average-minimum-distance

Release 1.1.8

Daniel Widdowson

Apr 16, 2022

COMMON TASKS

Read below to get started with amd, or follow these links 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'))

# Use folder=True to read all cifs in a folder
crystals = list(amd.CifReader(folder_path, folder=True))

# if you need the file names as well
import os
crystals = [amd.CifReader(os.path.join(folder_path, file)).read_one(), file
            for file in os.listdir(folder_path)]
```

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',                  # path to file/folder
              reader='ase',                # backend cif parser
              remove_hydrogens=False,      # remove H/D
              disorder='skip',             # handling disorder
              heaviest_component=False,    # just keep the heaviest component in asym␣
↪unit
              folder=False)                # if path is to a folder
```

- `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'`.
- `folder` will read all crystals from files in a folder. If true, the sets yielded will have a tag/attribute 'filename'.

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)
```

- 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'.

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. The readers in *io* yield *periodicset.PeriodicSet* objects which represent the crystals, and the `io.SetWriter` and `io.SetReader` are for writing and reading these crystals to a compressed hdf5 file.

```
# write the crystals and their AMDs to a file using the crystal's .tags
# amd.SetReader and amd.SetWriter can both be used in context managers
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 Fingerprints 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).

6.2 Simplex-wise distance distributions

As the name suggests

6.3 Inverse design

It is possible to reconstruct a periodic set up to isometry from its PDD if the periodic set satisfies certain conditions (a 'general position') and the PDD has enough columns. This is implemented via the functions *calculate.PDD_reconstructable()*, which returns the PDD of a periodic set with enough columns, and `reconstruct.reconstruct()` which returns the motif given the PDD and unit cell.

CHAPTER

SEVEN

AMD.CALCULATE MODULE

Functions for calculating AMDs and PDDs (and SDDs) of periodic and finite sets.

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

The AMD up to *k* of a periodic set.

Parameters

- **periodic_set** (*periodicset.PeriodicSet* or tuple of ndarrays) – A periodic set represented by a *periodicset.PeriodicSet* 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*, *lexsort: bool = True*, *collapse: bool = True*, *collapse_tol: float = 0.0001*) → numpy.ndarray

The PDD up to *k* of a periodic set.

Parameters

- **periodic_set** (*periodicset.PperiodicSet* or tuple of ndarrays) – A periodic set represented by a *periodicset.PperiodicSet* or by a tuple (motif, cell) with coordinates in Cartesian form.
- **k** (*int*) – Number of columns in the PDD (the returned matrix has an additional first column containing weights).
- **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'):
    # 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 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.**AMD_finite**(*motif: numpy.ndarray*) → numpy.ndarray

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

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

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

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)
- **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.PDD_finite(trapezium)
kite_pdd = amd.PDD_finite(kite)

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

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

The SSD (simplex-wise distance distribution) of a finite point set, with *len(motif) - 1* columns. The SDD with order h considers h-sized collection of points in the motif; the first-order SDD is equivalent to the PDD for finite sets.

Parameters

- **motif** (*ndarray*) – Cartesian coordinates of points in a set. Shape (n_points, dimensions)
- **order** (*int*) – Order of the SDD, default 1. See papers for a description of higher-order SDDs.
- **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 The h-order SDD of `motif`. A tuple of 3 arrays is returned, `weights`, `dist` and `sdd`. If order=1, dist is None.

Return type tuple of ndarrays

Examples

Find the SDD of the 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_sdd = amd.SDD(trapezium, order=2)
kite_sdd = amd.SDD(kite)
```

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* or tuple of ndarrays) – A periodic set represented by a *periodicset.PeriodicSet* 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'):
    # 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.**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 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

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()*.

These intermediate *periodicset.PeriodicSet* representations can be written to a .hdf5 file with `SetWriter`, which can be read back with `SetReader`. This is much faster than rereading a .CIF and recomputing invariants.

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

Bases: `amd._reader._Reader`

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

Examples

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

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

# If a folder has several .CIFs each with one crystal, use
structures = list(amd.CifReader('path/to/folder', folder=True))

# Make list of AMDs (with 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*, *extract_data=None*, *include_if=None*)

Bases: `amd._reader._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 by name 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.

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

ccdc.crystal.Crystal –> amd.periodicset.PeriodicSet. Ignores disorder, missing sites/coords, checks & no options. Is a stripped-down version of the function used in CifReader.

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

ase.io.cif.CIFBlock –> amd.periodicset.PeriodicSet. Ignores disorder, missing sites/coords, checks & no options. Is a stripped-down version of the function used in CifReader.

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`

A periodic set is the mathematical representation of a crystal by putting a single point in the center of every atom. A periodic set is defined by a basis (unit cell) and collection of points (motif) which repeats according to the basis. Has attributes motif, cell and name (which can be `None`).

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

copy()

Return copy of the PeriodicSet.

astype(*dtype*)

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

CHAPTER

ELEVEN

AMD.UTILS MODULE

Helpful utility functions.

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

Diameter of a unit cell in 3 or fewer dimensions.

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

Simplified version of function from ase.geometry. 3D unit cell parameters a,b,c,,, –> cell as 3x3 ndarray.

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

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

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

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*)

Random unit cell.

class amd.utils.**ETA**(*to_do*, *update_rate=100*)

Bases: object

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

update()

Call when one item is finished.

CHAPTER

TWELVE

AVERAGE-MINIMUM-DISTANCE: ISOMETRICALLY INVARIANT CRYSTAL FINGERPRINTS

Implements fingerprints (*isometry invariants*) of crystals based on geometry: average minimum distances (AMD) and point-wise distance distributions (PDD). Includes .cif reading tools.

- **Papers:** https://doi.org/10.46793/match.87-3.529W or on arXiv at https://arxiv.org/abs/2009.02488
- **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

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

12.1 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`.

12.2 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.

12.2.1 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 and handling disorder. 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.

12.2.2 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.

12.3 Example: PDD-based dendrogram of crystals in a .CIF

This example reads crystals from a .CIF, 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.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

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.

```
import numpy as np
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)

# the following uses np.argpartiton (like argsort but not for the whole list)
# and np.take_along_axis to find nearest neighbours of each item given the
# distance matrix.
# 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_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)

# 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]
```

(continues on next page)

(continued from previous page)

```
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])
```

12.5 Cite us

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

```
@article{amd2022,
  title = {Average Minimum Distances of periodic point sets - foundational invariants␣
↪for mapping all periodic crystals},
  author = {Daniel Widdowson and Marco M Mosca and Angeles Pulido and Vitaliy Kurlin and␣
↪Andrew I Cooper},
  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}
}
```

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX