average-minimum-distance

Release 1.1.7

Daniel Widdowson

Feb 01, 2022

COMMON TASKS

Read below to get started with amd, or follow the links below for more details about specific tasks.

CHAPTER

ONE

READING .CIFS

If you have a .CIF file, use `amd.CifReader` to extract the crystals. Here are some common patterns when reading .CIFs:

```
# 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))

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

# create list of crystals in individual .cifs in a folder
# read_one returns the first crystal, nice when the .cif has one crystal
import os
crystals = [amd.CifReader(os.path.join(folder, file)).read_one()
            for file in os.listdir(folder)]
```

The `CifReader` yields `PeriodicSet` objects, which can be passed to `amd.AMD()` or `amd.PDD()`. The `PeriodicSet` has attributes `.name`, `.motif`, `.cell`, `.types` (atomic types), and information about the asymmetric unit to help with AMD and PDD calculations.

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

1.1 Reading options

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

```
amd.CifReader('file.cif',
              reader='ase',
              remove_hydrogens=False,
              disorder='skip',
              heaviest_component=False,
              extract_data=None,
              include_if=None)
```

- `reader` controls the backend package used to parse the file. The default is `ase`; to use csd-python-api pass `ccdc`. The ccdc reader can read any format accepted by ccdc's 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` takes the heaviest connected molecule in the motif, intended for removing solvents. Only available when `reader='ccdc'`.
- `extract_data` is used to extract more data from crystals, as in the example below.
- `include_if` can remove unwanted structures, as in the example below.

An example only reading entries with a `.polymorph` label and extracting additional data:

```
import amd
from amd import ccdc_utils as cu

def has_polymorph_label(entry):
    if entry.polymorph is None:
        return False
    return True

include_if = [has_polymorph_label]

# data to extract. can be any functions accepting an entry, but ccdc_utils has some␣
↪useful ones
extract = {
    'density':    cu.density,
    'formula':    cu.formula,
    'cell_str':   cu.cell_str_as_html,
    'spacegroup': cu.spacegroup
}

reader = amd.CifReader('file.cif', reader='ccdc', extract_data=extract, include_
↪if=include_if)
...
```

CHAPTER

TWO

READING FROM THE CSD

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

`amd.CSDReader` accepts refcode(s) and yields the chosen crystals. If `None` or `'CSD'` are passed instead of refcode(s), it reads the whole CSD. If the optional parameter `families` is `True`, the refcode(s) given are interpreted as refcode families: any entry whose ID starts with the given string is included.

Here are some common patterns for the CSDReader:

```
# 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 periodic_set in amd.CSDReader(refcodes, families=True):
    amds.append(amd.AMD(periodic_set, 100))
```

The `CSDReader` yields `PeriodicSet` objects, which can be passed to `amd.AMD()` or `amd.PDD()`. The `PeriodicSet` has attributes `.name`, `.motif`, `.cell`, `.types` (atomic types), and information about the asymmetric unit to help with AMD and PDD calculations.

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

2.1 Optional parameters

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

```
amd.CSDReader(refcodes=None,
              families=False,
              remove_hydrogens=False,
              disorder='skip',
              heaviest_component=False,
              extract_data=None,
              include_if=None)
```

- 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. Only available when reader='ccdc'.
- extract_data is used to extract more data from crystals.
- include_if can remove unwanted structures.

CHAPTER

THREE

USING AMDS

3.1 Calculating AMDs

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

If you have a .cif file, use `amd.CifReader` to read the crystals. If `csd-python-api` is installed and you have CSD refcodes, use `amd.CSDReader`. You can also give the coordinates of motif points and unit cell as a tuple of numpy arrays, in Cartesian form. Examples:

```
# 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
amds = [amd.AMD(crystal, 100) for crystal in amd.CSDReader('DEBXIT', families=True)]

# AMD of 3D cubic lattice
motif = np.array([[0,0,0]])
cell  = np.identity(3)
cubic_amd = amd.AMD((motif, cell), 100)
```

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

Note: If you want both the AMD and PDD of a crystal, first get the PDD and then use `amd.PDD_to_AMD()` to get the AMD from it to save time.

3.2 Comparing by AMD

Any metric can be used to compare AMDs, but the *amd.compare* module has functions to compare for you.

