average-minimum-distance

Daniel Widdowson

Aug 09, 2022

CONTENTS

Below is the readme for amd, explaining what it is and how to use it. More detailed documentation can be found in the sidebar.

amd.compare() is one function for reading crystals and comparing them by their AMD or PDD. For example, to compare all crystals in a cif by PDD with k = 100:

```
import amd
df = amd.compare('crystals.cif', by='PDD', k=100)
```

A pandas DataFrame is returned, a table of the distance matrix with names with rows and columns indexed by name. The function can also take a second path to a cif, and compare all crystals in one with all in the other. To compare by AMD, just change `by='AMD'`.

If `csd-python-api` is installed, the compare function can also accept CSD refcodes instead of cif files.

CHAPTER

ONE

COMPARE OPTIONS

The compare function reads crystals, computes their invariants and compares them in one function for convinience. It accepts many keyword arguments for reading, calculating and comparing, all listed in one place below for convinience.

1.1 Reader options

- `reader` (default `ase`) controls the backend package used to parse the file. To use csd-python-api change to `ccdc`. The ccdc reader should be able to read any format accepted by `ccdc.io.EntryReader`, though only .cifs have been tested.
- `remove_hydrogens` (default `False`) removes Hydrogen atoms from the structure.
- `disorder` (default `skip`) controls how disordered structures are handled. The default skips any crystal with disorder, since disorder conflicts with the periodic set model. Alternatively, `ordered_sites` removes sites with disorder and `all_sites` includes all sites regardless.
- `heaviest_component` (default `False`, `csd-python-api` only) removes all but the heaviest molecule in the asymmetric unit, intended for removing solvents.
- `show_warnings` (default `True`) chooses whether to print warnings during reading, e.g. from disordered structures or crystals with missing data.
- `families` (default `False`, `csd-python-api` only) chooses whether to read refcodes or refcode families.

1.2 PDD options

- `collapse` (default `True`) chooses whether to collpase rows of PDDs which are similar enough (elementwise).
- `collapse_tol` (default `1e-4`) is the tolerance for collapsing PDD rows into one. The merged row is the average of those collapsed.

1.3 Comparison options

- `metric` (default `chebyshev`) chooses the metric used to compare AMDs or PDD rows. See SciPy's cdist/pdist for a list of accepted metrics.
- `n_jobs` (new in 1.2.3, default `None`) is the number of cores to use for multiprocessing (passed to `joblib.Parallel`). Pass -1 to use the maximum.
- `verbose` (changed in 1.2.3, default 0) controls the verbosity level, increasing with larger numbers. This is passed to `joblib.Parallel`, see their documentation for details.

- `low_memory` (default `False`, by='AMD' only) uses an alternative slower algorithm that keeps memory use low for much larger inputs. Only `metric='chebyshev'` is accepted with `low_memory`.

CHAPTER

TWO

DESCRIPTION OF AMD/PDD

The AMD of a crystal is an infinite sequence calculated from inter-atomic distances in the crystal. In contrast, the PDD is a matrix which can have arbitrarily many columns. In practice, both are calculated up to some chosen number k of entries/columns.

The kth AMD value of a periodic set is the average distance to the kth nearest neighbour over atoms in a unit cell. That is, to find the AMD for a periodic set up to k, list (in order) distances to the nearest k neighbours (in the infinite crystal) for every atom in a unit cell take the average, giving a vector length k.

The PDD is related to AMD but contains more information as it avoids the averaging step. Like AMD, list distances to the nearest k neighbours in order for each atom in a unit cell. Collect these lists into one matrix with a row for each atom. Then order the rows of the matrix lexicographically. If any rows are not unique, keep only one and give each a weight proportional to how many copies there are. The result is the kth PDD of the periodic set. In practice, the weights are kept in the first column of the matrix.

A much more detailed description can be found in the papers on AMD and PDD:

- Average minimum distances of periodic point sets - foundational invariants for mapping periodic crystals. MATCH Communications in Mathematical and in Computer Chemistry, 87(3):529-559 (2022). https://doi.org/10.46793/match.87-3.529W
- Pointwise distance distributions of periodic point sets. arXiv preprint arXiv:2108.04798 (2021). https://arxiv.org/abs/2108.04798

2.1 Comparing by AMD/PDD

AMDs are just vectors which can be compared with any metric, as long as k (length of the AMD) is the same. The default metric used in this package is L-infinity (aka Chebyshev), since it does not so much accumulate differences in distances across many neighbours. PDDs are matrices with weighted rows; the appropriate metric to compare them is the *Earth mover's distance* (aka Wasserstein metric), which itself needs a metric to compare two PDD rows (without their weights), where L-infinity is again our default.

CHAPTER

THREE

READING CIFS

If you have a .cif file, use `amd.CifReader` to extract the crystals:

```
# create list of crystals in a .cif
crystals = list(amd.CifReader('file.cif'))

# Can also accept path to a directory, reading all files inside
crystals = list(amd.CifReader('path/to/folder'))

# loop over the reader and get AMDs (k=100) of crystals
amds = []
for p_set in amd.CifReader('file.cif'):
    amds.append(amd.AMD(p_set, 100))
```

The `CifReader` returns *periodicset.PeriodicSet* objects representing the crystals, which can be passed to `amd.AMD()` or `amd.PDD()` to calculate their invariants. The `PeriodicSet` has attributes `.name`, `.motif`, `.cell`, `.types` (atomic numbers), as well as `.asymmetric_unit` and `.wyckoff_multiplicities` for use in calculations. When the path is to a folder, it will also have an attribute `.filename`.

`CifReader` can accept other file formats, if you have csd-python-api installed. This should work if you pass `reader='ccdc'` to the reader, though formats other than .cif have not been tested.

3.1 Reading options

amd.io.CifReader accepts the following parameters (many shared by *io.CSDReader*):

```
amd.CifReader(
    'file.cif',                 # path to file or folder
    reader='ase',               # backend cif parser
    remove_hydrogens=False,     # remove Hydrogens
    disorder='skip',            # handling disorder
    heaviest_component=False,   # just keep the heaviest component in asym unit
    show_warnings=True          # silence warnings
)
```

- `reader` controls the backend package used to parse the file. The default is `ase`; to use csd-python-api change to `ccdc`. The ccdc reader should be able to read any format accepted by `ccdc.io.EntryReader`, though only .cifs have been tested.
- `remove_hydrogens` removes Hydrogen atoms from the structure.

- `disorder` controls how disordered structures are handled. The default is to `skip` any crystal with disorder, since disorder conflicts with the periodic set model. To read disordered structures anyway, choose either `ordered_sites` to remove sites with disorder or `all_sites` include all sites regardless.
- `heaviest_component` csd-python-api only. Removes all but the heaviest molecule in the asymmetric unit, intended for removing solvents.
- `show_warnings` will not print warnings during reading if False, e.g. from disordered structures or crystals with missing data.

