predicted energies
- `scan_summary.txt`: human-readable summary of the scan

1.7.12 Notes and Limitations

- This stage performs **single-point ML energy screening only**; it does not relax structures.
- Symmetry reduction depends on the tolerance `symprec`; too large values may merge distinct configurations, too small values may reduce symmetry detection.
- Enumeration can become infeasible for large sublattices and/or high dopant counts; limits `max_enum` and `max_unique` are enforced to prevent runaway runtime/memory.
- The current label enumerator supports up to **three distinct dopant species**.

- Predicted single-point energies are surrogate ML values and should be interpreted as a ranking heuristic rather than absolute thermodynamic energies.

1.8 3. Candidate Relaxation

1.8.1 Implementation

This stage is implemented in:

```
src/dopingflow/relax.py
```

The public entry point is:

```
run_relax(...)
```

1.8.2 Purpose

This stage performs **full structural relaxation** of the low-energy candidates selected during the scanning stage.

For each candidate structure:

- atomic positions are relaxed
- lattice vectors are relaxed
- the final total energy is extracted
- relaxed structures are written to disk
- a relaxation ranking is generated

This stage refines the single-point screening results by allowing full geometry optimization.

1.8.3 Inputs

This stage uses settings from the following sections of `input.toml`:

- `[structure]`: provides the output directory containing structure folders.
- `[generate]`: provides the species ordering used when writing POSCAR files.
- `[relax]`: controls convergence tolerance, parallelism, and skipping logic.

The relaxation stage processes:

```
<outdir>/<structure_folder>/candidate_*/01_scan/POSCAR
```

1.8.4 Method Summary

For each structure folder inside `[structure].outdir`:

1. Detect all `candidate_*` subfolders.
2. For each candidate: a. Read `01_scan/POSCAR`. b. Perform full structural relaxation. c. Write relaxed structure to `02_relax/POSCAR`. d. Write relaxation metadata.
3. Rank candidates by relaxed total energy.
4. Write `ranking_relax.csv` per structure folder.

1.8.5 Relaxation Model

Structural relaxation is performed using:

- Interatomic potential: **M3GNet (pretrained)**
- Optimizer: internal Relaxer (FIRE-based)
- Convergence criterion: maximum force below `[relax].fmax`

Both:

- atomic coordinates
- lattice parameters

are allowed to relax.

1.8.6 Parallel Execution

Relaxation is parallelized **over candidates** using multiprocessing.

Each worker process:

- initializes its own M3GNet Relaxer instance
- sets thread limits to control TensorFlow and OpenMP usage
- runs independently

The number of parallel workers is controlled by:

```
[relax]
n_workers = ...
```

Additional thread control parameters:

```
tf_threads
omp_threads
```

These settings allow tuning CPU usage on workstations or HPC systems.

1.8.7 Ranking Logic

After all candidates are processed within a structure folder:

- Only successfully relaxed candidates (status `ok`) are ranked.
- Ranking is based on ascending relaxed total energy.

The resulting CSV file:

```
ranking_relax.csv
```

contains:

- candidate name
- relaxed rank
- relaxed energy
- original single-point rank
- original single-point energy

- signature
- status
- walltime
- error message (if any)

1.8.8 Reproducibility and Skipping

Two levels of skipping are supported:

Folder-level skipping

If:

```
[relax].skip_if_done = true
```

and `ranking_relax.csv` already exists in a structure folder, the entire folder is skipped.

Candidate-level skipping

If:

```
[relax].skip_candidate_if_done = true
```

and:

```
candidate_*/02_relax/meta.json
candidate_*/02_relax/POSCAR
```

already exist, that individual candidate is skipped.

This enables safe restart of interrupted runs.

1.8.9 Outputs

For each candidate:

```
candidate_###/
    01_scan/
    02_relax/
        POSCAR
        meta.json
```

The relaxation metadata includes:

- relaxed total energy
- walltime
- species counts
- convergence target (fmax)
- link to original scan metadata

For each structure folder:

```
ranking_relax.csv
```

which ranks relaxed candidates by energy.

1.8.10 Error Handling

If relaxation fails:

- an error is written to `02_relax/meta.json`
- the candidate is marked with status `fail`
- ranking proceeds with remaining successful candidates

Failures do not abort the entire structure folder.

1.8.11 Notes and Limitations

- Relaxation uses a pretrained ML potential (not DFT-level relaxation).
- No charge states or external fields are included.
- No temperature or vibrational effects are considered.
- Energies are total energies from the ML model.
- Convergence depends on the chosen `fmax` threshold.
- Parallel performance depends on CPU availability and thread configuration.

1.9 4. Relaxed-Candidate Filtering

1.9.1 Implementation

This stage is implemented in:

```
src/dopingflow/filtering.py
```

The public entry point is:

```
run_filtering(...)
```

1.9.2 Purpose

This stage selects a **reduced set of relaxed candidates** for downstream property calculations by filtering the results of the relaxation stage.

It operates on the per-folder relaxation ranking produced in Step 03 and writes:

- a filtered ranking table
- a plain text list of selected candidate names

This stage is a lightweight post-processing step; it does not run any atomistic calculations.

1.9.3 Inputs

This stage uses settings from the following sections of `input.toml`:

- `[structure]`: provides the output directory containing structure folders.
- `[filter]`: defines the filtering strategy and thresholds.

It expects that Step 03 has already produced, in each structure folder:

```
ranking_relax.csv
```

1.9.4 Method Summary

For each structure folder inside `[structure].outdir`:

1. Read `ranking_relax.csv`.
2. Keep only candidates with `status == "ok"`.
3. Determine the minimum relaxed energy:
   - $E_{\text{min}} = \min(E_{\text{relaxed}})$
4. Apply one of two filtering modes:
   - *window mode*: keep candidates within an energy window above E_{min}
   - *top-n mode*: keep the lowest-energy `N` candidates
5. Write filtered outputs:
   - `ranking_relax_filtered.csv`
   - `selected_candidates.txt`

1.9.5 Filtering Modes

Window mode

If the filter mode is set to `window`, candidates are kept if:

$$E_{\text{relaxed}} - E_{\text{min}} \leq \Delta E$$

where:

- $\Delta E = \text{window\_meV}/1000$ (converted from meV to eV)

This selects all structures that lie within a user-defined energy window above the best relaxed candidate.

Top-n mode

If the filter mode is set to `topn`, candidates are kept by selecting the first `max_candidates` entries after sorting by relaxed energy.

This guarantees a fixed number of candidates per structure folder (unless fewer successful relaxations exist).

1.9.6 Selection Basis

Filtering is based exclusively on:

- `energy_relaxed_eV` from `ranking_relax.csv`

The filter ignores candidates that:

- are missing required fields
- have non-numeric energies
- have `status` values other than `ok`

1.9.7 Outputs

For each structure folder, this stage writes:

Filtered ranking table

```
ranking_relax_filtered.csv
```

with columns including:

- `rank_filtered`: rank within the filtered set (starting at 1)
- `candidate`: candidate folder name
- `energy_relaxed_eV`: relaxed energy (eV)
- `delta_e_eV`: energy relative to the folder minimum
- provenance columns copied from the relaxation stage (e.g. scan rank/signature)
- `filter_mode`: a string describing the applied filter rule

Selected candidate list

```
selected_candidates.txt
```

This is a newline-separated list of candidate folder names, in the same order as the filtered ranking table.

1.9.8 Command-line Overrides and Forcing

The implementation supports runtime overrides that can force the filter behavior independently of the default TOML settings:

- overriding `window_meV` forces window mode
- overriding `topn` forces top-n mode

A force flag can be used to regenerate outputs even if they already exist.

1.9.9 Reproducibility and Skipping

If:

```
[filter].skip_if_done = true
```

and both output files already exist in a folder, that folder is skipped unless a force option is used.

Since this stage is pure file processing, its results are deterministic given the input `ranking_relax.csv` and filter parameters.

1.9.10 Notes and Limitations

- This stage performs **no new calculations**; it filters results from Step 03.
- Filtering is done independently per structure folder; it does not compare energies across different compositions/folders.
- Energy windows are applied relative to the minimum energy within each folder, not relative to a global minimum.

1.10 5. Bandgap Prediction

1.10.1 Implementation

This stage is implemented in:

```
src/dopingflow/bandgap.py
```

The public entry point is:

```
run_bandgap(...)
```

1.10.2 Purpose

This stage predicts the **electronic band gap** of relaxed candidate structures using a locally available **ALIGNN** model.

Band gaps are evaluated for candidates produced by the previous steps (typically after relaxation, and optionally after filtering) and written to:

- a per-folder summary CSV
- a per-candidate metadata record

This is a machine-learning inference step; no electronic-structure calculation is performed here.

1.10.3 Inputs

This stage uses settings from the following sections of `input.toml`:

- `[structure]`: provides the output directory containing structure folders.
- `[bandgap]`: defines graph-construction parameters and skipping behavior.

In addition, this stage requires an environment variable pointing to a local ALIGNN model directory:

```
ALIGNN_MODEL_DIR=/path/to/your/alignn_model_folder
```

1.10.4 Method Summary

For each structure folder inside `[structure].outdir`:

1. Determine the list of candidate structures to evaluate:
   a. If `selected_candidates.txt` exists, only those candidates are used.
   b. Otherwise, all `candidate_*/02_relax/POSCAR` files are used.
2. For each selected candidate:
   a. Read the relaxed `POSCAR` (from Step 03).
   b. Convert the structure to a graph representation (DGL graph) using a neighbor cutoff and maximum neighbor count.
   c. Run a forward pass of the ALIGNN model to obtain the predicted band gap.
   d. Write a per-candidate metadata file.
3. Write a per-folder summary CSV listing band gaps for all evaluated candidates.

1.10.5 Selection of Candidates

This stage supports two selection modes:

- **Filtered selection (recommended)**: If `selected_candidates.txt` exists in the folder, the band gap is computed only for those candidates. This integrates naturally with Step 04 filtering.
- **Fallback selection**: If no selection list is present, all relaxed candidates are used.

The relaxed structures are always taken from:

```
candidate_*/02_relax/POSCAR
```

1.10.6 Model and Graph Construction

ALIGNN model loading

The model is loaded from a local directory under `ALIGNN_MODEL_DIR`. The code searches for a folder containing a `config.json` and a checkpoint file (e.g. `checkpoint_*.pt` or `*.pt`), then loads:

- the model configuration (from `config.json`)
- the checkpoint weights

Graph parameters

For each structure, a graph is constructed using:

- `[bandgap].cutoff`: neighbor cutoff distance
- `[bandgap].max_neighbors`: maximum neighbors per atom

These parameters control how local environments are encoded for inference.

The DGL backend is set to PyTorch in this stage (via `DGLBACKEND=pytorch`), which must be configured before importing DGL/ALIGNN.

1.10.7 Outputs

Per-folder summary

For each structure folder, this stage writes:

```
bandgap_alignn_summary.csv
```

This file contains:

- `candidate`: candidate directory name
- `bandgap_eV_ALIGNN_MBJ`: predicted band gap (eV)

The CSV is sorted by band gap (ascending) for convenience.

Per-candidate metadata

For each evaluated candidate, this stage writes:

```
candidate_XXX/03_band/meta.json
```

This metadata includes:

- predicted band gap value

- model provenance (model directory, config path, checkpoint path)
- input structure path
- structure size and composition
- graph parameters used for inference
- walltime for prediction

1.10.8 Reproducibility and Skipping

If:

```
[bandgap].skip_if_done = true
```

and `bandgap_alignn_summary.csv` already exists for a folder, that folder is skipped.

Given the same relaxed POSCARs, model checkpoint, and graph parameters, this stage is deterministic.

1.10.9 Notes and Limitations

- Predicted band gaps depend on the chosen ALIGNN model and its training domain.
- Band gaps are ML predictions and should not be interpreted as DFT-quality values unless the chosen model has been validated for the target material class.
- This stage assumes candidate structures are already relaxed; it does not perform geometry optimization.

1.11 6. Formation Energy Evaluation

1.11.1 Implementation

This stage is implemented in:

```
src/dopingflow/formation.py
```

The public entry point is:

```
run_formation(...)
```

1.11.2 Purpose

This stage computes the **formation energy** of relaxed doped structures using reference energies constructed in Step 00.

It combines:

- the relaxed total energy of each doped candidate structure (E_{doped})
- the relaxed total energy of the pristine supercell (E_{pristine})
- elemental chemical potentials (μ_i) for host and dopant species

Formation energies are written per composition folder to:

- `formation_energies.csv` (summary table)
- `candidate_*/04_formation/meta.json` (per-candidate provenance)

1.11.3 Inputs

This stage uses settings from the following sections of `input.toml`:

- `[structure]`: provides the output directory containing structure folders.
- `[doping]`: defines the substitution host species.
- `[scan]`: provides the anion species list used to identify dopants.
- `[formation]`: controls skipping and the normalization convention.

It also requires the reference-energy JSON from Step 00:

```
reference_structures/reference_energies.json
```

1.11.4 Formation Energy Framework

Substitutional doping model

The workflow assumes substitutional doping on a host sublattice. Dopants are identified as all species that are:

- not equal to the host species (`[doping].host_species`), and
- not in the anion list (`[scan].anion_species`)

The set of dopant counts n_i is extracted from each candidate POSCAR.

Formation energy definition

The formation energy is defined as:

$$E_{\text{form}} = E_{\text{doped}} - E_{\text{pristine}} + \sum_i n_i \left(\mu_{\text{host}} - \mu_i\right)$$

where:

- E_{doped} is the relaxed total energy of the doped supercell
- E_{pristine} is the relaxed total energy of the pristine supercell
- μ_{host} is the host chemical potential (per atom)
- μ_i is the dopant chemical potential (per atom)
- n_i is the number of dopant atoms of species i in the supercell

This corresponds to replacing n_i host atoms by n_i dopant atoms for each dopant species i, while keeping the same supercell size.

Reference energies are taken from Step 00 and must be consistent with the supercell size and host species used here.

1.11.5 Method Summary

For each structure folder inside `[structure].outdir`:

1. Load the reference data from:

   ```
   reference_structures/reference_energies.json
   ```

   extracting E_{pristine} and chemical potentials μ_i.

2. Determine which candidates to evaluate:

   a. If `selected_candidates.txt` exists, only those candidates are used.

   b. Otherwise, all `candidate_*/02_relax/POSCAR` files are used.

3. For each selected candidate:

   a. Read the relaxed energy E_{doped} from `candidate_*/02_relax/meta.json`.

   b. Read species counts from `candidate_*/02_relax/POSCAR` and infer dopant counts under the substitutional model.

   c. Evaluate E_{form} using the equation above.

   d. Apply the requested normalization (see below).

   e. Write `candidate_*/04_formation/meta.json`.

4. Write `formation_energies.csv` in the folder, sorted by total formation energy.

1.11.6 Normalization Options

This stage supports three reporting modes controlled by:

```
[formation]
normalize = "total" | "per_dopant" | "per_host"
```

The internal formation energy is always computed as a **total supercell energy** (E_{form} in eV). The reported value can be:

- `total`: report E_{form} in eV (no normalization)
- `per_dopant` (default): report $E_{\text{form}}/N_{\text{dop}}$, where $N_{\text{dop}} = \sum_i n_i$ is the total number of dopant atoms
- `per_host`: report $E_{\text{form}}/N_{\text{atoms}}$, where N_{atoms} is the total number of atoms in the pristine supercell (as stored in the reference JSON)

Note: `per_host` currently uses the total number of atoms in the pristine supercell. If you later want normalization per *host-sublattice* site, that quantity can be stored explicitly in the reference JSON and used here.

1.11.7 Outputs

Per-folder summary

For each structure folder, this stage writes:

```
formation_energies.csv
```

Columns:

- `candidate`: candidate directory name
- `E_doped_eV`: relaxed total energy of the doped candidate
- `E_form_eV_total`: total formation energy in eV
- `E_form_<normalize>`: normalized formation energy (according to config)
- `n_dopant_atoms`: total dopant atoms N_{dop}
- `dopant_counts`: compact dopant count string (e.g. `Sb:2;Zr:1`)

Rows are sorted by `E_form_eV_total` (ascending).

Per-candidate metadata

For each evaluated candidate, this stage writes:

```
candidate_XXX/04_formation/meta.json
```

This file includes:

- full formation-energy definition string from the reference JSON
- E_{doped}, E_{pristine}
- chemical potentials used for the involved species
- inferred dopant counts
- total formation energy and the reported normalized value

1.11.8 Reproducibility and Skipping

If:

```
[formation].skip_if_done = true
```

and `formation_energies.csv` already exists for a folder, that folder is skipped.

Given unchanged relaxed energies, POSCARs, reference JSON, and configuration, this stage is deterministic.

1.11.9 Notes and Limitations

- This stage assumes substitutional doping and uses a simple species-based dopant identification rule (host vs anions vs dopants).
- No charged-defect corrections, finite-size corrections, entropy terms, or competing-phase chemical potential bounds are included.
- The absolute values depend on the reference energies and the chosen bulk phases used to define μ_i.

1.12 7. Database Collection

1.12.1 Implementation

This stage is implemented in:

```
src/dopingflow/collect.py
```

The public entry point is:

```
run_collect(...)
```

1.12.2 Purpose

This stage consolidates results from previous workflow stages into a **single flat CSV** that can be used as a lightweight database for downstream analysis, plotting, and reporting.

The database is written to the workflow root as:

```
results_database.csv
```

Only **filtered / selected candidates** are included (strict policy), so the database represents the subset of structures that passed your selection criteria.

1.12.3 Inputs

This stage uses settings from two sections of `input.toml`:

- `[structure]`: provides the workflow output directory containing composition folders.
- `[database]`: controls skipping/overwriting of the database CSV.

It expects the standard workflow outputs inside each composition folder (if present):

Composition-level (folder-level)

- `metadata.json` (from Step 01 generation)
- `selected_candidates.txt` (from Step 04 filtering) **preferred**
- `ranking_relax_filtered.csv` (from Step 04 filtering) **fallback**
- `ranking_scan.csv` (from Step 02 scanning)
- `bandgap_alignn_summary.csv` (from Step 05 bandgap prediction)
- `formation_energies.csv` (from Step 06 formation energies)

Candidate-level

- `candidate_*/02_relax/meta.json` (from Step 03 relaxation)

1.12.4 Selection Policy

This stage follows a **strict selection rule**: it will never include unfiltered candidates.

Selection priority within each composition folder:

1. If `selected_candidates.txt` exists: use exactly those candidate names (one per line).
2. Else, if `ranking_relax_filtered.csv` exists: include the candidates listed there.
3. Else: the folder is skipped (no candidates are collected).

This ensures that the database represents only the candidates you explicitly kept after filtering.

1.12.5 Method Summary

For each composition folder inside `[structure].outdir`:

1. Determine the candidate list using the selection policy above.
2. Load composition metadata from `metadata.json` (if available).
3. Load per-stage summary tables (if available):
   - scan ranking table
   - filtered ranking table
   - bandgap summary table
   - formation energy table
4. For each selected candidate:
   a. Read relaxed-energy metadata from `candidate_*/02_relax/meta.json` (if available).

    b. Combine composition-level and candidate-level information into one row.

5. Write all rows to:

```
results_database.csv
```

1.12.6 Database Schema

The output database contains the following columns:

Composition-level fields

- `composition_tag`: Folder name of the composition (effective tag used by the generator).
- `requested_index`: Index of the composition in the generator loop (if available).
- `requested_pct_json`: JSON string of requested dopant percentages (if available).
- `effective_pct_json`: JSON string of effective dopant percentages after rounding (if available).
- `rounded_counts_json`: JSON string of rounded integer substitution counts (if available).
- `host_species`: Host species used for substitution (from generation metadata).
- `n_host`: Number of host sites in the supercell (from generation metadata).
- `supercell_json`: JSON string of the supercell dimensions (if available).

Candidate-level identity

- `candidate`: Candidate folder name (e.g., `candidate_001`).
- `candidate_path`: Absolute path to the candidate folder (useful for linking back to files).

Relax (filtered) fields

- `rank_relax_filtered`: Rank within the filtered set (if available).
- `E_relaxed_eV_filtered`: Relaxed energy read from the filtered ranking table (if available).

Scan fields

- `rank_scan`: Single-point rank from scanning stage (if available).
- `E_scan_eV`: Single-point energy from scanning stage (if available).

Relax (meta) fields

- `E_relaxed_eV`: Relaxed energy extracted from `candidate_*/02_relax/meta.json` (if available).

Bandgap fields

- `bandgap_eV`: Predicted bandgap value from the bandgap summary (if available).

Formation-energy fields

- `E_form_eV_total`: Total formation energy in eV (if available).
- `E_form_norm`: Normalized formation energy (as written by Step 06, depends on your normalization mode).
- `n_dopant_atoms`: Total dopant atoms used in the candidate (if available).
- `dopant_counts`: Compact dopant-count string from Step 06 (if available).

1.12.7 Outputs

This stage writes one file in the workflow root:

```
results_database.csv
```

Each row corresponds to **one selected candidate** in **one composition folder**.

1.12.8 Reproducibility and Skipping

If:

```
[database].skip_if_done = true
```

and `results_database.csv` already exists, the stage is skipped.

Set `skip_if_done = false` to overwrite and regenerate the database.

1.12.9 Notes and Limitations

- The database is intentionally flat and file-based; it is designed for quick use in pandas, spreadsheets, or plotting scripts.
- Missing upstream files are handled gracefully: columns are left empty/`None` when a stage output is unavailable.
- Only the filtered/selected candidate subset is included by design. If you later want a "full database" including all candidates, the selection policy can be relaxed as an optional mode.

1.13 Explicit Example — Single Target Composition

1.13.1 Purpose

This example generates and evaluates **one explicit co-doped composition**.

Use this mode when: - You already know the exact dopant percentages. - You want exactly one generated structure per composition. - You want the full workflow executed (00–07).

1.13.2 Workflow

Run the complete pipeline:

```
dopingflow run-all -c input.toml
```

1.13.3 Required Files

The working directory must contain:

```
input.toml
reference_structures/
    base.POSCAR
    Sn.POSCAR
    Sb.POSCAR
    Zr.POSCAR
```

- `base.POSCAR`: pristine unit cell.
- Elemental `*.POSCAR`: bulk reference structures for chemical potentials.

1.13.4 Example input.toml

```toml
[structure]
base_poscar = "reference_structures/base.POSCAR"
supercell = [5, 2, 1]
outdir = "random_structures"

[doping]
mode = "explicit"
host_species = "Sn"
compositions = [
  { Sb = 5.0, Zr = 5.0 }
]

[generate]
poscar_order = ["Zr","Sb","Sn","O"]
seed_base = 12345
clean_outdir = true

[scan]
topk = 15
symprec = 1e-3
max_enum = 50000000
nproc = 12
chunksize = 50
anion_species = ["O"]
max_unique = 200000
skip_if_done = true

[relax]
fmax = 0.05
n_workers = 6
tf_threads = 1
omp_threads = 1
skip_if_done = true
skip_candidate_if_done = true

[filter]
mode = "window"
window_meV = 50.0
max_candidates = 12
skip_if_done = true

[bandgap]
skip_if_done = true
cutoff = 8.0
max_neighbors = 12

[formation]
skip_if_done = true
normalize = "per_dopant"

[references]
```

(continues on next page)

(continued from previous page)

```
source = "local"
bulk_dir = "reference_structures/"
pristine_poscar = "reference_structures/base.POSCAR"
fmax = 0.02
skip_if_done = true
```

1.13.5 Outputs

- `reference_structures/reference_energies.json`
- `random_structures/<composition>/` - `ranking_scan.csv` - `ranking_relax.csv` - `ranking_relax_filtered.csv` - `bandgap_alignn_summary.csv` - `formation_energies.csv`
- `results_database.csv`

1.14 Explicit Example — Multiple Target Compositions

1.14.1 Purpose

Generate several explicitly defined compositions in one run.

Each composition produces its own folder under `random_structures/`.

1.14.2 Run

```
dopingflow run-all -c input.toml
```

1.14.3 Example input.toml

```
[structure]
base_poscar = "reference_structures/base.POSCAR"
supercell = [5, 2, 1]
outdir = "random_structures"

[doping]
mode = "explicit"
host_species = "Sn"
compositions = [
  { Sb = 5.0, Ti = 5.0 },
  { Sb = 5.0, Zr = 5.0 },
  { Sb = 10.0, Nb = 5.0 }
]

[generate]
poscar_order = ["Ti","Zr","Nb","Sb","Sn","O"]
seed_base = 2026
clean_outdir = true

# Other sections identical to previous example
```

1.14.4 Result

Each composition gets its own directory:

```
random_structures/
    Sb5_Ti5/
    Sb5_Zr5/
    Sb10_Nb5/
```

Each folder contains full scan, relax, filter, bandgap and formation results.

1.15 Enumerate Example — Systematic Dopant Screening

1.15.1 Purpose

Screen multiple dopants and concentrations automatically.

Use this mode when: - You want combinatorial exploration. - You define allowed dopants and doping levels. - You limit the maximum number of dopants per structure.

1.15.2 Note

The current implementation supports **up to 3 dopants per structure** via:

```
max_dopants_total = 3
```

1.15.3 Run

```
dopingflow run-all -c input.toml
```

1.15.4 Example input.toml

```
[structure]
base_poscar = "reference_structures/base.POSCAR"
supercell = [5, 2, 1]
outdir = "random_structures"

[doping]
mode = "enumerate"
host_species = "Sn"
dopants = ["Sb", "Ti", "Zr", "Nb"]
must_include = ["Sb"]
max_dopants_total = 3
allowed_totals = [5.0, 10.0, 15.0]
levels = [5.0, 10.0, 15.0]

[generate]
poscar_order = ["Ti","Zr","Nb","Sb","Sn","O"]
seed_base = 12345
clean_outdir = true

# other sections identical to previous examples
```

1.15.5 Result

The workflow automatically generates composition folders like:

```
random_structures/
    Sb5/
    Sb5_Ti5/
    Sb5_Ti5_Zr5/
    Sb10_Nb5/
    ...
```

Each is processed independently through all stages.