Most useful are *compare.AMD_pdist()* and *compare.AMD_cdist()*, which mimic the interface of 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). 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')]
```

(continues on next page)

(continued from previous page)

```
distance_matrix = amd.AMD_cdist(amds1, amds2)

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

3.2.1 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.
- `k` will truncate the passed AMDs to length `k` before comaparing (so `k` must not be larger than the passed AMDs). Useful if comparing for several `k` values.
- `low_memory` (default `False`) uses an alternative slower algorithm that keeps memory use low for much larger input sizes. Currently only `metric='chebyshev'` is accepted with `low_memory`.

CHAPTER

FOUR

USING PDDS

4.1 Calculating PDDs

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

If you have a .cif file, use `amd.CifReader` to read the crystals. If `csd-python-api` is installed and you have CSD refcodes, use `amd.CSDReader`. You can also give the coordinates of motif points and unit cell as a tuple of numpy arrays, in Cartesian form. Examples:

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

# get PDDs of crystals in DEBXIT family
pdds = [amd.PDD(crystal, 100) for crystal in amd.CSDReader('DEBXIT', families=True)]

# PDD of 3D cubic lattice
motif = np.array([[0,0,0]])
cell  = np.identity(3)
cubic_pdd = amd.PDD((motif, cell), 100)
```

Each PDD returned by `amd.PDD(c, k)` is a matrix with `k`+1 columns.

4.1.1 Calculation options

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

```
amd.PDD(periodic_set, k, order=True, collapse=True, collapse_tol=1e-4)
```

`order` lexicograpgically orders the rows of the PDD, and `collapse` merges rows if all the elements are within `collapse_tol`. The technical definition of PDD requires doing both in order for PDD to satisfy invariance, but sometimes it's useful to disable the behaviours, particularly so that the PDD rows are in the same order as the passed motif points. Without ordering and collapsing, isometric inputs could give different PDDs; but the Earth mover's distance between the PDDs would still be 0.

4.2 Comparing by PDD

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

Most useful are *compare.PDD_pdist()* and *compare.PDD_cdist()*, which mimic the interface of 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). 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 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).

4.2.1 Comparison options

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

- `metric` chooses the metric used for comparison of PDD rows, as explained above. See scipy's cdist/pdist for a list of accepted metrics.
- `k` will truncate the passed PDDs to length `k` before comaparing (so `k` must not be larger than the passed PDDs). Useful if comparing for several `k` values.
- `verbose` (default `False`) prints an ETA to the terminal.

CHAPTER

FIVE

WRITE CRYSTALS AND FINGERPRINTS TO A FILE

For large sets of crystals, parsing .CIF files can be slow. *amd.io.SetWriter* and *amd.io.SetReader* are for writing and reading back crystals (with additional data) to a compressed hdf5 file.

```
# write the crystals and their AMDs to a file using the crystal's .tags
with amd.SetWriter('crystals_and_AMD100.hdf5') as writer:
    for crystal in amd.CifReader('file.cif'):
        crystal.tags['AMD100'] = amd.AMD(crystal, 100)
        writer.write(crystal)

# read back the crystals and their AMDs
crystals = list(amd.SetReader('crystals_and_AMD100.hdf5'))
amds = [crystal.tags['AMD100'] for crystal in crystals]
```

CHAPTER

SIX

MISCELLANEOUS

6.1 Minimum spanning trees

Previously, minimum spanning trees have been used to visualise crystal landscapes with AMD or PDD, particularly with tmap. The *compare* module includes two functions to produce MSTs, *compare.AMD_mst()* and *compare.PDD_mst()*. They each accept a list of AMDs/PDDs and return an edge list `(i, j, d)` where the crystals at indices `i` and `j` are connected by an edge with weight `d`.

The mst functions take the same optional arguments as other compare functions, including `low_memory` for `AMD_mst` and `filter` for `PDD_mst`.

6.2 AMDs and PDDs of finite point sets

AMDs and PDDs also work for finite point sets. The functions *calculate.finite_AMD()* and *calculate.finite_PDD()* accept just a numpy array containing the points, returning the fingerprint of the finite point set. Unlike `amd.AMD` and `amd.PDD` no integer `k` is passed; instead the distances to all neighbours are found (number of columns = no of points - 1).

6.3 Higher order PDDs

In addition to the AMD and regular PDD, there is a sequence of 'higher order' PDD invariants accessible by passing an int to the optional parameter `order` in `amd.PDD()` or `amd.finite_PDD()`. Each invariant in the sequence contains more information and grows in complexity.

To summarise, each row of PDD^h pertains to an h-tuple of motif points, hence PDD^h contains m choose h rows before collapsing. Apart from weights, the first element of each row is the finite PDD (order 1) of the h-tuple of points (just the distance between them if h = 2). The rest of a row is lexicographically ordered h-tuples of distances from the 3 points to other points in the set.

Note that before 1.1.7 the `order` parameter controlled the sorting of rows, that is now `lexsort`. The default is `order`=1 which is the normal PDD (a matrix shape `(m, k+1)` with weights in the first column). For any integer > 1, `amd.PDD()` returns a tuple `(weights, dist, pdd)` where `weights` are the usual weights (number of row appearances / total rows), `dist` contains the finite PDDs (or distances) and `pdd` contains the h-tuples (as described above).

6.4 Inverse design

Not yet implemented.

CHAPTER

SEVEN

AMD.CALCULATE MODULE

Functions to calculate AMDs and PDDs.

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

Computes an AMD vector up to *k* from a periodic set.

Parameters

- **periodic_set** (*periodicset.PeriodicSet* or tuple of ndarrays) – A periodic set represented by a *periodicset.PeriodicSet* object or by a tuple (motif, cell) with coordinates in Cartesian form.
- **k** (*int*) – Length of AMD returned.

Returns An ndarray of shape (k,), the AMD of `periodic_set` up to *k*.

Return type ndarray

Examples

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

```
amds = []
for periodic_set in amd.CifReader('mycif.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*, *order: int = 1*, *lexsort: bool = True*, *collapse: bool = True*, *collapse_tol: float = 0.0001*) → numpy.ndarray

Computes a PDD up to k from a periodic set.

Parameters

- **periodic_set** (*periodicset.PeriodicSet* or tuple of ndarrays) – A periodic set represented by a *periodicset.PeriodicSet* object or by a tuple (motif, cell) with coordinates in Cartesian form.
- **k** (*int*) – Number of columns in the PDD, plus one for the first column of weights.
- **order** (*int*) – Order of the PDD, default 1. See papers for a description of higher-order PDDs.
- **lexsort** (*bool, optional*) – Whether or not to lexicographically order the rows. Default True.
- **collapse** (*bool, optional*) – Whether or not to collapse identical rows (within a tolerance). Default True.
- **collapse_tol** (*float*) – If two rows have all entries closer than collapse_tol, they get collapsed. Default is 1e-4.

Returns An ndarray with k+1 columns, the PDD of `periodic_set` up to *k*.

Return type ndarray

Examples

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

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

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):
    pdds.append(amd.PDD(periodic_set, 10, collapse=False))  # do not collapse rows
```

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.**finite_AMD**(*motif: numpy.ndarray*)

Computes the AMD of a finite point set (up to k = *len(motif) - 1*).

Parameters **motif** (*ndarray*) – Cartesian coordinates of points in a set. Shape (n_points, dimensions)

Returns An vector length len(motif) - 1, the AMD of `motif`.

Return type ndarray

Examples

Find AMD 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_amd = amd.finite_AMD(trapezium)
kite_amd = amd.finite_AMD(kite)

dist = amd.AMD_pdist(trap_amd, kite_amd)
```

amd.calculate.**finite_PDD**(*motif: numpy.ndarray*, *order: int = 1*, *lexsort: bool = True*, *collapse: bool = True*, *collapse_tol: float = 0.0001*)

Computes the PDD of a finite point set (up to k = *len(motif) - 1*).

Parameters

- **motif** (*ndarray*) – Cartesian coordinates of points in a set. Shape (n_points, dimensions)
- **order** (*int*) – Order of the PDD, default 1. See papers for a description of higher-order PDDs.
- **lexsort** (*bool, optional*) – Whether or not to lexicographically order the rows. Default True.
- **collapse** (*bool, optional*) – Whether or not to collapse identical rows (within a tolerance). Default True.
- **collapse_tol** (*float*) – If two rows have all entries closer than collapse_tol, they get collapsed. Default is 1e-4.

Returns An ndarray with len(motif) columns, the PDD of `motif`.

Return type 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.finite_PDD(trapezium)
kite_pdd = amd.finite_PDD(kite)

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

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

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

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

Returns The AMD of the periodic set.

Return type ndarray

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

Calculate the point packing coefficient (PPC) of `periodic_set`.

The PPC is a constant of any periodic set determining the asymptotic behaviour of its AMD or PDD as $k \to \infty$.

As $k \to \infty$, the ratio $\text{AMD}_k / \sqrt[n]{k}$ approaches 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) – ndarrays (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 or PDDs, finding nearest neighbours and minimum spanning trees.

amd.compare.**emd**(*pdd*, *pdd_*, *metric='chebyshev'*, *return_transport=False*, ***kwargs*)

Earth mover's distance between two PDDs.

Parameters

- **pdd** (*ndarray*) – A PDD given by *calculate.PDD()*.
- **pdd_** (*ndarray*) – A PDD given by *calculate.PDD()*.
- **metric** (*str or callable, optional*) – Usually rows are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by `scipy.spatial.distance.cdist`.

Returns Earth mover's distance between PDDs, where rows of the PDDs are compared with `metric`.

Return type float

Raises **ValueError** – Thrown if reference and comparison do not have the same number of columns.

amd.compare.**AMD_cdist**(*amds: Union[numpy.ndarray, List[numpy.ndarray]]*, *amds_: Union[numpy.ndarray, List[numpy.ndarray]]*, *k: Optional[int] = None*, *metric: str = 'chebyshev'*, *low_memory: bool = False*, ***kwargs*) → numpy.ndarray

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

Parameters

- **amds** (*array_like*) – An array/list of AMDs.
- **amds_** (*array_like*) – An array/list of AMDs.
- **k** (*int, optional*) – If `None`, compare entire AMDs. Set `k` to an int to compare for a specific `k` (less than the maximum).
- **low_memory** (*bool, optional*) – Optionally use a slower but more memory efficient method for large collections of AMDs (Chebyshev/l-inf distance only).
- **metric** (*str or callable, optional*) – Usually AMDs are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by `scipy.spatial.distance.cdist`.

Returns Returns a distance matrix shape `(len(amds), len(amds_))`. The ij th entry is the distance between `amds[i]` and `amds[j]` given by the `metric`.

Return type ndarray

amd.compare.**AMD_pdist**(*amds: Union[numpy.ndarray, List[numpy.ndarray]]*, *k: Optional[int] = None*, *low_memory: bool = False*, *metric: str = 'chebyshev'*, ***kwargs*) → numpy.ndarray

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

Parameters

- **amds** (*array_like*) – An array/list of AMDs.
- **k** (*int, optional*) – If None, compare whole AMDs (largest k). Set k to an int to compare for a specific k (less than the maximum).
- **low_memory** (*bool, optional*) – Optionally use a slightly slower but more memory efficient method for large collections of AMDs (Chebyshev/l-inf distance only).
- **metric** (*str or callable, optional*) – Usually AMDs are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by scipy.spatial.distance.cdist.

Returns Returns a condensed distance matrix. Collapses a square distance matrix into a vector just keeping the upper half. Use scipy.spatial.distance.squareform to convert to a square distance matrix.

Return type ndarray

amd.compare.**PDD_cdist**(*pdds: List[numpy.ndarray]*, *pdds_: List[numpy.ndarray]*, *k: Optional[int] = None*, *metric: str = 'chebyshev'*, *verbose: bool = False*, ***kwargs*) → numpy.ndarray

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

Parameters

- **pdds** (*list of ndarrays*) – A list of PDDs.
- **pdds_** (*list of ndarrays*) – A list of PDDs.
- **k** (*int, optional*) – If None, compare whole PDDs (largest k). Set k to an int to compare for a specific k (less than the maximum).
- **metric** (*str or callable, optional*) – Usually PDD rows are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by scipy.spatial.distance.cdist.
- **verbose** (*bool, optional*) – Optionally print an ETA to terminal as large collections can take some time.

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 ndarray

amd.compare.**PDD_pdist**(*pdds: List[numpy.ndarray]*, *k: Optional[int] = None*, *metric: str = 'chebyshev'*, *verbose: bool = False*, ***kwargs*) → numpy.ndarray

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

Parameters

- **pdds** (*list of ndarrays*) – A list of PDDs.
- **k** (*int, optional*) – If None, compare whole PDDs (largest k). Set k to an int to compare for a specific k (less than the maximum).
- **metric** (*str or callable, optional*) – Usually PDD rows are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by scipy.spatial.distance.cdist.
- **verbose** (*bool, optional*) – Optionally print an ETA to terminal as large collections can take some time.

Returns Returns a condensed distance matrix. Collapses a square distance matrix into a vector just keeping the upper half. Use `scipy.spatial.distance.squareform` to convert to a square distance matrix.

Return type ndarray

amd.compare.**filter**(*n: int*, *pdds: List[numpy.ndarray]*, *pdds_: Optional[List[numpy.ndarray]] = None*, *k: Optional[int] = None*, *low_memory: bool = False*, *metric: str = 'chebyshev'*, *verbose: bool = False*, ***kwargs*) → Tuple[numpy.ndarray, numpy.ndarray]

For each item in `pdds`, get the `n` nearest items in `pdds_` by AMD, then compare references to these nearest items with PDDs. Tries to comprimise between the speed of AMDs and the accuracy of PDDs.

If `pdds_` is `None`, this essentially sets `pdds_ = pdds`, i.e. do an 'AMD neighbourhood graph' for one set whose weights are PDD distances.

Parameters

- **n** (*int*) – Number of nearest neighbours to find.
- **pdds** (*list of ndarrays*) – A list of PDDs.
- **pdds_** (*list of ndarrays, optional*) – A list of PDDs.
- **k** (*int, optional*) – If `None`, compare entire PDDs. Set `k` to an int to compare for a specific `k` (less than the maximum).
- **low_memory** (*bool, optional*) – Optionally use a slightly slower but more memory efficient method for large collections of AMDs (Chebyshev/l-inf distance only).
- **metric** (*str or callable, optional*) – Usually PDD rows are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by `scipy.spatial.distance.cdist`.
- **verbose** (*bool, optional*) – Optionally print an ETA to terminal as large collections can take some time.

Returns For the i th item in reference and some $j < n$, `distance_matrix[i][j]` is the distance from reference `i` to its j-th nearest neighbour in comparison (after the AMD filter). `indices[i][j]` is the index of said neighbour in `pdds_`.

Return type tuple of ndarrays (distance_matrix, indices)

amd.compare.**AMD_mst**(*amds: Union[int, float, complex, str, bytes, numpy.generic, Sequence[Union[int, float, complex, str, bytes, numpy.generic]], Sequence[Sequence[Any]], numpy.typing._array_like._SupportsArray]*, *k: Optional[int] = None*, *low_memory: bool = False*, *metric: str = 'chebyshev'*, ***kwargs*) → List[Tuple[int, int, float]]

Return list of edges in a minimum spanning tree based on AMDs.

Parameters

- **amds** (*ndarray or list of ndarrays*) – An array/list of AMDs.
- **k** (*int, optional*) – If `None`, compare whole PDDs (largest `k`). Set `k` to an int to compare for a specific `k` (less than the maximum).
- **metric** (*str or callable, optional*) – Usually PDD rows are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by `scipy.spatial.distance.cdist`.

Returns Each tuple `(i,j,w)` is an edge in the mimimum spanning tree, where `i` and `j` are the indices of nodes and `w` is the AMD distance.

Return type list of tuples

amd.compare.**PDD_mst**(*pdds: List[numpy.ndarray]*, *amd_filter_cutoff: Optional[int] = None*, *k: Optional[int] = None*, *metric: str = 'chebyshev'*, *verbose: bool = False*, ***kwargs*) → List[Tuple[int, int, float]]

Return list of edges in a minimum spanning tree based on PDDs.

Parameters

- **pdds** (*list of ndarrays*) – A list of PDDs.
- **amd_filter_cutoff** (*int, optional*) – If specified, apply the AMD filter behaviour of *filter()*. This is the `n` passed to *filter()*, the number of neighbours to connect in the neighbourhood graph.
- **k** (*int, optional*) – If `None`, compare whole PDDs (largest `k`). Set `k` to an int to compare for a specific `k` (less than the maximum).
- **metric** (*str or callable, optional*) – Usually PDD rows are compared with the Chebyshev/l-infinity distance. Can take any metric + kwargs accepted by `scipy.spatial.distance.cdist`.
- **verbose** (*bool, optional*) – Optionally print an ETA to terminal as large collections can take some time.

Returns Each tuple `(i,j,w)` is an edge in the mimimum spanning tree, where `i` and `j` are the indices of nodes and `w` is the PDD distance.

Return type list of tuples

amd.compare.**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** (*ndarray*) – 2D distance matrix or 1D condensed distance matrix.

Returns For item `i`, `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 of ndarrays (nn_dm, inds)

amd.compare.**mst_from_distance_matrix**(*dm*)

CHAPTER

NINE

AMD.IO MODULE

Contains I/O tools, including a .CIF reader and CSD reader (`csd-python-api` only) to extract periodic set representations of crystals which can be passed to *calculate.AMD()* and *calculate.PDD()*.

The *CifReader* and *CSDReader* return *periodicset.PeriodicSet* objects, which can be given to *calculate.AMD()* and *calculate.PDD()* to calculate the AMD/PDD of a crystal.

These intermediate *periodicset.PeriodicSet* representations can be written to a .hdf5 file with *SetWriter* (along with their metadata), which can be read back with *SetReader*. This is much faster than rereading a .CIF or recomputing invariants.

class amd.io.**CifReader**(*filename*, *reader='ase'*, *remove_hydrogens=False*, *disorder='skip'*, *heaviest_component=False*, *extract_data=None*, *include_if=None*, *show_warnings=True*)

Bases: `amd.io._Reader`

Read all structures in a .CIF with `ase` or `ccdc`, yielding *periodicset.PeriodicSet* objects which can be passed to *calculate.AMD()* or *calculate.PDD()*.

Examples

Put all crystals in mycif.cif in a list:

```
structures = list(amd.CifReader('mycif.cif'))
```

Reads just one, convinient when the .CIF has one crystal:

```
periodic_set = amd.CifReader('mycif.cif').read_one()
```

If the folder 'cifs' has many .CIFs each with one crystal:

```
import os
folder = 'cifs'
structures = []
for filename in os.listdir(folder):
    p_set = amd.CifReader(os.path.join(folder, filename)).read_one()
    structures.append(p_set)
```

Make list of AMD (with k=100) for crystals in mycif.cif:

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

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

Bases: amd.io._Reader

Read Entries from the CSD, yielding *periodicset.PeriodicSet* objects.

The CSDReader returns *periodicset.PeriodicSet* objects which can be passed to *calculate.AMD()* or *calculate.PDD()*.

Examples

Get crystals with 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 AMD (with k=100) for crystals in these families:

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

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

Read a PeriodicSet given any CSD refcode.

class amd.io.**SetWriter**(*filename: str*)

Bases: object

Write several *periodicset.PeriodicSet* objects to a .hdf5 file. Reading the .hdf5 is much faster than parsing a .CIF file.

write(*periodic_set:* amd.periodicset.PeriodicSet, *name: Optional[str] = None*)

Write a PeriodicSet object to file.

iwrite(*periodic_sets: Iterable[*amd.periodicset.PeriodicSet*]*)

Write *periodicset.PeriodicSet* objects from an iterable to file.

close()

Close the *SetWriter*.

class amd.io.**SetReader**(*filename: str*)

Bases: object

Read *periodicset.PeriodicSet* objects from a .hdf5 file written with *SetWriter*. Acts like a read-only dict that can be iterated over (preserves write order).

close()
Close the *SetReader*.

family(*refcode: str*) → Iterable[*amd.periodicset.PeriodicSet*]
Yield any :class:`.periodicset.PeriodicSet`s whose name starts with input refcode.

keys()
Yield names of items in the *SetReader*.

extract_tags() → dict
Return `dict` with scalar data in the tags of items in *SetReader*.

Dict is in format easily passable to `pandas.DataFrame`, as in:

```
reader = amd.SetReader('periodic_sets.hdf5')
data = reader.extract_tags()
df = pd.DataFrame(data, index=reader.keys(), columns=data.keys())
```

Format of returned dict is for example:

```
{
    'density': [1.231, 2.532, ...],
    'family':  ['CBMZPN', 'SEMFAU', ...],
    ...
}
```

where the inner lists have the same order as the items in the SetReader.

amd.io.**crystal_to_periodicset**(*crystal*)

amd.io.**cifblock_to_periodicset**(*block*)

CHAPTER

TEN

AMD.PERIODICSET MODULE

Implements the class *PeriodicSet* representing a periodic set, defined by a motif and unit cell.

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. They can be written to a file with *io.SetWriter* which can be read with *io.SetReader*.

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

Bases: `object`

Represents a periodic set (which mathematically represents a crystal).