See the references *io.CifReader* or *periodicset.PeriodicSet* for more.

CHAPTER

FOUR

READING FROM THE CSD

If csd-python-api is installed, `amd` can use it to read crystals directly from the CSD.

amd.io.CSDReader accepts a list of CSD refcode(s) and yields the crystals. If `None` or `'CSD'` are passed instead of refcodes, it reads the whole CSD:

```
# Put crystals with these refcodes in a list
refcodes = ['DEBXIT01', 'DEBXIT05', 'HXACAN01']
structures = list(amd.CSDReader(refcodes))

# Read refcode families (any whose refcode starts with strings in the list)
refcodes = ['ACSALA', 'HXACAN']
structures = list(amd.CSDReader(refcodes, families=True))

# Create a generic reader, read crystals 'on demand' with CSDReader.entry()
reader = amd.CSDReader()
debxit01 = reader.entry('DEBXIT01')

# looping over this generic reader will yield all CSD entries
for periodic_set in reader:
    ...

# Make list of AMDs for crystals in these families
refcodes = ['ACSALA', 'HXACAN']
amds = []
for structure in amd.CSDReader(refcodes, families=True):
    amds.append(amd.AMD(structure, 100))
```

The `CSDReader` returns *periodicset.PeriodicSet* objects representing the crystals, which can be passed to `amd.AMD()` or `amd.PDD()` to calculate their invariants. The `PeriodicSet` has attributes `.name`, `.motif`, `.cell`, `.types` (atomic numbers), as well as `.asymmetric_unit` and `.wyckoff_multiplicities` for use in calculations.

4.1 Reading options

The `amd.io.CSDReader` accepts the following parameters (many shared by *io.CifReader*):

```
amd.CSDReader(
    refcodes=None,              # list of refcodes (or families) or 'CSD'
    families=False,             # interpret refcodes as families (include if refcode␣
↪starts with)
    remove_hydrogens=False,     # remove Hydrogens
```

(continues on next page)

(continued from previous page)

```
    disorder='skip',            # handling disorder
    heaviest_component=False    # just keep the heaviest component in asym unit
)
```

- As described above, `families` chooses whether to read refcodes or refcode families.
- `remove_hydrogens` removes Hydrogen atoms from the structure.
- `disorder` controls how disordered structures are handled. The default is to `skip` any crystal with disorder, since disorder conflicts with the periodic set model. To read disordered structures anyway, choose either `ordered_sites` to remove sites with disorder or `all_sites` include all sites regardless.
- `heaviest_component` takes the heaviest connected molecule in the motif, intended for removing solvents.

See the references *amd.io.CSDReader* or *amd.periodicset.PeriodicSet* for more.

CHAPTER

FIVE

USING AMDS

5.1 Calculation

The average minimum distance (AMD) of a crystal is given by `amd.AMD()`. It accepts a crystal and an integer k, returning AMD_k as a 1D NumPy array.

If you have a .cif file, use *amd.io.CifReader* to read the crystals (see *Reading cifs*). If you have CSD refcodes and `csd-python-api` is installed, use *amd.io.CSDReader* (see *Reading from the CSD*).

```
# get AMDs of crystals in a .cif
crystals = list(amd.CifReader('file.cif'))
amds = [amd.AMD(crystal, 100) for crystal in crystals]

# get AMDs of crystals in DEBXIT family
csd_reader = amd.CSDReader('DEBXIT', families=True)
amds = [amd.AMD(crystal, 100) for crystal in csd_reader]
```

You can also give the coordinates of motif points and unit cell as a tuple of numpy arrays, in Cartesian form:

```
# AMD (k=10) of 3D cubic lattice
motif = np.array([[0,0,0]])
cell = np.identity(3)
cubic_lattice = (motif, cell)
cubic_amd = amd.AMD(cubic_lattice, 10)
```

Each AMD returned by `amd.AMD(crystal, k)` is a vector length `k`.

Note: The AMD of a crystal can be calculated from its PDD. If both are needed, you can calculate the PDD and then use `amd.calculate.PDD_to_AMD()`.

5.2 Comparison

AMDs are just vectors that can be compared with any metric, but the *amd.compare* module has functions to compare AMDs for you.

compare.AMD_pdist() and *compare.AMD_cdist()* are like scipy's functions `pdist` and `cdist`. `pdist` takes a set and compares all elements pairwise, whereas `cdist` takes two sets and compares elements in one with the other. `cdist` returns a 2D distance matrix, but `pdist` returns a condensed distance matrix (see scipy's pdist). The default metric for AMD comparisons is l-infinity, but it can be changed to any metric accepted by scipy's pdist/cdist.

```
# compare crystals in file1.cif with those in file2.cif by AMD, k=100
amds1 = [amd.AMD(crystal, 100) for crystal in amd.CifReader('file1.cif')]
amds2 = [amd.AMD(crystal, 100) for crystal in amd.CifReader('file2.cif')]
distance_matrix = amd.AMD_cdist(amds1, amds2)

# compare everything in file1.cif with each other (using l-inf)
condensed_dm = amd.AMD_pdist(amds1)
```

5.2.1 Comparison options

`amd.AMD_cdist` and `amd.AMD_pdist` share the following optional arguments:

- `metric` chooses the metric used for comparison, see scipy's cdist/pdist for a list of accepted metrics.
- `low_memory` (default `False`) uses an alternative slower algorithm that keeps memory use low for much larger inputs. Currently only `metric='chebyshev'` is accepted with `low_memory`.

CHAPTER

SIX

USING PDDS

6.1 Calculation

The pointwise distance distribution (PDD) of a crystal is given by `amd.PDD()`. It accepts a crystal and an integer k, returning the PDD_k as a 2D NumPy array with k+1 columns, the weights of each row being in the first column.

If you have a .cif file, use *amd.io.CifReader* to read the crystals (see *Reading cifs*). If you have CSD refcodes and `csd-python-api` is installed, use *amd.io.CSDReader* (see *Reading from the CSD*).

```
# get PDDs of crystals in a .cif
reader = amd.CifReader('file.cif')
pdds = [amd.PDD(crystal, 100) for crystal in reader]

# get PDDs of DEBXIT01 and DEBXIT02 from the CSD
crystals = list(amd.CSDReader(['DEBXIT01', 'DEBXIT02']))
pdds = [amd.PDD(crystal, 50) for crystal in crystals]
```

You can also give the coordinates of motif points and unit cell as a tuple of numpy arrays, in Cartesian form:

```
# PDD (k=10) of 3D cubic lattice
motif = np.array([[0,0,0]])
cell = np.identity(3)
cubic_lattice = (motif, cell)
cubic_pdd = amd.PDD(cubic_lattice, 10)
```

The object returned by `amd.PDD` is a NumPy array with `k+1` columns.