1.16 Smoke Test — Minimal Fast Run

1.16.1 Purpose

Quickly test installation and pipeline functionality with reduced computational cost.

This example: - Uses smaller supercell - Uses small topk - Uses top-N filtering - Can stop early

Run only steps 00–04:

```
dopingflow run-all -c input.toml --start 0 --stop 4
```

1.16.2 Example input.toml

```
[structure]
base_poscar = "reference_structures/base.POSCAR"
supercell = [3, 1, 1]
outdir = "random_structures"

[doping]
mode = "explicit"
host_species = "Sn"
compositions = [
  { Sb = 5.0, Ti = 5.0 },
  { Sb = 5.0, Zr = 5.0 }
]

[generate]
poscar_order = ["Ti","Zr","Sb","Sn","O"]
seed_base = 1
clean_outdir = true

[scan]
topk = 5
nproc = 4
anion_species = ["O"]
skip_if_done = false

[relax]
fmax = 0.08
n_workers = 2
```

(continues on next page)

(continued from previous page)

```
skip_if_done = false
skip_candidate_if_done = false

[filter]
mode = "topn"
max_candidates = 3
skip_if_done = false

[references]
source = "local"
bulk_dir = "reference_structures/"
pristine_poscar = "reference_structures/base.POSCAR"
fmax = 0.03
skip_if_done = false
```

1.16.3 Expected Result

You should obtain:

```
random_structures/<composition>/
    ranking_scan.csv
    ranking_relax.csv
    ranking_relax_filtered.csv
```

This confirms: - generation works - scan works - relaxation works - filtering works

1.17 dopingflow

1.17.1 dopingflow package

Submodules

dopingflow.bandgap module

class dopingflow.bandgap.**BandgapConfig**(*outdir: 'Path', skip_if_done: 'bool', cutoff: 'float', max_neighbors: 'int'*)

Bases: `object`

cutoff: float

max_neighbors: int

outdir: Path

skip_if_done: bool

dopingflow.bandgap.**run_bandgap**(*raw_cfg: dict[str, Any], root: Path, * (Keyword-only parameters separator (PEP 3102)), config_path: Path | None = None*) → None

Step 05: Predict bandgap (ALIGNN local model) for relaxed candidates.

Selection:

1) If selected_candidates.txt exists in a composition folder -> use it
2) Else fallback to candidate_*/02_relax/POSCAR

Outputs per composition folder:

- bandgap_alignn_summary.csv
- candidate_*/03_band/meta.json

`dopingflow.bandgap.`**`run_bandgap_from_toml`**`(`*config_path: Path*`)` → None

dopingflow.cli module

`dopingflow.cli.`**`bandgap_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 05: Predict bandgap for filtered relaxed candidates (ALIGNN).

`dopingflow.cli.`**`collect_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 07: Collect selected candidates into one CSV database.

`dopingflow.cli.`**`filter_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *only: str = <typer.models.OptionInfo object>*, *force: bool = <typer.models.OptionInfo object>*, *window_meV: float = <typer.models.OptionInfo object>*, *topn: int = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 04: Filter relaxed candidates (window or top-N).

`dopingflow.cli.`**`formation_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 06: Compute formation energies using cached references.

`dopingflow.cli.`**`generate_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 01: Generate random doped structures.

`dopingflow.cli.`**`refs_build_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 00: Build/cache reference energies.

`dopingflow.cli.`**`relax_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 03: Relax scanned candidates with M3GNet Relaxer.

`dopingflow.cli.`**`run_all_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *start: str = <typer.models.OptionInfo object>*, *stop: str = <typer.models.OptionInfo object>*, *only: str | None = <typer.models.OptionInfo object>*, *dry_run: bool = <typer.models.OptionInfo object>*, *filter_only: str | None = <typer.models.OptionInfo object>*, *force: bool = <typer.models.OptionInfo object>*, *window_meV: float | None = <typer.models.OptionInfo object>*, *topn: int | None = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Run the full pipeline in order, with optional step selection.

Step keys:
refs -> generate -> scan -> relax -> filter -> bandgap -> formation -> collect

`dopingflow.cli.`**`scan_cmd`**`(`*config: Path = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Step 02: Symmetry-unique scan + M3GNet single-point energies (top-k).

dopingflow.collect module

class `dopingflow.collect.`**`DBConfig`**(*outdir: 'Path'*, *skip_if_done: 'bool'*)

Bases: `object`

`outdir: Path`

`skip_if_done: bool`

`dopingflow.collect.`**`read_bandgap_summary`**(*path: Path*) → Dict[str, Dict[str, Any]]

`dopingflow.collect.`**`read_filtered_table`**(*path: Path*) → Dict[str, Dict[str, Any]]

Parse ranking_relax_filtered.csv (fallback) into:
candidate -> {“rank_relax_filtered”: int, “E_relaxed_eV_filtered”: float}

`dopingflow.collect.`**`read_formation_csv`**(*path: Path*) → Dict[str, Dict[str, Any]]

`dopingflow.collect.`**`read_json`**(*path: Path*) → dict | None

`dopingflow.collect.`**`read_scan_ranking`**(*path: Path*) → Dict[str, Dict[str, Any]]

`dopingflow.collect.`**`read_selected_txt`**(*path: Path*) → List[str]

`dopingflow.collect.`**`run_collect`**(*raw_cfg: dict[str, Any]*, *root: Path*, ***, *config_path: Path | None = None*) → Path

Step 07: Collect results into ONE flat CSV database (results_database.csv), ONLY for the filtered/selected candidates (Step 04 output).