Has attributes motif, cell and name (which can be `None`). *PeriodicSet* objects are returned by the readers in the *io* module.

astype(*dtype*)
Returns copy of the *PeriodicSet* with `.motif` and `.cell` casted to `dtype`.

to_dict() → dict
Return `dict` with scalar data in the tags of the *PeriodicSet*.

Format of returned `dict` is for example:

```
{
    'density': 1.231,
    'family': 'CBMZPN',
    ...
}
```

CHAPTER

ELEVEN

AMD.UTILS MODULE

General utility functions and classes.

amd.utils.**cellpar_to_cell**(*a*, *b*, *c*, *alpha*, *beta*, *gamma*)
Simplified version of function from ase.geometry.

Unit cell params a,b,c,,, –> cell as 3x3 ndarray.

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

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

class amd.utils.**ETA**(*to_do*, *update_rate=100*)
Bases: `object`

Pass total amount to do on construction, then call .update() on every loop. ETA will estimate an ETA and print it to the terminal.

update()
Call when one item is finished.

amd.utils.**extract_tags**(*periodic_sets*) → dict
Return `dict` with scalar data in the tags of PeriodicSets in the passed list.

Dict is in format passable to `pandas.DataFrame`, as in:

```
periodic_sets = list(amd.SetReader('periodic_sets.hdf5'))
names = [s.name for s in periodic_sets]
data = amd.utils.extract_tags(periodic_sets)
df = pd.DataFrame(data, index=names, columns=data.keys())
```

Format of returned dict is for example:

```
{
    'density': [1.231, 2.532, ...],
    'family':  ['CBMZPN', 'SEMFAU', ...],
    ...
}
```

where the inner lists have the same order as the items in `periodic_sets`.

amd.utils.**neighbours_df_dict**(*n_neighbours*, *references*, *comparisons*, *invariant_key=None*)
n, reference psets, comparison psets –> dict passable to DataFrame

Example:: data = amd.neighbours_df_dict(10, pdds, **pdds_**) df = pd.DataFrame(data, index=ref_names)

CHAPTER

TWELVE

AMD.CCDC_UTILS MODULE

Functions to use with the `extract_data` parameter of *io.CifReader* or *io.CSDReader*.

amd.ccdc_utils.**density**(*entry*)
Calculated density of the crystal.

amd.ccdc_utils.**chemical_name**(*entry*)
The chemical name of the entry.

amd.ccdc_utils.**chemical_name_as_html**(*entry*)
The chemical name of the entry formatted as HTML.

amd.ccdc_utils.**refcode_family**(*entry*)
First 6 characters of the refcode, aka the refcode 'family'.

amd.ccdc_utils.**formula**(*entry*)
The published chemical formula in an entry. If no published chemical formula is available it will be calculated from the molecule.

amd.ccdc_utils.**is_organic**(*entry*)
Whether the structure is organic.

amd.ccdc_utils.**polymorph**(*entry*)
Polymorphic information about the crystal if given otherwise *None*.

amd.ccdc_utils.**cell_str_as_html**(*entry*)
Lengths + angles cell parameters of the crystal formatted as HTML.

amd.ccdc_utils.**crystal_system**(*entry*)
The space group system of the crystal.

amd.ccdc_utils.**spacegroup**(*entry*)
The space group symbol of the crystal.

amd.ccdc_utils.**has_disorder**(*entry*)
Returns True if any disorder is found in the crystal.

amd.ccdc_utils.**molecular_centres**(*entry*)
(Geometric) molecular centres lying in the unit cell.

CHAPTER

THIRTEEN

AVERAGE-MINIMUM-DISTANCE: ISOMETRICALLY INVARIANT CRYSTAL FINGERPRINTS

If you use our code in your work, please cite our paper at arxiv.org/abs/2009.02488. The bib reference is at the bottom of this page; *click here jump to it*.

13.1 What's amd?

A crystal is an arrangement of atoms which periodically repeats according to some lattice. The atoms and lattice defining a crystal are typically recorded in a .CIF file, but this representation is ambiguous, i.e. different .CIF files can define the same crystal. This package implements new *isometric invariants* called AMD (average minimum distance) and PDD (point-wise distance distribution) based on inter-point distances, which are guaranteed to take the same value for all equivalent representations of a crystal. They do this in a continuous way; crystals which are similar have similar AMDs and PDDs.

For a technical description of AMD, see our paper on arXiv. Detailed documentation of this package is available on readthedocs.