6.1.1 Calculation options

`amd.PDD` accepts a few optional arguments (not relevant to `amd.AMD`):

```
amd.PDD(periodic_set, k, lexsort=True, collapse=True, collapse_tol=1e-4, return_row_
↪groups=False)
```

`lexsort` lexicograpgically orders the rows of the PDD, and `collapse` merges rows if all elements of rows are within `collapse_tol`. The definition of PDD requires both in order to satisfy invariance (that two isometric sets have equal PDDs). However, earth mover's distance does not depend on the order of rows, so `lexsort=False` makes no difference to comparisons. Setting `collapse=False` will not change the earth mover's distances either, but comparisons can take longer if rows aren't collapsed.

Sometimes it's useful to know which rows of the PDD came from which motif points in the input. Setting `return_row_groups=True` makes the function return a tuple `(pdd, groups)`, where `groups[i]` contains the indices of the point(s) corresponding to `pdd[i]`. Note that these indices are for the asymmetric unit of the set, whose indices in `periodic_set.motif` are accessible through `periodic_set.asymmetric_unit`.

6.2 Comparison

The Earth mover's distance is the appropriate metric to compare PDDs. The *amd.compare* module contains functions for these comparisons.

compare.PDD_pdist() and *compare.PDD_cdist()* are like scipy's functions `pdist` and `cdist`. `pdist` takes one set and compares all elements pairwise, whereas `cdist` takes two sets and compares elements in one with the other. `cdist` returns a 2D distance matrix, but `pdist` returns a condensed distance matrix (see scipy's pdist function).

```
# compare crystals in file1.cif with those in file2.cif by PDD, k=100
pdds1 = [amd.PDD(crystal, 100) for crystal in amd.CifReader('file1.cif')]
pdds2 = [amd.PDD(crystal, 100) for crystal in amd.CifReader('file2.cif')]
distance_matrix = amd.PDD_cdist(pdds1, pdds2)

# compare everything in file1.cif with each other
condensed_dm = amd.PDD_pdist(pdds1)
```

You can compare one PDD with another with *compare.EMD()*:

```
# compare DEBXIT01 and DEBXIT02 by PDD, k=100
pdds = [amd.PDD(crystal, 100) for crystal in amd.CSDReader(['DEBXIT01', 'DEBXIT02'])]
distance = amd.EMD(pdds[0], pdds[1])
```

compare.EMD(), *compare.PDD_pdist()* and *compare.PDD_cdist()* all accept an optional argument `metric`, which can be anything accepted by scipy's pdist/cdist functions. The metric used to compare PDD matrices is always Earth mover's distance, but this still requires another metric between the rows of PDDs (so there's a different Earth mover's distance for each choice of metric).

6.2.1 Comparison options and multiprocessing

`amd.PDD_cdist` and `amd.PDD_pdist` share the following optional arguments:

- `metric` (default `chebyshev`) chooses the metric used to compare PDD rows, as explained above. See scipy's cdist/pdist for a list of accepted metrics.
- `n_jobs` (new in 1.2.3, default `None`) is the number of cores to use for multiprocessing (passed to `joblib.Parallel`). Pass -1 to use the maximum.
- `verbose` (changed in 1.2.3, default 0) controls the verbosity level, increasing with larger numbers. This is passed to `joblib.Parallel`, see their documentation for details.

CHAPTER

SEVEN

AMD.CALCULATE MODULE

Functions for calculating the average minimum distance (AMD) and point-wise distance distribution (PDD) isometric invariants of periodic crystals and finite sets.

amd.calculate.**AMD**(*periodic_set: Union[*amd.periodicset.PeriodicSet*, Tuple[*numpy.ndarray*, numpy.ndarray]]*, *k: int*) → numpy.ndarray

The AMD of a periodic set (crystal) up to k.

Parameters

- **periodic_set** (*periodicset.PeriodicSet* or tuple of numpy.ndarray s) – A periodic set represented by a *periodicset.PeriodicSet* or by a tuple (motif, cell) with coordinates in Cartesian form and a square unit cell.
- **k** (*int*) – Length of the AMD returned; the number of neighbours considered for each atom in the unit cell to make the AMD.

Returns A numpy.ndarray shape (k,), the AMD of `periodic_set` up to k.

Return type numpy.ndarray

Examples

Make list of AMDs with k = 100 for crystals in data.cif:

```
amds = []
for periodic_set in amd.CifReader('data.cif'):
    amds.append(amd.AMD(periodic_set, 100))
```

Make list of AMDs with k = 10 for crystals in these CSD refcode families:

```
amds = []
for periodic_set in amd.CSDReader(['HXACAN', 'ACSALA'], families=True):
    amds.append(amd.AMD(periodic_set, 10))
```

Manually pass a periodic set as a tuple (motif, cell):

```
# simple cubic lattice
motif = np.array([[0,0,0]])
cell = np.array([[1,0,0], [0,1,0], [0,0,1]])
cubic_amd = amd.AMD((motif, cell), 100)
```

amd.calculate.**PDD**(*periodic_set: Union[*amd.periodicset.PeriodicSet*, Tuple[*numpy.ndarray*, numpy.ndarray]]*, *k: int*, *lexsort: bool = True*, *collapse: bool = True*, *collapse_tol: float = 0.0001*, *return_row_groups: bool = False*) → numpy.ndarray

The PDD of a periodic set (crystal) up to k.

Parameters

- **periodic_set** (*periodicset.PperiodicSet* tuple of numpy.ndarray s) – A periodic set represented by a *periodicset.PperiodicSet* or by a tuple (motif, cell) with coordinates in Cartesian form and a square unit cell.
- **k** (*int*) – The returned PDD has k+1 columns, an additional first column for row weights. k is the number of neighbours considered for each atom in the unit cell to make the PDD.
- **lexsort** (*bool, default True*) – Lexicographically order the rows. Default True.
- **collapse** (*bool, default True*) – Collapse repeated rows (within the tolerance `collapse_tol`). Default True.
- **collapse_tol** (*float, default 1e-4*) – If two rows have all elements closer than `collapse_tol`, they are merged and weights are given to rows in proportion to the number of times they appeared. Default is 0.0001.
- **return_row_groups** (*bool, default False*) – Return data about which PDD rows correspond to which points. If True, a tuple is returned `(pdd, groups)` where `groups[i]` contains the indices of the point(s) corresponding to `pdd[i]`. Note that these indices are for the asymmetric unit of the set, whose indices in `periodic_set.motif` are accessible through `periodic_set.asymmetric_unit`.

Returns A numpy.ndarray with k+1 columns, the PDD of `periodic_set` up to k. The first column contains the weights of rows. If `return_row_groups` is True, returns a tuple (numpy.ndarray, list).

Return type numpy.ndarray

Examples

Make list of PDDs with `k=100` for crystals in data.cif:

```
pdds = []
for periodic_set in amd.CifReader('data.cif'):
    # do not lexicographically order rows
    pdds.append(amd.PDD(periodic_set, 100, lexsort=False))
```

Make list of PDDs with `k=10` for crystals in these CSD refcode families:

```
pdds = []
for periodic_set in amd.CSDReader(['HXACAN', 'ACSALA'], families=True):
    # do not collapse rows
    pdds.append(amd.PDD(periodic_set, 10, collapse=False))
```

Manually pass a periodic set as a tuple (motif, cell):

```
# simple cubic lattice
motif = np.array([[0,0,0]])
cell = np.array([[1,0,0], [0,1,0], [0,0,1]])
cubic_amd = amd.PDD((motif, cell), 100)
```

amd.calculate.**PDD_to_AMD**(*pdd: numpy.ndarray*) → numpy.ndarray

Calculates an AMD from a PDD. Faster than computing both from scratch.

Parameters **pdd** (*numpy.ndarray*) – The PDD of a periodic set.

Returns The AMD of the periodic set.

Return type numpy.ndarray

amd.calculate.**AMD_finite**(*motif: numpy.ndarray*) → numpy.ndarray

The AMD of a finite m-point set up to k = m-1.

Parameters **motif** (*numpy.ndarray*) – Coordinates of a set of points.

Returns A vector length m-1 (where m is the number of points), the AMD of `motif`.

Return type numpy.ndarray

Examples

The AMD distance (L-infinity) between finite trapezium and kite point sets:

```
trapezium = np.array([[0,0],[1,1],[3,1],[4,0]])
kite      = np.array([[0,0],[1,1],[1,-1],[4,0]])

trap_amd = amd.AMD_finite(trapezium)
kite_amd = amd.AMD_finite(kite)

l_inf_dist = np.amax(np.abs(trap_amd - kite_amd))
```

amd.calculate.**PDD_finite**(*motif: numpy.ndarray*, *lexsort: bool = True*, *collapse: bool = True*, *collapse_tol: float = 0.0001*, *return_row_groups: bool = False*) → numpy.ndarray

The PDD of a finite m-point set up to k = m-1.

Parameters

- **motif** (*numpy.ndarray*) – Coordinates of a set of points.
- **lexsort** (*bool, default True*) – Whether or not to lexicographically order the rows. Default True.
- **collapse** (*bool, default True*) – Whether or not to collapse repeated rows (within the tolerance `collapse_tol`). Default True.
- **collapse_tol** (*float, default 1e-4*) – If two rows have all elements closer than `collapse_tol`, they are merged and weights are given to rows in proportion to the number of times they appeared. Default is 0.0001.
- **return_row_groups** (*bool, default False*) – Whether to return data about which PDD rows correspond to which points. If True, a tuple is returned `(pdd, groups)` where `groups[i]` contains the indices of the point(s) corresponding to `pdd[i]`.

Returns A `numpy.ndarray` with m columns (where m is the number of points), the PDD of `motif`. The first column contains the weights of rows.

Return type numpy.ndarray

Examples

Find PDD distance between finite trapezium and kite point sets:

```
trapezium = np.array([[0,0],[1,1],[3,1],[4,0]])
kite      = np.array([[0,0],[1,1],[1,-1],[4,0]])

trap_pdd = amd.PDD_finite(trapezium)
kite_pdd = amd.PDD_finite(kite)

dist = amd.EMD(trap_pdd, kite_pdd)
```

amd.calculate.**PDD_reconstructable**(*periodic_set: Union[*amd.periodicset.PeriodicSet*, Tuple[*numpy.ndarray*, numpy.ndarray]]*, *lexsort: bool = True*) → numpy.ndarray

The PDD of a periodic set with k (no of columns) large enough such that the periodic set can be reconstructed from the PDD.

Parameters

- **periodic_set** (*periodicset.PeriodicSet* tuple of numpy.ndarray s) – A periodic set represented by a *periodicset.PeriodicSet* or by a tuple (motif, cell) with coordinates in Cartesian form and a square unit cell.
- **lexsort** (*bool, default True*) – Whether or not to lexicographically order the rows. Default True.

Returns An ndarray, the PDD of `periodic_set` with enough columns to be reconstructable.

Return type numpy.ndarray

amd.calculate.**PPC**(*periodic_set: Union[*amd.periodicset.PeriodicSet*, Tuple[*numpy.ndarray*, numpy.ndarray]]*) → float

The point packing coefficient (PPC) of `periodic_set`.

The PPC is a constant of any periodic set determining the asymptotic behaviour of its AMD and PDD. As $k \to \infty$, the ratio $\text{AMD}_k / \sqrt[n]{k}$ converges to the PPC, as does any row of its PDD.

For a unit cell U and m motif points in n dimensions,

$$\text{PPC} = \sqrt[n]{\frac{\text{Vol}[U]}{mV_n}}$$

where V_n is the volume of a unit sphere in n dimensions.

Parameters **periodic_set** (*periodicset.PeriodicSet* or tuple of) – numpy.ndarray s (motif, cell) representing the periodic set in Cartesian form.

Returns The PPC of `periodic_set`.

Return type float

amd.calculate.**AMD_estimate**(*periodic_set: Union[*amd.periodicset.PeriodicSet*, Tuple[*numpy.ndarray*, numpy.ndarray]]*, *k: int*) → numpy.ndarray

Calculates an estimate of AMD based on the PPC, using the fact that

$$\lim_{k\to\infty} \frac{\text{AMD}_k}{\sqrt[n]{k}} = \sqrt[n]{\frac{\text{Vol}[U]}{mV_n}}$$

where U is the unit cell, m is the number of motif points and V_n is the volume of a unit sphere in n-dimensional space.

CHAPTER

EIGHT

AMD.COMPARE MODULE

Functions for comparing AMDs and PDDs of crystals.

amd.compare.**compare**(*crystals*, *crystals_=None*, *by='AMD'*, *k=100*, ***kwargs*)

Given one or two sets of periodic set(s), refcode(s) or cif(s), compare them returning a DataFrame of the distance matrix. Default is to comapre by PDD with k=100. Accepts most keyword arguments accepted by the CifReader, CSDReader and compare functions, for a full list see the documentation Quick Start page. Note that using refcodes requires csd-python-api.

Parameters

- **crystals** (*array or list of arrays*) – One or a collection of paths, refcodes, file objects or *periodicset.PeriodicSet* s.
- **crystals_** (*array or list of arrays, optional*) – One or a collection of paths, refcodes, file objects or *periodicset.PeriodicSet* s.
- **by** (*str, default 'AMD'*) – Invariant to compare by, either 'AMD' or 'PDD'.
- **k** (*int, default 100*) – k value to use for the invariants (length of AMD, or number of columns in PDD).

Returns **df** – DataFrame of the distance matrix for the given crystals compared by the chosen invariant.