Selection priority:

1) selected_candidates.txt
2) ranking_relax_filtered.csv

`dopingflow.collect.`**`run_collect_from_toml`**(*config_path: Path*) → Path

`dopingflow.collect.`**`safe_get`**(*d: dict | None*, **keys*, *default=None*)

dopingflow.filtering module

class `dopingflow.filtering.`**`FilterConfig`**(*outdir: 'Path'*, *mode: 'str'*, *window_meV: 'float'*, *max_candidates: 'int'*, *skip_if_done: 'bool'*)

Bases: `object`

`max_candidates: int`

`mode: str`

`outdir: Path`

`skip_if_done: bool`

`window_meV: float`

`dopingflow.filtering.`**`run_filtering`**(*raw_cfg: dict[str, Any]*, *root: Path*, ***, *only: str | None = None*, *force: bool = False*, *window_meV: float | None = None*, *topn: int | None = None*) → None

Step 04: filter relaxed candidates after Step 03.

For each composition folder in [structure].outdir:
reads: ranking_relax.csv writes: ranking_relax_filtered.csv, selected_candidates.txt

Filtering:

- [filter].mode="window": keep candidates within window_meV above Emin
- [filter].mode="topn": keep lowest-energy max_candidates

`dopingflow.filtering.`**`run_filtering_from_toml`**(*config_path: Path*, ***, *only: str | None = None*, *force: bool = False*, *window_meV: float | None = None*, *topn: int | None = None*) → None

dopingflow.formation module

class `dopingflow.formation.`**`FormationConfig`**(*outdir: 'Path'*, *host_species: 'str'*, *anion_species: 'List[str]'*, *skip_if_done: 'bool'*, *normalize: 'str'*)

Bases: `object`

`anion_species: List[str]`

`host_species: str`

`normalize: str`

`outdir: Path`

`skip_if_done: bool`

`dopingflow.formation.`**`run_formation`**(*raw_cfg: dict[str, Any]*, *root: Path*, ***, *config_path: Path | None = None*) → None

Step 06: Compute formation energies for relaxed (and optionally filtered) candidates.

Reads:

- E_doped from candidate_*/02_relax/meta.json (energy_relaxed_eV)
- mu, E_pristine from reference_structures/reference_energies.json

Writes per composition folder:

- formation_energies.csv
- candidate_*/04_formation/meta.json

`dopingflow.formation.`**`run_formation_from_toml`**(*config_path: Path*) → None

dopingflow.generate module

class `dopingflow.generate.`**`GenerateConfig`**(*base_poscar: 'str'*, *supercell: 'Tuple[int, int, int]'*, *outdir: 'str'*, *poscar_order: 'List[str]'*, *seed_base: 'int'*, *clean_outdir: 'bool'*, *mode: 'str'*, *host_species: 'str'*, *compositions: 'List[Dict[str, float]]'*, *dopants: 'List[str]'*, *must_include: 'List[str]'*, *max_dopants_total: 'int'*, *allowed_totals: 'List[float]'*, *levels: 'List[float]'*)

Bases: `object`

`allowed_totals: List[float]`

`base_poscar: str`

`clean_outdir: bool`

`compositions: List[Dict[str, float]]`

`dopants: List[str]`

`host_species: str`

`levels: List[float]`

`max_dopants_total: int`

`mode: str`

`must_include: List[str]`

`outdir: str`

`poscar_order: List[str]`

`seed_base: int`

`supercell: Tuple[int, int, int]`

`dopingflow.generate.`**`build_structure_from_counts`**(*pristine: Structure*, *host_species: str*, *dopant_counts: Dict[str, int]*, *seed: int*) → Structure

Randomly substitute host_species sites with dopants according to dopant_counts. Deterministic given seed + tag.

`dopingflow.generate.`**`composition_tag`**(*effective_pct: Dict[str, float]*, *must_first: List[str] | None = None*) → str

Tag used as folder name. Example: Sb5_Ba5 (numbers are integers if near-int).

`dopingflow.generate.`**`enumerate_compositions`**(*dopants: List[str]*, *must_include: List[str]*, *max_dopants_total: int*, *allowed_totals: List[float]*, *levels: List[float]*) → List[Dict[str, float]]

Generate composition dictionaries according to rules. (Same intent as your original script.)

`dopingflow.generate.`**`normalize_to_counts_and_effective`**(*n_host: int*, *requested_pct: Dict[str, float]*) → tuple[Dict[str, int], Dict[str, float], List[str], float, float]

Convert requested dopant percentages (relative to host sites) into integer dopant counts by rounding to the nearest integer number of substitutions.

Parameters

- **`n_host`** (`int`) – Number of host sites available for substitution.
- **`requested_pct`** (`dict[str, float]`) – Dopant element -> requested percent (relative to host sites).

Returns

- **counts** (*dict[str, int]*) – Dopant element -> integer dopant count after rounding.
- **effective_pct** (*dict[str, float]*) – Dopant element -> effective percent after rounding.
- **warnings** (*list[str]*) – Human-readable warnings about rounding deviations.
- **requested_total_pct** (*float*) – Sum of requested dopant percentages.
- **effective_total_pct** (*float*) – Sum of effective dopant percentages after rounding.

Raises
`ValueError` – If requested total exceeds 100% or rounded dopant atoms exceed host sites.

`dopingflow.generate.`**`reorder_structure_by_species`**`(`*s: Structure*, *order: List[str]*`)` → Structure

Reorder sites so POSCAR species order matches user preference. If order is empty, keep pymatgen default ordering.

`dopingflow.generate.`**`run_generate`**`(`*raw_cfg: dict[str, Any]*, *root: Path*, *, *config_path: Path | None = None*`)` → Path

Generate one random doped POSCAR per composition.

Output:
<outdir>/<tag>/POSCAR <outdir>/<tag>/metadata.json

Returns
outdir path

`dopingflow.generate.`**`run_generate_from_toml`**`(`*config_path: Path*`)` → Path

`dopingflow.generate.`**`stable_seed_from_tag`**`(`*tag: str*, *seed_base: int*`)` → int

`dopingflow.generate.`**`validate_composition_minimal`**`(`*requested_pct: Dict[str, float]*`)` → None

dopingflow.logging module

`dopingflow.logging.`**`setup_logging`**`(`*root: Path*, *, *verbose: bool = False*`)` → Path

Configure logging for dopingflow.

- Console output
- Per-run log file
- Noise suppression

dopingflow.refs module

`class dopingflow.refs.`**`RefConfig`**`(`*supercell: 'tuple[int, int, int]'*, *host_species: 'str'*, *pristine_poscar: 'str'*, *source: 'str'*, *bulk_dir: 'Path'*, *mp_ids: 'Dict[str, str]'*, *fmax: 'float'*, *skip_if_done: 'bool'*`)`

Bases: `object`

`bulk_dir: Path`

`fmax: float`

`host_species: str`

`mp_ids: Dict[str, str]`

`pristine_poscar: str`

`skip_if_done: bool`

`source: str`

`supercell: tuple[int, int, int]`

`dopingflow.refs.`**`run_refs_build`**`(`*raw_cfg: dict[str, Any]*, *root: Path*, ***, *config_path: Path | None = None*`)` → Path

Build/cache reference energies needed for formation energy calculations.

Writes: references/reference_energies.json Returns: path to written JSON

`dopingflow.refs.`**`run_refs_build_from_toml`**`(`*config_path: Path*`)` → Path

Convenience wrapper used by the CLI: reads input.toml -> raw dict, sets root=config parent, calls run_refs_build.

dopingflow.relax module

`class dopingflow.relax.`**`RelaxConfig`**`(`*fmax: 'float'*, *n_workers: 'int'*, *tf_threads: 'int'*, *omp_threads: 'int'*, *order: 'List[str]'*, *outdir: 'Path'*, *skip_if_done: 'bool'*, *skip_candidate_if_done: 'bool'*`)`

Bases: `object`

`fmax:` `float`

`n_workers:` `int`

`omp_threads:` `int`

`order:` `List[str]`

`outdir:` `Path`

`skip_candidate_if_done:` `bool`

`skip_if_done:` `bool`

`tf_threads:` `int`

`dopingflow.relax.`**`run_relax`**`(`*raw_cfg: dict[str, Any]*, *root: Path*, ***, *config_path: Path | None = None*`)` → None

Step 03: Relax candidates produced by Step 02.

For each structure folder in [structure].outdir:

- relax candidate_*/01_scan/POSCAR in parallel
- write candidate_*/02_relax/POSCAR and meta.json
- write ranking_relax.csv per structure folder

`dopingflow.relax.`**`run_relax_from_toml`**`(`*config_path: Path*`)` → None

dopingflow.scan module

`class dopingflow.scan.`**`ScanConfig`**`(`*poscar_in: 'str'*, *topk: 'int'*, *symprec: 'float'*, *max_enum: 'int'*, *nproc: 'int'*, *chunksize: 'int'*, *order: 'List[str]'*, *anion_species: 'List[str]'*, *host_species: 'str'*, *max_unique: 'int'*, *skip_if_done: 'bool'*`)`

Bases: `object`

`anion_species:` `List[str]`

`chunksize:` `int`

`host_species:` `str`

`max_enum:` `int`

max_unique: int

nproc: int

order: List[str]

poscar_in: str

skip_if_done: bool

symprec: float

topk: int

`dopingflow.scan.`**`run_scan`**(*raw_cfg: dict[str, Any]*, *root: Path*, ***, *config_path: Path | None = None*) → None

Step 02: For each structure folder in [structure].outdir:

- enumerate symmetry-unique dopant permutations on the cation sublattice
- evaluate single-point energy using M3GNet in parallel
- keep top-k lowest energies

`dopingflow.scan.`**`run_scan_from_toml`**(*config_path: Path*) → None

Module contents

PYTHON MODULE INDEX

INDEX