Use pip to install average-minimum-distance:

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

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

13.2 Getting started

The central functions of this package are `amd.AMD()` and `amd.PDD()`, which take a crystal and a positive integer k, returning the crystal's AMD/PDD up to k. An AMD is a 1D numpy array, whereas PDDs are 2D arrays. The AMDs or PDDs can then be passed to functions to compare them.

13.2.1 Reading crystals

The following example reads a .CIF with `amd.CifReader` and computes the AMDs (k=100):

```
import amd

# read all structures in a .cif and put their amds (k=100) in a list
reader = amd.CifReader('path/to/file.cif')
amds = [amd.AMD(crystal, 100) for crystal in reader]
```

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

A crystal can also be read from the CSD using `amd.CSDReader` (if csd-python-api is installed), or created manually.

13.2.2 Comparing AMDs or PDDs

The package includes functions for comparing sets of AMDs or PDDs.

They behave like scipy's function `scipy.distance.spatial.pdist`,

which takes a set of points and compares them pairwise, returning a *condensed distance matrix*, a 1D vector containing the distances. This vector is the upper half of the 2D distance matrix in one list, since for pairwise comparisons the matrix is symmetric. The function `amd.AMD_pdist` similarly takes a list of AMDs and compares them pairwise, returning the condensed distance matrix:

```
cdm = amd.AMD_pdist(amds)
```

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

It is preferable to store the condensed matrix, though if you want the symmetric 2D distance matrix, use scipy's `squareform`:

```
from scipy.distance.spatial import squareform
dm = squareform(cdm)
# now dm[i][j] is the AMD distance between amds[i] and amds[j].
```

The function `amd.AMD_pdist` has an equivalent for PDDs, `amd.PDD_pdist`. There are also the equivalents of `scipy.distance.spatial.cdist`, `amd.AMD_cdist` and `amd.PDD_cdist`, which take two sets and compares one vs the other, returning a 2D distance matrix.

13.3 Example: PDD-based dendrogram of a family of crystals

This example reads crystals in the DEBXIT family, compares them by PDD and plots a single linkage dendrogram:

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

crystals = list(amd.CSDReader('DEBXIT', families=True))
names = [crystal.name for crystal in crystals]
pdds = [amd.PDD(crystal, 100) for crystal in crystals]
```

(continues on next page)

(continued from previous page)

```
cdm = amd.PDD_pdist(pdds)
Z = hierarchy.linkage(cdm, 'single')
dn = hierarchy.dendrogram(Z, labels=names)
plt.show()
```

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

Here is an example showing how to read two sets of crystals from .CIFs `set1.cif` and `set2.cif` and find the 10 nearest PDD-neighbours in set 2 for every crystal in set 1. This can be done with the handy function `amd.neighbours_from_distance_matrix`, which also accepts condensed distance matrices.

```
import amd

n = 10
k = 100

set1 = list(amd.CifReader('set1.cif'))
set2 = list(amd.CifReader('set2.cif'))

set1_pdds = [amd.PDD(s, k) for s in set1]
set2_pdds = [amd.PDD(s, k) for s in set2]

dm = amd.PDD_cdist(set1_pdds, set2_pdds)

# amd.neighbours_from_distance_matrix calculates nearest neighbours for you
# nn_dists[i][j] = distance from set1[i] to its (j+1)st nearest neighbour in set2
# nn_inds[i][j] = index of set1[i]'s (j+1)st nearest neighbour in set2
# it's (j+1)st as index 0 refers to the first nearest neighbour
nn_dists, nn_inds = amd.neighbours_from_distance_matrix(n, dm)

# now to print the names of these nearest neighbours and their distances:
set1_names = [s.name for s in set1]
set2_names = [s.name for s in set2]

for i in range(len(set1)):
    print('neighbours of', set1_names[i])
    for j in range(n):
        jth_nn_index = nn_inds[i][j]
        print('neighbour', j+1, set2_names[jth_nn_index], 'dist:', nn_dists[i][j])
```

13.5 Cite us

The arXiv paper for this package is here. Use the following bib reference to cite us:

@**article{widdowson2020average,** title={Average Minimum Distances of periodic point sets}, author={Daniel Widdowson and Marco Mosca and Angeles Pulido and Vitaliy Kurlin and Andrew Cooper}, journal={arXiv:2009.02488}, year={2020}

}

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

R

S

T

U

W