Return type pandas.DataFrame

Raises **ValueError** – If by is not 'AMD' or 'PDD', if either set given have no valid crystals to compare, or if crystals or crystals_ are an invalid type.

Examples

Compare everything in a .cif (deafult, AMD with k=100):

```
df = amd.compare('data.cif')
```

Compare everything in one cif with all crystals in all cifs in a directory (PDD, k=50):

```
df = amd.compare('data.cif', 'dir/to/cifs', by='PDD', k=50)
```

Examples (csd-python-api only)

Compare two crystals by CSD refcode (PDD, k=50):

```
df = amd.compare('DEBXIT01', 'DEBXIT02', by='PDD', k=50)
```

Compare everything in a refcode family (AMD, k=100):

```
df = amd.compare('DEBXIT', families=True)
```

amd.compare.**EMD**(*pdd: numpy.ndarray*, *pdd_: numpy.ndarray*, *metric: Optional[str] = 'chebyshev'*, *return_transport: Optional[bool] = False*, ***kwargs*)

Earth mover's distance (EMD) between two PDDs, also known as the Wasserstein metric.

Parameters

- **pdd** (*numpy.ndarray*) – PDD of a crystal.
- **pdd_** (*numpy.ndarray*) – PDD of a crystal.
- **metric** (*str or callable, default 'chebyshev'*) – EMD between PDDs requires defining a distance between PDD rows. By default, Chebyshev (L-infinity) distance is chosen as with AMDs. Accepts any metric accepted by `scipy.spatial.distance.cdist()`.
- **return_transport** (*bool, default False*) – Return a tuple `(distance, transport_plan)` with the optimal transport.

Returns **emd** – Earth mover's distance between two PDDs.

Return type float

Raises **ValueError** – Thrown if pdd and pdd_ do not have the same number of columns (k value).

amd.compare.**AMD_cdist**(*amds: Union[numpy.ndarray, List[numpy.ndarray]]*, *amds_: Union[numpy.ndarray, List[numpy.ndarray]]*, *metric: str = 'chebyshev'*, *low_memory: bool = False*, ***kwargs*) → numpy.ndarray

Compare two sets of AMDs with each other, returning a distance matrix. This function is essentially identical to `scipy.spatial.distance.cdist()` with the default metric `chebyshev`.

Parameters

- **amds** (*array_like*) – A list of AMDs.
- **amds_** (*array_like*) – A list of AMDs.
- **metric** (*str or callable, default 'chebyshev'*) – Usually AMDs are compared with the Chebyshev (L-infinitys) distance. Can take any metric accepted by `scipy.spatial.distance.cdist()`.
- **low_memory** (*bool, default False*) – Use a slower but more memory efficient method for large collections of AMDs (Chebyshev metric only).

Returns **dm** – A distance matrix shape `(len(amds), len(amds_))`. `dm[ij]` is the distance (given by `metric`) between `amds[i]` and `amds[j]`.

Return type numpy.ndarray

amd.compare.**AMD_pdist**(*amds: Union[numpy.ndarray, List[numpy.ndarray]]*, *metric: str = 'chebyshev'*, *low_memory: bool = False*, ***kwargs*) → numpy.ndarray

Compare a set of AMDs pairwise, returning a condensed distance matrix. This function is essentially identical to `scipy.spatial.distance.pdist()` with the default metric `chebyshev`.

Parameters

- **amds** (*array_like*) – An array/list of AMDs.
- **metric** (*str or callable, default 'chebyshev'*) – Usually AMDs are compared with the Chebyshev (L-infinity) distance. Can take any metric accepted by `scipy.spatial.distance.pdist()`.

- **low_memory** (*bool, default False*) – Optionally use a slightly slower but more memory efficient method for large collections of AMDs (Chebyshev metric only).

Returns Returns a condensed distance matrix. Collapses a square distance matrix into a vector, just keeping the upper half. See `scipy.spatial.distance.squareform()` to convert to a square distance matrix or for more on condensed distance matrices.

Return type numpy.ndarray

amd.compare.**PDD_cdist**(*pdds: List[numpy.ndarray]*, *pdds_: List[numpy.ndarray]*, *metric: str = 'chebyshev'*, *n_jobs=None*, *verbose=0*, ***kwargs*) → numpy.ndarray

Compare two sets of PDDs with each other, returning a distance matrix.

Parameters

- **pdds** (*List[numpy.ndarray]*) – A list of PDDs.
- **pdds_** (*List[numpy.ndarray]*) – A list of PDDs.
- **metric** (*str or callable, default 'chebyshev'*) – Usually PDD rows are compared with the Chebyshev/l-infinity distance. Can take any metric accepted by `scipy.spatial.distance.cdist()`.
- **n_jobs** (*int, default None*) – Maximum number of concurrent jobs for parallel processing with joblib. Set to -1 to use the maximum possible. Note that for small inputs (< 100), using parallel processing may be slower than the default n_jobs=None.
- **verbose** (*int, default 0*) – The verbosity level. Higher = more verbose, see joblib.Parallel.

Returns Returns a distance matrix shape `(len(pdds), len(pdds_))`. The ij th entry is the distance between `pdds[i]` and `pdds_[j]` given by Earth mover's distance.

Return type numpy.ndarray

amd.compare.**PDD_pdist**(*pdds: List[numpy.ndarray]*, *metric: str = 'chebyshev'*, *n_jobs=None*, *verbose=0*, ***kwargs*) → numpy.ndarray

Compare a set of PDDs pairwise, returning a condensed distance matrix.

Parameters

- **pdds** (*List[numpy.ndarray]*) – A list of PDDs.
- **metric** (*str or callable, default 'chebyshev'*) – Usually PDD rows are compared with the Chebyshev/l-infinity distance. Can take any metric accepted by `scipy.spatial.distance.pdist()`.
- **n_jobs** (*int, default None*) – Maximum number of concurrent jobs for parallel processing with joblib. Set to -1 to use the maximum possible. Note that for small inputs (< 100), using parallel processing may be slower than the default n_jobs=None.
- **verbose** (*int, default 0*) – The verbosity level. Higher = more verbose, see joblib.Parallel for more.

Returns Returns a condensed distance matrix. Collapses a square distance matrix into a vector just keeping the upper half. See `scipy.spatial.distance.squareform()` to convert to a square distance matrix or for more on condensed distance matrices.

Return type numpy.ndarray

amd.compare.**emd**(*pdd: numpy.ndarray*, *pdd_: numpy.ndarray*, *metric: Optional[str] = 'chebyshev'*, *return_transport: Optional[bool] = False*, ***kwargs*)

Alias for amd.EMD().

CHAPTER

NINE

AMD.IO MODULE

Tools for reading crystals from files, or from the CSD with `csd-python-api`. The readers return *periodicset.PeriodicSet* objects representing the crystal which can be passed to *calculate.AMD()* and *calculate.PDD()* to get their invariants.

class amd.io.**CifReader**(*path*, *reader='ase'*, *remove_hydrogens=False*, *disorder='skip'*, *heaviest_component=False*, *show_warnings=True*)

Bases: amd.io._Reader

Read all structures in a .cif file or all files in a folder with ase or csd-python-api (if installed), yielding *periodicset.PeriodicSet* s.

Parameters

- **path** (*str*) – Path to a .cif file or directory. (Other files are accepted when using `reader='ccdc'`, if csd-python-api is installed.)
- **reader** (*str, optional*) – The backend package used for parsing. Default is `ase`, to use csd-python-api change to `ccdc`. The ccdc reader should be able to read any format accepted by `ccdc.io.EntryReader`, though only cifs have been tested.
- **remove_hydrogens** (*bool, optional*) – Remove Hydrogens from the crystal.
- **disorder** (*str, optional*) – Controls how disordered structures are handled. Default is `skip` which skips any crystal with disorder, since disorder conflicts with the periodic set model. To read disordered structures anyway, choose either `ordered_sites` to remove sites with disorder or `all_sites` include all sites regardless.
- **heaviest_component** (*bool, optional*) – csd-python-api only. Removes all but the heaviest molecule in the asymmeric unit, intended for removing solvents.
- **show_warnings** (*bool, optional*) – Controls whether warnings that arise during reading are printed.

Yields *periodicset.PeriodicSet* – Represents the crystal as a periodic set, consisting of a finite set of points (motif) and lattice (unit cell). Contains other useful data, e.g. the crystal's name and information about the asymmetric unit for calculation.

Examples

```
# Put all crystals in a .CIF in a list
structures = list(amd.CifReader('mycif.cif'))

# Can also accept path to a directory, reading all files inside
structures = list(amd.CifReader('path/to/folder'))

# Reads just one if the .CIF has just one crystal
periodic_set = amd.CifReader('mycif.cif').read_one()

# List of AMDs (k=100) of crystals in a .CIF
amds = [amd.AMD(periodic_set, 100) for periodic_set in amd.CifReader('mycif.cif')]
```

class amd.io.**CSDReader**(*refcodes=None*, *families=False*, *remove_hydrogens=False*, *disorder='skip'*, *heaviest_component=False*, *show_warnings=True*)

Bases: amd.io._Reader

Read structures from the CSD with csd-python-api, yielding *periodicset.PeriodicSet* s.

Parameters

- **refcodes** (*List[str], optional*) – List of CSD refcodes to read. If None or 'CSD', iterates over the whole CSD.
- **families** (*bool, optional*) – Read all entries whose refcode starts with the given strings, or 'families' (e.g. giving 'DEBXIT' reads all entries starting with DEBXIT).
- **remove_hydrogens** (*bool, optional*) – Remove hydrogens from the crystal.
- **disorder** (*str, optional*) – Controls how disordered structures are handled. Default is `skip` which skips any crystal with disorder, since disorder conflicts with the periodic set model. To read disordered structures anyway, choose either `ordered_sites` to remove sites with disorder or `all_sites` include all sites regardless.
- **heaviest_component** (*bool, optional*) – csd-python-api only. Removes all but the heaviest molecule in the asymmeric unit, intended for removing solvents.
- **show_warnings** (*bool, optional*) – Controls whether warnings that arise during reading are printed.

Yields *periodicset.PeriodicSet* – Represents the crystal as a periodic set, consisting of a finite set of points (motif) and lattice (unit cell). Contains other useful data, e.g. the crystal's name and information about the asymmetric unit for calculation.

Examples

```
# Put these entries in a list
refcodes = ['DEBXIT01', 'DEBXIT05', 'HXACAN01']
structures = list(amd.CSDReader(refcodes))

# Read refcode families (any whose refcode starts with strings in the list)
refcode_families = ['ACSALA', 'HXACAN']
structures = list(amd.CSDReader(refcode_families, families=True))

# Get AMDs (k=100) for crystals in these families
```

(continues on next page)

(continued from previous page)

```
refcodes = ['ACSALA', 'HXACAN']
amds = []
for periodic_set in amd.CSDReader(refcodes, families=True):
    amds.append(amd.AMD(periodic_set, 100))

# Giving the reader nothing reads from the whole CSD.
reader = amd.CSDReader()

# looping over this generic reader will yield all CSD entries
for periodic_set in reader:
    ...

# or, read structures by refcode on demand
debxit01 = reader.entry('DEBXIT01')
```

entry(*refcode: str*, ***kwargs*) → *amd.periodicset.PeriodicSet*

Read a crystal given a CSD refcode, returning a *periodicset.PperiodicSet*. If given kwargs, overrides the kwargs given to the Reader.

family(*refcode_family: str*, ***kwargs*)

amd.io.**entry_to_periodicset**(*entry*, *remove_hydrogens=False*, *disorder='skip'*, *heaviest_component=False*) → *amd.periodicset.PeriodicSet*

ccdc.entry.Entry -> *amd.periodicset.PeriodicSet*. Entry is the type returned by ccdc.io.EntryReader.

Parameters

- **entry** (ccdc.entry.Entry) – A ccdc Entry object representing a database entry.
- **remove_hydrogens** (*bool, optional*) – Remove Hydrogens from the crystal.
- **disorder** (*str, optional*) – Controls how disordered structures are handled. Default is `skip` which skips any crystal with disorder, since disorder conflicts with the periodic set model. To read disordered structures anyway, choose either `ordered_sites` to remove sites with disorder or `all_sites` include all sites regardless.
- **heaviest_component** (*bool, optional*) – Removes all but the heaviest molecule in the asymmeric unit, intended for removing solvents.

Returns Represents the crystal as a periodic set, consisting of a finite set of points (motif) and lattice (unit cell). Contains other useful data, e.g. the crystal's name and information about the asymmetric unit for calculation.

Return type *periodicset.PeriodicSet*

Raises **_ParseError :** – Raised if the structure can/should not be parsed for the following reasons: 1. entry.has_3d_structure is False, 2. disorder == 'skip' and any of: (a) any disorder flag is True, (b) any atom has fractional occupancy, (c) any atom's label ends with '?', 3. entry.crystal.molecule.all_atoms_have_sites is False, 4. a.fractional_coordinates is None for any a in entry.crystal.disordered_molecule, 5. motif is empty after removing H, disordered sites or solvents.

amd.io.**cifblock_to_periodicset**(*block*, *remove_hydrogens=False*, *disorder='skip'*) → *amd.periodicset.PeriodicSet*

ase.io.cif.CIFBlock -> *amd.periodicset.PeriodicSet*. CIFBlock is the type returned by ase.io.cif.parse_cif.

Parameters

- **block** (ase.io.cif.CIFBlock) – An ase CIFBlock object representing a crystal.
- **remove_hydrogens** (*bool, optional*) – Remove Hydrogens from the crystal.
- **disorder** (*str, optional*) – Controls how disordered structures are handled. Default is `skip` which skips any crystal with disorder, since disorder conflicts with the periodic set model. To read disordered structures anyway, choose either `ordered_sites` to remove sites with disorder or `all_sites` include all sites regardless.

Returns Represents the crystal as a periodic set, consisting of a finite set of points (motif) and lattice (unit cell). Contains other useful data, e.g. the crystal's name and information about the asymmetric unit for calculation.

Return type *periodicset.PeriodicSet*

Raises **_ParseError** – Raised if the structure can/should not be parsed for the following reasons: 1. no sites found or motif is empty after removing H or disordered sites, 2. a site has missing coordinates, 3. disorder == 'skip' and any of: (a) any atom has fractional occupancy, (b) any atom's label ends with '?'.

CHAPTER

TEN

AMD.PERIODICSET MODULE

Implements the *PeriodicSet* class representing a periodic set, defined by a motif and unit cell. This models a crystal with a point at the center of each atom.

This is the object type yielded by the readers *io.CifReader* and *io.CSDReader*. The *PeriodicSet* can be passed as the first argument to *calculate.AMD()* or *calculate.PDD()* to calculate its invariants.

class amd.periodicset.**PeriodicSet**(*motif: numpy.ndarray*, *cell: numpy.ndarray*, *name: Optional[str] = None*, *asymmetric_unit: Optional[numpy.ndarray] = None*, *wyckoff_multiplicities: Optional[numpy.ndarray] = None*, *types: Optional[numpy.ndarray] = None*)

Bases: `object`

A periodic set is the mathematical representation of a crystal by putting a single point in the center of every atom. It is defined by a basis (unit cell) and collection of points (motif) which repeats according to the basis.

PeriodicSet s are returned by the readers in the *io* module. Instances of this object can be passed to *calculate.AMD()* or *calculate.PDD()* to calculate the invariant.

Parameters

- **motif** (*numpy.ndarray*) – Cartesian (orthogonal) coordinates of the motif, shape (no points, dims).
- **cell** (*numpy.ndarray*) – Cartesian (orthogonal) square array representing the unit cell, shape (dims, dims). Use *utils.cellpar_to_cell()* to convert 6 cell parameters to an orthogonal square matrix.
- **name** (*str, optional*) – Name of the periodic set.
- **asymmetric_unit** (*numpy.ndarray, optional*) – Indices for the asymmetric unit, pointing to the motif.
- **wyckoff_multiplicities** (*numpy.ndarray, optional*) – Wyckoff multiplicities of each atom in the asymmetric unit (number of unique sites generated under all symmetries).
- **types** (*numpy.ndarray, optional*) – Array of atomic numbers of motif points.

CHAPTER

ELEVEN

AMD.UTILS MODULE

Helpful utility functions, e.g. unit cell diameter, converting cell parameters to Cartesian form, and an ETA class.

amd.utils.**diameter**(*cell*)

Diameter of a unit cell (as a square matrix in Cartesian form) in 3 or fewer dimensions.

amd.utils.**cellpar_to_cell**(*a*, *b*, *c*, *alpha*, *beta*, *gamma*)

Simplified version of function from `ase.geometry` of the same name. 3D unit cell parameters a,b,c,,, –> cell as 3x3 NumPy array.

amd.utils.**cellpar_to_cell_2D**(*a*, *b*, *alpha*)

2D unit cell parameters a,b, –> cell as 2x2 ndarray.

amd.utils.**neighbours_from_distance_matrix**(*n: int*, *dm: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Given a distance matrix, find the n nearest neighbours of each item.

Parameters

- **n** (*int*) – Number of nearest neighbours to find for each item.
- **dm** (*numpy.ndarray*) – 2D distance matrix or 1D condensed distance matrix.

Returns **nn_dm, inds** – `nn_dm[i][j]` is the distance from item `i` to its `j+1` st nearest neighbour, and `inds[i][j]` is the index of this neighbour (`j+1` since index 0 is the first nearest neighbour).

Return type Tuple[numpy.ndarray, numpy.ndarray]

amd.utils.**lattice_cubic**(*scale=1*, *dims=3*)

Return a pair `(motif, cell)` representing a hexagonal lattice, passable to `amd.AMD()` or `amd.PDD()`.

amd.utils.**lattice_hexagonal**(*scale=1*)

Dimension 3 only. Return a pair `(motif, cell)` representing a cubic lattice, passable to `amd.AMD()` or `amd.PDD()`.

amd.utils.**random_cell**(*length_bounds=(1, 2)*, *angle_bounds=(60, 120)*, *dims=3*)

Dimensions 2 and 3 only. Random unit cell with uniformally chosen length and angle parameters between bounds.

CHAPTER

TWELVE

AVERAGE-MINIMUM-DISTANCE: ISOMETRICALLY INVARIANT CRYSTAL FINGERPRINTS

Implements fingerprints (*isometry invariants*) of crystal structures based on geometry: average minimum distances (AMD) and pointwise distance distributions (PDD).

- **PyPI project:** https://pypi.org/project/average-minimum-distance
- **Documentation:** https://average-minimum-distance.readthedocs.io
- **Source code:** https://github.com/dwiddo/average-minimum-distance
- **References** (*jump to bib references*):
  - *Average minimum distances of periodic point sets - foundational invariants for mapping periodic crystals*. MATCH Communications in Mathematical and in Computer Chemistry, 87(3):529-559 (2022). https://doi.org/10.46793/match.87-3.529W
  - *Pointwise distance distributions of periodic point sets*. arXiv preprint arXiv:2108.04798 (2021). https://arxiv.org/abs/2108.04798

12.1 What's amd?

The typical representation of a crystal as a motif and cell is ambiguous, as there are many ways to define the same crystal. This package implements new *isometric invariants*: average minimum distances (AMD) and pointwise distance distributions (PDD), which always take the same value for any two (isometrically) identical input crystals. They do this in a continuous way, so similar crystals have a small distance between their invariants.

12.1.1 Brief description of AMD and PDD

The pointwise distance distribution (PDD) records the environment of each atom in a unit cell by listing the distances from each atom to neighbouring atoms in order, with some extra steps to ensure independence of cell and motif. A PDD is a collection of lists with attached weights (a matrix). Two PDDs are compared by finding an optimal matching between the two sets of lists while respecting the weights (Earth Mover's distance), and when the crystals are geometrically identical (regardless of choice of motif and cell) there is always a perfect matching resulting in a distance of zero.

The average minimum distance (AMD) averages the PDD over atoms in a unit cell to make a vector, which is also the same for any choice of cell and motif. Since AMDs are just vectors, comparing by AMD is much faster than PDD, though AMD contains less information in theory.

Both AMD and PDD have a parameter k, the number of nearest neighbours to consider for each atom, which is the length of the AMD vector or the number of columns in the PDD (plus an extra column for weights of rows).

12.2 Getting started

Use pip to install average-minimum-distance:

```
pip install average-minimum-distance
```

Then import average-minimum-distance with `import amd`.

`amd.compare()` compares sets of crystals by AMD or PDD in one line, e.g. by PDD with k = 100:

```
import amd
df = amd.compare('crystals.cif', by='PDD', k=100)
```

A pandas DataFrame is returned of the distance matrix with names of crystals in rows and columns. It can also take two paths and compare crystals in one file with the other, for example

```
df = amd.compare('crystals_1.cif', 'crystals_2.cif' by='AMD', k=100)
```

Either first or second argument can be lists of cif paths (or file objects) which are combined in the final distance matrix.

`amd.compare()` reads crystals and calculates their AMD or PDD, but throws them away. It may be faster to save these to a file (e.g. `pickle`), see sections below on how to separately read, calculate and compare.

If `csd-python-api` is installed, the compare function can also accept one or more CSD refcodes or other file formats instead of cifs (pass `reader='ccdc'`).

12.2.1 Choosing a value of k

The parameter k of the invariants is the number of nearest neighbour atoms considered for each atom in the unit cell, e.g. k = 5 looks at the 5 nearest neighbours of each atom. Two crystals with the same unit molecule will have a small AMD/PDD distance for small enough k. A larger k will mean the environments of atoms in one crystal must line up with those in the other up to a larger radius to have a small AMD/PDD distance. Very large k does not mean better comparisons, as the invariants start to converge to depend only on density.

12.2.2 Reading crystals from a file, calculating the AMDs and PDDs

This code reads a .cif with `amd.CifReader` and computes the AMDs (k = 100):

```
import amd
reader = amd.CifReader('path/to/file.cif')
amds = [amd.AMD(crystal, 100) for crystal in reader]  # calc AMDs
```

Note: CifReader accepts optional arguments, e.g. for removing hydrogen and handling disorder. See the documentation for details.

To calculate PDDs, just replace `amd.AMD` with `amd.PDD`.

If `csd-python-api` is installed, crystals can be read directly from your local copy of the CSD with `amd.CSDReader`, which accepts a list of refcodes. CifReader can accept file formats other than cif by passing `reader='ccdc'`.

12.2.3 Comparing by AMD or PDD

`amd.AMD_pdist` and `amd.PDD_pdist` take a list of invariants and compares them pairwise, returning a *condensed distance matrix* like SciPy's `pdist` function.

```
# read and calculate AMDs and PDDs (k=100)
crystals = list(amd.CifReader('path/to/file.cif'))
amds = [amd.AMD(crystal, 100) for crystal in reader]
pdds = [amd.PDD(crystal, 100) for crystal in reader]

amd_cdm = amd.AMD_pdist(amds) # compare a list of AMDs pairwise
pdd_cdm = amd.PDD_pdist(pdds) # compare a list of PDDs pairwise

# Use SciPy's squareform for a symmetric 2D distance matrix
from scipy.distance.spatial import squareform
amd_dm = squareform(amd_cdm)
```

Note: if you want both AMDs and PDDs like above, it's faster to compute the PDDs first and use ``amd.PDD_to_AMD()`` rather than computing both from scratch.

The default metric for comparison is `chebyshev` (L-infinity), though it can be changed to anything accepted by SciPy's `pdist`, e.g. `euclidean`.

If you have two sets of crystals and want to compare all crystals in one to the other, use `amd.AMD_cdist` or `amd.PDD_cdist`.

```
set1 = amd.CifReader('set1.cif')
set2 = amd.CifReader('set2.cif')
amds1 = [amd.AMD(crystal, 100) for crystal in set1]
amds2 = [amd.AMD(crystal, 100) for crystal in set2]

# dm[i][j] = distance(amds1[i], amds2[j])
dm = amd.AMD_cdist(amds)
```

12.3 Example: PDD-based dendrogram

This example compares some crystals in a cif by PDD (k = 100) and plots a single linkage dendrogram:

```
import amd
import matplotlib.pyplot as plt
from scipy.cluster import hierarchy

crystals = list(amd.CifReader('crystals.cif'))
names = [crystal.name for crystal in crystals]
pdds = [amd.PDD(crystal, 100) for crystal in crystals]
cdm = amd.PDD_pdist(pdds)
Z = hierarchy.linkage(cdm, 'single')
dn = hierarchy.dendrogram(Z, labels=names)
plt.show()
```

12.4 Example: Finding n nearest neighbours in one set from another

This example finds the 10 nearest PDD-neighbours in set 2 for every crystal in set 1.

```
import numpy as np
import amd

n = 10
df = amd.compare('set1.cif', 'set2.cif', k=100)
dm = df.values

# Uses np.argpartiton (partial argsort) and np.take_along_axis to find
# nearest neighbours of each item in set1. Works for any distance matrix.
nn_inds = np.array([np.argpartition(row, n)[:n] for row in dm])
nn_dists = np.take_along_axis(dm, nn_inds, axis=-1)
sorted_inds = np.argsort(nn_dists, axis=-1)
nn_inds = np.take_along_axis(nn_inds, sorted_inds, axis=-1)
nn_dists = np.take_along_axis(nn_dists, sorted_inds, axis=-1)

for i in range(len(set1)):
    print('neighbours of', df.index[i])
    for j in range(n):
        print('neighbour', j+1, df.columns[nn_inds[i][j]], 'dist:', nn_dists[i][j])
```

12.5 Cite us

Use the following bib references to cite AMD or PDD.

Average minimum distances of periodic point sets - foundational invariants for mapping periodic crystals. MATCH Communications in Mathematical and in Computer Chemistry, 87(3), 529-559 (2022). https://doi.org/10.46793/match.87-3.529W.

```
@article{10.46793/match.87-3.529W,
  title = {Average Minimum Distances of periodic point sets - foundational invariants␣
↪for mapping periodic crystals},
  author = {Widdowson, Daniel and Mosca, Marco M and Pulido, Angeles and Kurlin, Vitaliy␣
↪and Cooper, Andrew I},
  journal = {MATCH Communications in Mathematical and in Computer Chemistry},
  doi = {10.46793/match.87-3.529W},
  volume = {87},
  number = {3},
  pages = {529-559},
  year = {2022}
}
```

Pointwise distance distributions of periodic point sets. arXiv preprint arXiv:2108.04798 (2021). https://arxiv.org/abs/2108.04798.

```
@misc{arXiv:2108.04798,
  author = {Widdowson, Daniel and Kurlin, Vitaliy},
  title = {Pointwise distance distributions of periodic point sets},
  year = {2021},
  eprint = {arXiv:2108.04798},
}
```

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX