e3fp Documentation

Release 1.2.4

Seth Axen

Jun 04, 2022

CONTENTS

Release
1.2.4

Date
Jun 04, 2022

CHAPTER

ONE

CONTENTS

1.1 Overview of E3FP

1.1.1 Introduction

The Extended 3-Dimensional FingerPrint (E3FP)[1] is a 3D molecular fingerprinting method inspired by Extended Connectivity FingerPrints (ECFP)[2], integrating tightly with the RDKit. It is developed by the Keiser Lab at UCSF and maintained primarily by Seth Axen.

For a thorough description of E3FP, please consult the original paper[1] and paper repository or *Usage and Examples*.

Documentation is hosted by ReadTheDocs.

1.1.2 Contributing

Development occurs on GitHub. Contributions, feature requests, and bug reports are greatly appreciated. Please consult the issue tracker.

1.1.3 License

E3FP is released under the GNU Lesser General Public License version 3.0 (LGPLv3).

Briefly, this means E3FP can be used in any manner without modification, with proper attribution. However, if the source code is modified for an application, this modified source must also be released under LGPLv3 so that the community may benefit.

[1] Axen SD, Huang XP, Caceres EL, Gendelev L, Roth BL, Keiser MJ. A Simple Representation Of Three-Dimensional Molecular Structure. *J. Med. Chem.* **60** (17): 7393–7409 (2017). doi 10.1021/acs.jmedchem.7b00696 bioRxiv 136705 F1000Prime RECOMMENDED

[2] Rogers D & Hahn M. Extended-connectivity fingerprints. *J. Chem. Inf. Model.* **50**: 742-54 (2010). doi 10.1021/ci100050t

1.1.4 Citing E3FP

To cite E3FP, please reference the original paper[1].

References

1.2 Setup and Installation

1.2.1 Dependencies

E3FP is compatible with Python 3.x. It additionally has the following dependencies:

Required

- NumPy
- SciPy
- RDKit
- mmh3
- python_utilities
- smart_open

Optional

The following packages are required for the specified features:

- parallelization:
  - mpi4py
- molecular standardisation:
  - standardiser
- protonation states:
  - cxcalc
- storing conformer energies:
  - h5py

1.2.2 Installation

The following installation approaches are listed in order of recommendation. All but the first of these approaches requires a prior installation of RDKit.

Option 1: Install with Conda

E3FP is on the Anaconda distribution. Conda is a cross-platform package manager. This approach is highly recommended as it installs *all* required packages.

1. Install with

```
$ conda create -c conda-forge --name e3fp_env e3fp
$ conda activate e3fp_env
```

2. To install the optional Python dependencies, run

```
$ conda install -c conda-forge mpi4py h5py standardiser
```

To get the latest version of E3FP, follow *Option 3: Clone the Repository*.

Option 2: Install with Pip

1. Install with

```
$ pip install e3fp
```

2. To install the optional Python dependencies, run

```
$ pip install mpi4py h5py standardiser
```

Option 3: Clone the Repository

1. Download this repository to your machine.
   - Clone this repository to your machine with

```
$ git clone https://github.com/keiserlab/e3fp.git
$ cd e3fp
```

   - OR download an archive by navigating to the repository and clicking "Download ZIP". Extract the archive.
2. Install the optional dependencies and any required ones using pip or conda.

Note: The easiest way to install the dependencies is with

```
$ conda env create --name e3fp_env --file environment.yml
$ conda activate e3fp_env
```

---

3. Install with

```
$ python setup.py build_ext --inplace
$ python setup.py install
```

1.2.3 Testing

After installation, it is recommended to run all tests with `pytest`. After running `pip install pytest` or `conda install -c conda-forge pytest`, run

```
$ pytest e3fp
```

1.3 Usage and Examples

To facilitate flexible use of the E3FP package, we provide multiple interfaces for performing the same tasks. We have organized these below in the order in which we expect them to be most of use to the average user.

1.3.1 Configuration

E3FP configurational parameters are stored in the widely used INI file format. These may be passed to *Command Line Interface* programs or parsed to Python dicts for *Pipeline Methods* or other lower-level functions.

Loading Default Parameters

The below example shows all default parameters, accessed via the *e3fp.config* module.

Listing 1: defaults.cfg

```
[preprocessing]
standardise = False
protonate = False

[conformer_generation]
num_conf = -1
first = -1
pool_multiplier = 1
rmsd_cutoff = 0.5
max_energy_diff = None
forcefield = uff
out_dir = conformers
compress = 2
seed = -1

; Optimized parameters used in
; Axen et al. 2017
[fingerprinting]
bits = 1024
level = 5
first = 3
radius_multiplier = 1.718
stereo = True
counts = False
include_disconnected = True
rdkit_invariants = False
```

(continues on next page)

(continued from previous page)

```
remove_duplicate_substructs = True
exclude_floating = True
```

`configparser` is used internally to parse and store these config parameters.

```
>>> from e3fp.config.params import default_params
>>> default_params
<ConfigParser.ConfigParser instance at 0x...>
>>> print(default_params.sections())
['preprocessing', 'conformer_generation', 'fingerprinting']
>>> default_params.items('fingerprinting')
[('bits', '1024'), ('level', '5'), ('first', '3'), ('radius_multiplier', '1.718'), (
↪'stereo', 'True'), ('counts', 'False'), ('include_disconnected', 'True'), ('rdkit_
↪invariants', 'False'), ('merge_duplicate_substructs', 'True'), ('exclude_floating',
↪'True')]
```

Parsing User-Provided Parameters

A user may provide a custom config file.

Listing 2: new_params.cfg

```
[conformer_generation]
first = 10

[fingerprinting]
bits = 4096
first = 10
```

```
>>> from e3fp.config.params import read_params
>>> config = read_params("source/examples/data/new_params.cfg")
>>> config.items('fingerprinting')
[('bits', '4096'), ('first', '10')]
```

When passing these parameters to any downstream methods, default options will be used except where these options are specified.

Converting Parameters to Argument Dicts

To pass the parameters to Python methods for fingerprinting and conformer generation, we need to convert them to Python dicts.

```
>>> from e3fp.pipeline import params_to_dicts
>>> confgen_params, fprint_params = params_to_dicts(config)
>>> fprint_params
{'bits': 4096, 'first': 10}
```

1.3.2 Command Line Interface

Command line interfaces (CLI) are provided for the two most common tasks: conformer generation and fingerprinting. At the moment, using the CLI requires *downloading the E3FP source*.

In the below examples, we assume the E3FP repository is located at `$E3FP_REPO`.

Conformer Generation CLI

To see all available options, run

```
$ python $E3FP_REPO/e3fp/conformer/generate.py --help
usage: Generate conformers from mol2 or SMILES [-h] [-m MOL2 [MOL2 ...]]
                                               [-s SMILES [SMILES ...]]
                                               [--standardise STANDARDISE]
                                               [-n NUM_CONF] [--first FIRST]
                                               [--pool_multiplier POOL_MULTIPLIER]
                                               [-r RMSD_CUTOFF]
                                               [-e MAX_ENERGY_DIFF]
                                               [-f {uff,mmff94,mmff94s}]
                                               [--seed SEED] [-o OUT_DIR]
                                               [-C {0,1,2,None}] [-O]
                                               [--values_file VALUES_FILE]
                                               [--prioritize]
                                               [--params PARAMS] [-l LOG]
                                               [-p NUM_PROC]
                                               [--parallel_mode {mpi,processes,threads,
↪serial}]
                                               [-v]

optional arguments:
  -h, --help            show this help message and exit
  -m MOL2 [MOL2 ...], --mol2 MOL2 [MOL2 ...]
                        Path to mol2 file(s), each with one molecule.
                        (default: None)
  -s SMILES [SMILES ...], --smiles SMILES [SMILES ...]
                        Path to file(s) with SMILES and name. (space-
                        separated) (default: None)
  --standardise STANDARDISE
                        Clean molecules before generating conformers by
                        standardisation. (default: False)
  -n NUM_CONF, --num_conf NUM_CONF
                        Set single number of conformers to use. -1 results in
                        auto choosing. (default: -1)
  --first FIRST         Set maximum number of first conformers to accept.
                        Conformer generation is unaffected, except it may
                        terminate early when this number of conformers is
                        reached. (default: -1)
  --pool_multiplier POOL_MULTIPLIER
                        Factor to multiply `num_conf` by to generate
                        conformers. Results are then pruned to `num_conf`.
                        (default: 1)
  -r RMSD_CUTOFF, --rmsd_cutoff RMSD_CUTOFF
```

(continues on next page)

(continued from previous page)

```
                        Choose RMSD cutoff between conformers (default: 0.5)
  -e MAX_ENERGY_DIFF, --max_energy_diff MAX_ENERGY_DIFF
                        Maximum energy difference between lowest energy
                        conformer and any accepted conformer. (default: None)
  -f {uff,mmff94,mmff94s}, --forcefield {uff,mmff94,mmff94s}
                        Choose forcefield for minimization. (default: uff)
  --seed SEED           Random seed for conformer generation. (default: -1)
  -o OUT_DIR, --out_dir OUT_DIR
                        Directory to save conformers. (default: conformers)
  -C {0,1,2,None}, --compress {0,1,2,None}
                        Compression to use for SDF files. None and 0 default
                        to uncompressed ".sdf". 1 and 2 result in gzipped and
                        bzipped SDF files, respectively. (default: 2)
  -O, --overwrite       Overwrite existing conformer files. (default: False)
  --values_file VALUES_FILE
                        Save RMSDs and energies to specified hdf5 file.
                        (default: None)
  --prioritize          Prioritize likely fast molecules first. (default:
                        False)
  --params PARAMS       INI formatted file with parameters. If provided, all
                        parameters controlling conformer generation are
                        ignored. (default: None)
  -l LOG, --log LOG     Generate logfile. (default: None)
  -p NUM_PROC, --num_proc NUM_PROC
                        Set number of processors to use. (default: None)
  --parallel_mode {mpi,processes,threads,serial}
                        Set number of processors to use. (default: None)
  -v, --verbose         Run with extra verbosity. (default: False)
```

We will generate conformers for the molecule whose SMILES string is defined in `caffeine.smi`.

Listing 3: caffeine.smi

```
CN1C=NC2=C1C(=O)N(C(=O)N2C)C caffeine
```

The below example generates at most 3 conformers for this molecule.

```
$ python $E3FP_REPO/e3fp/conformer/generate.py -s caffeine.smi --num_conf 3 -o ./
2017-07-17 00:11:05,743|WARNING|Only 1 processes available. 'mpi' mode not available.
2017-07-17 00:11:05,748|INFO|num_proc is not specified. 'processes' mode will use all 8␣
↪processes
2017-07-17 00:11:05,748|INFO|Parallelizer initialized with mode 'processes' and 8␣
↪processors.
2017-07-17 00:11:05,748|INFO|Input type: Detected SMILES file(s)
2017-07-17 00:11:05,748|INFO|Input file number: 1
2017-07-17 00:11:05,748|INFO|Parallel Type: processes
2017-07-17 00:11:05,748|INFO|Out Directory: ./
2017-07-17 00:11:05,749|INFO|Overwrite Existing Files: False
2017-07-17 00:11:05,749|INFO|Target Conformer Number: 3
2017-07-17 00:11:05,749|INFO|First Conformers Number: all
2017-07-17 00:11:05,749|INFO|Pool Multiplier: 1
2017-07-17 00:11:05,749|INFO|RMSD Cutoff: 0.5
```

(continues on next page)

(continued from previous page)

```
2017-07-17 00:11:05,749|INFO|Maximum Energy Difference: None
2017-07-17 00:11:05,749|INFO|Forcefield: UFF
2017-07-17 00:11:05,749|INFO|Starting.
2017-07-17 00:11:05,779|INFO|Generating conformers for caffeine.
2017-07-17 00:11:05,823|INFO|Generated 1 conformers for caffeine.
2017-07-17 00:11:05,829|INFO|Saved conformers for caffeine to ./caffeine.sdf.bz2.
```

The result is a multi-conformer SDF file called `caffeine.sdf.bz2` in the current directory.

Fingerprinting CLI

To see all available options, run

```
$ python $E3FP_REPO/e3fp/fingerprint/generate.py --help
usage: Generate E3FP fingerprints from SDF files. [-h] [-b BITS]
                                                  [--first FIRST] [-m LEVEL]
                                                  [-r RADIUS_MULTIPLIER]
                                                  [--stereo STEREO]
                                                  [--counts COUNTS]
                                                  [--params PARAMS]
                                                  [-o OUT_DIR_BASE]
                                                  [--out_ext {.fp.pkl,.fp.gz,.fp.bz2}]
                                                  [-d DB_FILE] [--all_iters]
                                                  [-O] [-l LOG] [-p NUM_PROC]
                                                  [--parallel_mode {mpi,processes,
↪threads,serial}]
                                                  [-v]
                                                  sdf_files [sdf_files ...]

positional arguments:
  sdf_files             Path to SDF file(s), each with one molecule and
                        multiple conformers.

optional arguments:
  -h, --help            show this help message and exit
  -b BITS, --bits BITS  Set number of bits for final folded fingerprint. If -1
                        or None, unfolded (2^32-bit) fingerprints are
                        generated. (default: 4294967296)
  --first FIRST         Set maximum number of first conformers for which to
                        generate fingerprints. (default: 3)
  -m LEVEL, --level LEVEL, --max_iterations LEVEL
                        Maximum number of iterations for fingerprint
                        generation. If -1, fingerprinting is run until
                        termination, and `all_iters` is set to False.
                        (default: 5)
  -r RADIUS_MULTIPLIER, --radius_multiplier RADIUS_MULTIPLIER, --shell_radius RADIUS_
↪MULTIPLIER
                        Distance to increment shell radius at around each
                        atom, starting at 0.0. (default: 1.718)
  --stereo STEREO       Differentiate by stereochemistry. (default: True)
  --counts COUNTS       Store counts-based E3FC instead of default bit-based.
```

(continues on next page)

(continued from previous page)

```
                        (default: False)
  --params PARAMS       INI formatted file with parameters. If provided, all
                        parameters controlling conformer generation are
                        ignored. (default: None)
  -o OUT_DIR_BASE, --out_dir_base OUT_DIR_BASE
                        Basename for output directory to save fingerprints.
                        Iteration number is appended to basename. (default:
                        None)
  --out_ext {.fp.pkl,.fp.gz,.fp.bz2}
                        Extension for fingerprint pickles. (default: .fp.bz2)
  -d DB_FILE, --db_file DB_FILE
                        Output file containing FingerprintDatabase object
                        (default: fingerprints.fpz)
  --all_iters           Save fingerprints from all iterations to file(s).
                        (default: False)
  -O, --overwrite       Overwrite existing file(s). (default: False)
  -l LOG, --log LOG     Log filename. (default: None)
  -p NUM_PROC, --num_proc NUM_PROC
                        Set number of processors to use. (default: None)
  --parallel_mode {mpi,processes,threads,serial}
                        Set parallelization mode to use. (default: None)
  -v, --verbose         Run with extra verbosity. (default: False)
```

To continue the above example, we will fingerprint our caffeine conformers.

```
$ python $E3FP_REPO/e3fp/fingerprint/generate.py caffeine.sdf.bz2 --bits 1024
2017-07-17 00:12:33,797|WARNING|Only 1 processes available. 'mpi' mode not available.
2017-07-17 00:12:33,801|INFO|num_proc is not specified. 'processes' mode will use all 8␣
↪processes
2017-07-17 00:12:33,801|INFO|Parallelizer initialized with mode 'processes' and 8␣
↪processors.
2017-07-17 00:12:33,801|INFO|Initializing E3FP generation.
2017-07-17 00:12:33,801|INFO|Getting SDF files
2017-07-17 00:12:33,801|INFO|SDF File Number: 1
2017-07-17 00:12:33,802|INFO|Database File: fingerprints.fpz
2017-07-17 00:12:33,802|INFO|Max First Conformers: 3
2017-07-17 00:12:33,802|INFO|Bits: 1024
2017-07-17 00:12:33,802|INFO|Level/Max Iterations: 5
2017-07-17 00:12:33,802|INFO|Shell Radius Multiplier: 1.718
2017-07-17 00:12:33,802|INFO|Stereo Mode: True
2017-07-17 00:12:33,802|INFO|Connected-only mode: on
2017-07-17 00:12:33,802|INFO|Invariant type: Daylight
2017-07-17 00:12:33,802|INFO|Parallel Mode: processes
2017-07-17 00:12:33,802|INFO|Starting
2017-07-17 00:12:33,829|INFO|Generating fingerprints for caffeine.
2017-07-17 00:12:33,935|INFO|Generated 1 fingerprints for caffeine.
2017-07-17 00:12:34,011|INFO|Saved FingerprintDatabase with fingerprints to fingerprints.
↪fpz
```

The result is a file `fingerprints.fpz` containing a *FingerprintDatabase*. To use such a database, consult *Fingerprint Storage*.

1.3.3 Pipeline Methods

E3FP can be easily plugged into an existing pipeline using the methods in the *e3fp.pipeline* module. Each of these methods wraps functionality in other modules for generating various outputs from inputs and specified options.

Note: As fingerprinting many molecules is embarrassingly parallel, we highly recommend employing a parallelization strategy. We use our own python_utilities package.

First we must choose configuration options. See *Configuration* for detailed instructions. Here we will use defaults for all but a few options.

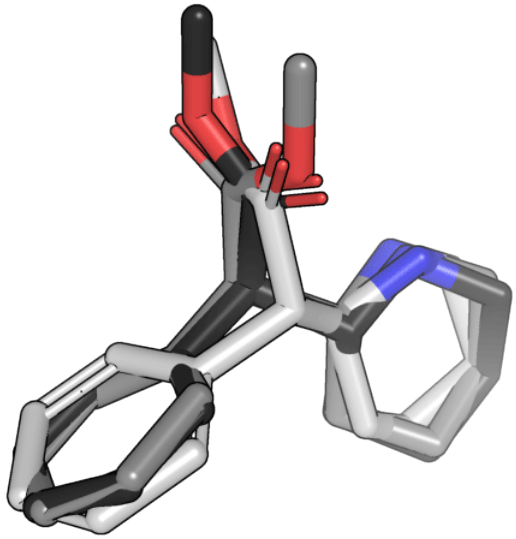
```
>>> fprint_params = {'bits': 4096, 'radius_multiplier': 1.5, 'rdkit_invariants': True}
>>> confgen_params = {'max_energy_diff': 20.0, 'first': 3}
>>> smiles = "COC(=O)C(C1CCCCN1)C2=CC=CC=C2"
```

Generating Conformers from SMILES

The following code snippet generates a multi-conformer molecule:

```
>>> from e3fp.pipeline import confs_from_smiles
>>> mol = confs_from_smiles(smiles, "ritalin", confgen_params=confgen_params)
>>> mol.GetNumConformers()
3
```

This produces the following conformers:



Generating Fingerprints from Conformers

```
>>> from e3fp.pipeline import fprints_from_mol
>>> fprints = fprints_from_mol(mol, fprint_params=fprint_params)
>>> len(fprints)
3
>>> fprints[0]
Fingerprint(indices=array([188, 224, ..., 3775, 4053]), level=5, bits=4096, name=ritalin_
↪0)
>>> fprints[1]
Fingerprint(indices=array([125, 188, ..., 3693, 4053]), level=5, bits=4096, name=ritalin_
↪1)
>>> fprints[2]
Fingerprint(indices=array([188, 206, ..., 3743, 4053]), level=5, bits=4096, name=ritalin_
↪2)
```

Generating Fingerprints from SMILES

```
>>> from e3fp.pipeline import fprints_from_smiles
>>> fprints = fprints_from_smiles(smiles, "ritalin", confgen_params=confgen_params,␣
↪fprint_params=fprint_params)
>>> fprints[0]
Fingerprint(indices=array([188, 224, ..., 3775, 4053]), level=5, bits=4096, name=ritalin_
↪0)
```

Parallel Fingerprinting

The following script demonstrates use of python_utilities for fingerprinting all SDF files in a directory in parallel. This essentially is the same as the *Fingerprinting CLI*, albeit with a less convenient interface.

```
>>> from glob import glob
>>> from python_utilities.parallel import Parallelizer
>>> from e3fp.conformer.util import smiles_to_dict
>>> smiles_dict = smiles_to_dict(smiles_file)
>>> print(smiles_dict)
{'CHEMBL1643866': 'CCCC[C@H](CN(O)C=O)C(=O)[C@@H](NC(=O)C(C)C)C(C)C', ...}
>>> len(smiles_dict)
10
>>> smiles_iter = ((smiles, name) for name, smiles in smiles_dict.items())
>>> kwargs = {"confgen_params": confgen_params, "fprint_params": fprint_params}
>>> parallelizer = Parallelizer(parallel_mode="processes")
>>> fprints_list = parallelizer.run(fprints_from_smiles, smiles_iter, kwargs=kwargs)
>>> len(fprints_list)
10
```

For all pipeline methods, please see the *e3fp.pipeline* module API.

1.3.4 Using Fingerprints

While molecular fingerprints are widely used, few packages provide simple interfaces for working with them and interfacing with machine learning packages. E3FP provides a number of general utility classes and methods for doing precisely this.

Fingerprints

The simplest interface for molecular fingerprints are through three classes in *e3fp.fingerprint.fprint*:

Fingerprint
: a fingerprint with "on" bits

CountFingerprint
: a fingerprint with counts for each "on" bit

FloatFingerprint
: a fingerprint with float values for each "on" bit, generated for example by averaging conformer fingerprints.

In addition to storing "on" indices and, for the latter two, corresponding values, they store fingerprint properties, such as name, level, and any arbitrary property. They also provide simple interfaces for fingerprint comparison, some basic processing, and comparison.

Note: Many of these operations are more efficient when operating on a *FingerprintDatabase*. See *Fingerprint Storage* for more information.

In the below examples, we will focus on *Fingerprint* and *CountFingerprint*. First, we execute the necessary imports.

```
>>> from e3fp.fingerprint.fprint import Fingerprint, CountFingerprint
>>> import numpy as np
```

See also:

Fingerprint Storage, *Fingerprint Comparison*

Creation and Conversion

Here we create a bit-fingerprint with random "on" indices.

```
>>> bits = 2**32
>>> indices = np.sort(np.random.randint(0, bits, 30))
>>> indices
array([ 243580376,  305097549, ..., 3975407269, 4138900056])
>>> fp1 = Fingerprint(indices, bits=bits, level=0)
>>> fp1
Fingerprint(indices=array([243580376, ..., 4138900056]), level=0, bits=4294967296,␣
↪name=None)
```

This fingerprint is extremely sparse

```
>>> fp1.bit_count
30
```

(continues on next page)

(continued from previous page)

```
>>> fp1.density
6.984919309616089e-09
```

We can therefore "fold" the fingerprint through a series of bitwise "OR" operations on halves of the sparse vector until it is of a specified length, with minimal collision of bits.

```
>>> fp_folded = fp1.fold(1024)
>>> fp_folded
Fingerprint(indices=array([9, 70, ..., 845, 849]), level=0, bits=1024, name=None)
>>> fp_folded.bit_count
29
>>> fp_folded.density
0.0283203125
```

A *CountFingerprint* may be created by also providing a dictionary matching indices with nonzero counts to the counts.

```
>>> indices2 = np.sort(np.random.randint(0, bits, 60))
>>> counts = dict(zip(indices2, np.random.randint(1, 10, indices2.size)))
>>> counts
{80701568: 8, 580757632: 7, ..., 800291326: 5, 4057322111: 7}
>>> cfp1 = CountFingerprint(counts=counts, bits=bits, level=0)
>>> cfp1
CountFingerprint(counts={80701568: 8, 580757632: 7, ..., 3342157822: 2, 4057322111: 7},␣
↪level=0, bits=4294967296, name=None)
```

Unlike folding a bit fingerprint, by default, folding a count fingerprint performs a "SUM" operation on colliding counts.

```
>>> cfp1.bit_count
60
>>> cfp_folded = cfp1.fold(1024)
>>> cfp_folded
CountFingerprint(counts={128: 15, 257: 4, ..., 1022: 2, 639: 7}, level=0, bits=1024,␣
↪name=None)
>>> cfp_folded.bit_count
57
```

It is trivial to interconvert the fingerprints.

```
>>> cfp_folded2 = CountFingerprint.from_fingerprint(fp_folded)
>>> cfp_folded2
CountFingerprint(counts={9: 1, 87: 1, ..., 629: 1, 763: 1}, level=0, bits=1024,␣
↪name=None)
>>> cfp_folded2.indices[:5]
array([  9,  70,  72,  87, 174])
>>> fp_folded.indices[:5]
array([  9,  70,  72,  87, 174])
```

RDKit Morgan fingerprints (analogous to ECFP) may easily be converted to a *Fingerprint*.

```
>>> from rdkit import Chem
>>> from rdkit.Chem import AllChem
>>> mol = Chem.MolFromSmiles('Cc1ccccc1')
```

(continues on next page)

(continued from previous page)

```
>>> mfp = AllChem.GetMorganFingerprintAsBitVect(mol, 2)
>>> mfp
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> Fingerprint.from_rdkit(mfp)
Fingerprint(indices=array([389, 1055, ..., 1873, 1920]), level=-1, bits=2048, name=None)
```

Likewise, *Fingerprint* can be easily converted to a NumPy ndarray or SciPy sparse matrix.

```
>>> fp_folded.to_vector()
<1x1024 sparse matrix of type '<type 'numpy.bool_'>'
...with 29 stored elements in Compressed Sparse Row format>
>>> fp_folded.to_vector(sparse=False)
array([False, False, False, ..., False, False, False], dtype=bool)
>>> np.where(fp_folded.to_vector(sparse=False))[0]
array([  9,  70,  72,  87, ...])
>>> cfp_folded.to_vector(sparse=False)
array([0, 0, 0, ..., 0, 2, 0], dtype=uint16)
>>> cfp_folded.to_vector(sparse=False).sum()
252
```

Algebra

Basic algebraic functions may be performed on fingerprints. If either fingerprint is a bit fingerprint, all algebraic functions are bit-wise. The following bit-wise operations are supported:

Equality

```
>>> fp1 = Fingerprint([0, 1, 6, 8, 12], bits=16)
>>> fp2 = Fingerprint([1, 2, 4, 8, 11, 12], bits=16)
>>> fp1 == fp2
False
>>> fp1_copy = Fingerprint.from_fingerprint(fp1)
>>> fp1 == fp1_copy
True
>>> fp1_copy.level = 5
>>> fp1 == fp1_copy
False
```

Union/OR

```
>>> fp1 + fp2
Fingerprint(indices=array([0, 1, 2, 4, 6, 8, 11, 12]), level=-1, bits=16, name=None)
>>> fp1 | fp2
Fingerprint(indices=array([0, 1, 2, 4, 6, 8, 11, 12]), level=-1, bits=16, name=None)
```

Intersection/AND

```
>>> fp1 & fp2
Fingerprint(indices=array([1, 8, 12]), level=-1, bits=16, name=None)
```

Difference/AND NOT

```
>>> fp1 - fp2
Fingerprint(indices=array([0, 6]), level=-1, bits=16, name=None)
>>> fp2 - fp1
Fingerprint(indices=array([2, 4, 11]), level=-1, bits=16, name=None)
```

XOR

```
>>> fp1 ^ fp2
Fingerprint(indices=array([0, 2, 4, 6, 11]), level=-1, bits=16, name=None)
```

With count or float fingerprints, bit-wise operations are still possible, but algebraic operations are applied to counts.

```
>>> fp1 = CountFingerprint(counts={0: 3, 1: 2, 5: 1, 9: 3}, bits=16)
>>> fp2 = CountFingerprint(counts={1: 2, 5: 2, 7: 3, 10: 7}, bits=16)
>>> fp1 + fp2
CountFingerprint(counts={0: 3, 1: 4, 5: 3, 7: 3, 9: 3, 10: 7}, level=-1, bits=16,␣
↪name=None)
>>> fp1 - fp2
CountFingerprint(counts={0: 3, 1: 0, 5: -1, 7: -3, 9: 3, 10: -7}, level=-1, bits=16,␣
↪name=None)
>>> fp1 * 3
CountFingerprint(counts={0: 9, 1: 6, 5: 3, 9: 9}, level=-1, bits=16, name=None)
>>> fp1 / 2
FloatFingerprint(counts={0: 1.5, 1: 1.0, 5: 0.5, 9: 1.5}, level=-1, bits=16, name=None)
```

Finally, fingerprints may be batch added and averaged, producing either a count or float fingerprint when sensible.

```
>>> from e3fp.fingerprint.fprint import add, mean
>>> fps = [Fingerprint(np.random.randint(0, 32, 8), bits=32) for i in range(100)]
>>> add(fps)
CountFingerprint(counts={0: 23, 1: 23, ..., 30: 20, 31: 14}, level=-1, bits=32,␣
↪name=None)
>>> mean(fps)
FloatFingerprint(counts={0: 0.23, 1: 0.23, ..., 30: 0.2, 31: 0.14}, level=-1, bits=32,␣
↪name=None)
```

Fingerprint Storage

The most efficient way to store and interact with fingerprints is through the *e3fp.fingerprint.db.FingerprintDatabase* class. This class wraps a matrix with sparse rows (`scipy.sparse.csr_matrix`), where each row is a fingerprint. This enables rapid I/O of the database while also minimizing the memory footprint. Accessing the underlying sparse representation with the .FingerprintDatabase.array attribute is convenient for machine learning purposes, while the database class itself provides several useful functions.

Note: We strongly recommend upgrading to at least SciPy v1.0.0 when working with large fingerprint databases, as old versions are much slower and have several bugs for database loading.

Database I/O and Indexing

See the full *e3fp.fingerprint.db.FingerprintDatabase* documentation for a description of basic database usage, attributes, and methods. Below, several additional use cases are documented.

Batch Database Operations

Due to the sparse representation of the underlying data structure, an un- folded database, a database with unfolded fingerprints does not use significantly more disk space than a database with folded fingerprints. However, it is usually necessary to fold fingerprints for machine learning tasks. The *FingerprintDatabase* does this very quickly.

```
>>> from e3fp.fingerprint.db import FingerprintDatabase
>>> from e3fp.fingerprint.fprint import Fingerprint
>>> import numpy as np
>>> db = FingerprintDatabase(fp_type=Fingerprint, name="TestDB")
>>> print(db)
FingerprintDatabase[name: TestDB, fp_type: Fingerprint, level: -1, bits: None, fp_num: 0]
>>> on_inds = [np.random.uniform(0, 2**32, size=30) for i in range(5)]
>>> fps = [Fingerprint(x, bits=2**32) for x in on_inds]
>>> db.add_fingerprints(fps)
>>> print(db)
FingerprintDatabase[name: TestDB, fp_type: Fingerprint, level: -1, bits: 4294967296, fp_
↪num: 5]
>>> db.get_density()
6.984919309616089e-09
>>> fold_db = db.fold(1024)
>>> print(fold_db)
FingerprintDatabase[name: TestDB, fp_type: Fingerprint, level: -1, bits: 1024, fp_num: 5]
>>> fold_db.get_density()
0.0287109375
```

A database can be converted to a different fingerprint type:

```
>>> from e3fp.fingerprint.fprint import CountFingerprint
>>> count_db = db.as_type(CountFingerprint)
>>> print(count_db)
FingerprintDatabase[name: TestDB, fp_type: CountFingerprint, level: -1, bits: 4294967296,
↪ fp_num: 5]
>>> count_db[0]
CountFingerprint(counts={2977004690: 1, ..., 3041471738: 1}, level=-1, bits=4294967296,␣
↪name=None)
```

The *e3fp.fingerprint.db.concat* method allows efficient joining of multiple databases.

```
>>> from e3fp.fingerprint.db import concat
>>> dbs = []
>>> for i in range(10):
...     db = FingerprintDatabase(fp_type=Fingerprint)
...     on_inds = [np.random.uniform(0, 1024, size=30) for j in range(5)]
...     fps = [Fingerprint(x, bits=2**32, name="Mol{}".format(i)) for x in on_inds]
...     db.add_fingerprints(fps)
...     dbs.append(db)
```

(continues on next page)

(continued from previous page)

```
>>> dbs[0][0]
Fingerprint(indices=array([94, 97, ..., 988, 994]), level=-1, bits=4294967296, name=Mol0)
>>> print(dbs[0])
FingerprintDatabase[name: None, fp_type: Fingerprint, level: -1, bits: 4294967296, fp_
↪num: 5]
>>> merge_db = concat(dbs)
>>> print(merge_db)
FingerprintDatabase[name: None, fp_type: Fingerprint, level: -1, bits: 4294967296, fp_
↪num: 50]
```

Database Comparison

Two databases may be compared using various metrics in *e3fp.fingerprint.metrics*. Additionally, all fingerprints in a database may be compared to each other simply by only providing a single database. See *Fingerprint Comparison* for more details.

Performing Machine Learning on the Database

The underlying sparse matrix may be passed directly to machine learning tools in any package that is compatible with SciPy sparse matrices, such as scikit-learn.

```
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(db.array, ypred)
BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None, fit_prior=True)
>>> clf.predict(db2.array)
...
```

Fingerprint Comparison

The *e3fp.fingerprint.metrics* sub-package provides several useful methods for batch comparison of fingerprints in various representations.

Fingerprint Metrics

These metrics operate directly on pairs of *Fingerprint* and *FingerprintDatabase* objects or on a combination of each. If only a single variable is specified, self-comparison is performed. The implemented methods are common functions for fingerprint similarity in the literature.

Array Metrics

To efficiently compare fingerprint databases above, we provide comparison metrics that can operate directly on the internal sparse matrix representation without the need to "densify it". We describe these here, as they have several additional features.

The array metrics implemented in *e3fp.fingerprint.metrics.array_metrics* are implemented such that they may take any combination of dense and sparse inputs. Additionally, they are designed to function as scikit-learn-compatible kernels for machine learning tasks. For example, one might perform an analysis using a support vector machine (SVM) and Tanimoto kernel.

```
>>> from sklearn.svm import SVC
>>> from e3fp.fingerprint.metrics.array_metrics import tanimoto
>>> clf = SVC(kernel=tanimoto)
>>> clf.fit(X, y)
...
>>> clf.predict(test)
...
```

Most common fingerprint comparison metrics only apply to binary fingerprints. We include several that operate equally well on count- and float-based fingerprints. For example, to our knowledge, we provide the only open source implementation of Soergel similarity, the analog to the Tanimoto coefficient for non-binary fingerprints that can efficiently operate on sparse inputs.

```
>>> from e3fp.fingerprint.metrics.array_metrics import soergel
>>> clf = SVC(kernel=soergel)
>>> clf.fit(X, y)
...
>>> clf.predict(test)
...
```

1.4 Developer Notes

We welcome contributions to E3FP! These notes are designed to help developers contribute code

1.4.1 Authoring Code

Code Formatting

E3FP's code should be *readable*. To ensure this, we rigorously follow the PEP8 style conventions and PEP257 docstring conventions, which maximize readability of the code and ease of future development. You may check your code for conformation to these conventions with the pycodestyle and pydocstyle utilities, respectively. Where the code is necessarily complicated, inline comments should reorient the reader.

Utility Methods and Classes

Three sets of utility methods and classes are provided: *e3fp.util*, *e3fp.conformer.util*, and *e3fp.fingerprint.util*. These provide general and often-used functionality in their corresponding packages. Additionally, they provide E3FP-specific errors and exceptions.

Warnings and Errors

By default, warnings in Python are silent. We therefore provide a warning base class *e3fp.util.E3FPWarning* that is not silent by default. We provide several general warnings:

E3FPDeprecationWarning
: warns when a deprecated method is called or class is instantiated.

  See also:

  Deprecation

E3FPEfficiencyWarning
: warns when a method, module version, or combination of parameters is known to be inefficient.

  Note: If possible, the warning message should advise on a more efficient approach.

E3FP-specific errors should inherit *e3fp.util.E3FPError* base class. Several fingerprinting-specific errors are defined in *e3fp.fingerprint.util*.

Deprecation

Whenever changing the interface or behavior of a user-facing method or class, it is proper to deprecate it for at least one release, so that the users have time to update their scripts accordingly. A deprecated method should providing an *e3fp.util.E3FPDeprecationWarning*, notifying the user in which release to expect the method or class to be removed, and updating the documentation accordingly. This functionality is automated with the *e3fp.util.deprecated* decorator, as shown in this example:

```
>>> import sys
>>> sys.stderr = sys.stdout
>>> from e3fp.util import deprecated
>>> @deprecated("1.1", remove_version="1.3", msg="Function no longer needed.")
... def deprecated_method():
...     """A method to demonstrate method deprecation."""
...     pass
>>> deprecated_method()
...: E3FPDeprecationWarning: Function `my_function` was deprecated in 1.1 and will be␣
↪removed in 1.3. Function no longer needed.
```

In the api documentation, the method will appear as:

deprecated_method()

Note: Deprecated in e3fp 1.1. *deprecated_method* will be removed in e3fp 1.3. Function no longer needed.

A method to demonstrate method deprecation.

Note: If no *remove_version* is specified, then the remove version defaults to the next release after deprecation. For example, if the method was deprecated in 1.1, it is by default marked for removal in 1.2.

Contributing Code

Before contributing code to E3FP, it is advisable for major modifications to submit an issue to the issue tracker to enable other developers to contribute to the design of the code and to reduce the amount of work necessary to conform the code to E3FP's standards. After writing the code, create a pull request. This is best even if you have push access to the E3FP repo, as it enables the test suite to be run on the new code prior to merging it with the remaining code base.

Writing Tests

The standard in E3FP is to commit a test for new functionality simultaneously with the new functionality or within the same pull request. While this slows development, it prevents building a large backlog of untested methods and classes.

These should ideally be unit tests, though for some complicated functionalities, such as fingerprinting, integration tests are also necessary. For these complicated functions, specific units may still be tested using `unittest.mock`. For example, `unittest.mock.patch()` may be used to force a high level method to produce a specific output. For examples, see the fingeprinting tests.

Continuous Integration

E3FP uses GitHub Actions for continuous integration. This ensures that each commit and pull request passes all tests on a variety of a systems and for all supported versions of Python. Additionally, GitHub Actions updates code coverage on Codecov and tests all usage examples in the documentation using `doctest`.

1.4.2 Documentation

In general, it is best to document the rationale and basic usage of a module, class, or method in its docstring instead of in a separate documentation file. See, for example, the docstring for *e3fp.fingerprint.db.FingerprintDatabase*. We use a variety of tools to ensure that our documentation is always up-to-date. The official documentation is hosted on ReadtheDocs and is automatically generated when new code is committed to the repository.

Documenting Code

E3FP uses NumPy's docstring conventions for all docstrings. These are parsed by Sphinx using Napoleon. All usage examples must be fully functional, as these are tested using `doctest`.

The purpose of a docstring is to explain the purpose of a class/method, any relevant implementation details, its parameters, its attributes, its outputs, and its usage. The goal is clarity. For self-evident methods with descriptive variables, a simple one- ine summary is all that is needed. For complicated use cases, often involving other methods/classes, it is better to document the usage elsewhere in the documentation.

Documentation Usage

Coming soon.

1.4.3 Releasing Code

1.5 e3fp API

1.5.1 e3fp package

Subpackages

e3fp.config package

Submodules

e3fp.config.params module

Get E3FP default parameters and read parameters from files.

Author: Seth Axen E-mail: seth.axen@gmail.com

`get_default_value`(**args*, ***kwargs*)

`get_value`(*params*, *section_name*, *param_name*, *dtype=<class 'str'>*, *auto=False*, *fallback=None*)

Get value from params with fallback.

Parameters

- **params** (*ConfigParser*) – Parameters
- **section_name** (*str*) – Name of section in *params*
- **param_name** (*str*) – Name of parameter in *section*
- **dtype** (*type, optional*) – Type to return data as.
- **auto** (*bool, optional*) – Auto-discover type of value. If provided, *dtype* is ignored.
- **fallback** (*any, optional*) – Value to return if getting value fails.

Returns

value – Value of parameter or *fallback*.

Return type

any

`params_to_sections_dict`(*params*, *auto=True*)

Get dict of sections dicts in params, with optional type discovery.

Parameters

- **params** (*str or ConfigParser*) – Params to read
- **auto** (*bool, optional*) – Auto typing of parameter values.

Returns

dict

Return type
dict matching sections to parameters to values.

`read_params`(*params=None*, *fill_defaults=False*)

Get combination of provided parameters and default parameters.

Parameters

- **params** (*str or ConfigParser, optional*) – User provided parameters as an INI file or `ConfigParser`. Any parameters provided will replace default parameters.
- **fill_defaults** (*bool, optional*) – Fill values that aren't provided with package defaults, if *params* is file.

Returns
all_params – Combination of default and user-provided parameters.

Return type
ConfigParser

`update_params`(*params_dict*, *params=None*, *section_name=None*, *fill_defaults=False*)

Set `ConfigParser` values from a sections dict.

Sections dict key must be parameter sections, and value must be dict matching parameter name to value. If existing `ConfigParser` is provided, parameter values are updated.

Parameters

- **params_dict** (*dict*) – If *section_name* is provided, dict must match parameter names to values. If *section_name* is not provided, dict key(s) must be parameter sections, and value(s) must be parameter dict.
- **params** (*ConfigParser, optional*) – Existing parameters.
- **section_name** (*str, optional*) – Name of section to which to add parameters in *params_dict*
- **fill_defaults** (*bool, optional*) – Fill values that aren't provided with package defaults, if *params* is file.

`write_params`(*params*, *params_file='params.cfg'*)

Write params to file.

Parameters

- **params** (*ConfigParser*) – Params
- **params_file** (*str*) – Params file

Module contents

e3fp.conformer package

Submodules

e3fp.conformer.generate module

Generate conformers from SMILES or MOL2 files.

Author: Seth Axen E-mail: seth.axen@gmail.com

generate_conformers(*input_mol*, *name=None*, *standardise=False*, *num_conf=- 1*, *first=- 1*, *pool_multiplier=1*, *rmsd_cutoff=0.5*, *max_energy_diff=None*, *forcefield='uff'*, *seed=- 1*, *out_file=None*, *out_dir='conformers'*, *save=False*, *compress='2'*, *overwrite=False*)

Generate and save conformers for molecules.

Parameters

- **input_mol** (*RDKit Mol*) – Mol with a single conformer from which to generate conformers.
- **name** (*str, optional*) – Name of molecule.
- **standardise** (*bool, optional*) – Standardise mol before generating conformers.
- **num_conf** (*int, optional*) – If int, this is the target number of conformations. If -1, number of conformations is automatically chosen based on number of rotatable bonds.
- **first** (*int, optional*) – Number of first conformers to return. Does not impact conformer generator process, except may terminate conformer generation early when this many of conformers have been accepted.
- **pool_multiplier** (*int, optional*) – Factor to multiply by *num_conf*. The resulting number of conformations will be generated, then pruned to *num_conf*.
- **rmsd_cutoff** (*float, optional*) – RMSD threshold above which to accept two conformations as different
- **max_energy_diff** (*float, optional*) – Maximum energy difference between lowest energy conformer and any accepted conformer.
- **forcefield** (*{'uff', 'mmff94', 'mmff94s'}, optional*) – Forcefield to use for minimization of conformers.
- **seed** (*int, optional*) – Random seed for conformer generation. If -1, the random number generator is unseeded.
- **out_file** (*str, optional*) – Filename to save output, if *save* is True. If None, filename will be *name*.sdf, optionally with a compressed extension.
- **out_dir** (*str, optional*) – Directory where output files will be saved if *save* is True.
- **save** (*bool, optional*) – Save conformers to *out_file* in *out_dir*.
- **compress** (*int, optional*) – Compression of SDF files. None: auto. Mode is chosen based on extension, defaulting to SDF. 0: File is not compressed. 1: File is gzipped (.gz) 2: File is bzipped (.bz2)
- **overwrite** (*bool, optional*) – Overwrite output files if they already exist.

Returns

- *bool* – If something went wrong, only return False. Otherwise return below.
- *tuple* – Tuple with molecule name, number of rotatable bonds, numpy array of indices of final conformations, numpy array of energies of all conformations generated, and 2D numpy array of pairwise RMSDs between final conformations.

run(*mol2=None*, *smiles=None*, *standardise=False*, *num_conf=- 1*, *first=- 1*, *pool_multiplier=1*, *rmsd_cutoff=0.5*, *max_energy_diff=None*, *forcefield='uff'*, *seed=- 1*, *params=None*, *prioritize=False*, *out_dir='conformers'*, *compress='2'*, *overwrite=False*, *values_file=None*, *log=None*, *num_proc=None*, *parallel_mode=None*, *verbose=False*)

Run conformer generation.

`values_to_hdf5`(*hdf5_buffer*, *values*)

Write specific values to *hdf5_buffer*.

Parameters

- **hdf5_buffer** (*HDF5Buffer*) – HDF5 buffer to write to.
- **values** (*tuple*) – Tuple of values to be written to buffer. Values and types should be *name* (str), *num_rotatable_bonds* (int), *target_conformers* (int), *indices* (numpy long array), *energies* (numpy float array), and *rmsd* (numpy float array). *rmsd* should be a square array with size along each dimension equal to length of *indices*.

Returns

True if success, False if not.

Return type

bool

e3fp.conformer.generator module

Conformer generation.

Author: Seth Axen E-mail: seth.axen@gmail.com

`class ConformerGenerator`(*num_conf=- 1*, *first=- 1*, *rmsd_cutoff=0.5*, *max_energy_diff=- 1.0*, *forcefield='uff'*, *pool_multiplier=1*, *seed=- 1*, *get_values=False*, *sparse_rmsd=True*, *store_energies=True*)

Bases: `object`

Generate conformers using RDKit.

1. Generate a pool of conformers.
2. Minimize conformers.
3. Filter conformers using an RMSD threshold and optional minimum energy difference.

Note that pruning is done _after_ minimization, which differs from the protocol described in the references.

References

- http://rdkit.org/docs/GettingStartedInPython.html #working-with-3d-molecules
- http://pubs.acs.org/doi/full/10.1021/ci2004658
- https://github.com/skearnes/rdkit-utils/blob/master/rdkit_utils/ conformers.py

`embed_molecule`(*mol*)

Generate conformers, possibly with pruning.

Parameters

mol (*RDKit Mol*) – Molecule.

`filter_conformers`(*mol*)

Filter conformers which do not meet an RMSD threshold.

Parameters

mol (*RDKit Mol*) – Molecule.

Returns

- *A new RDKit Mol containing the chosen conformers, sorted by*
- *increasing energy.*

`generate_conformers`(*mol*)

Generate conformers for a molecule.

Parameters
mol (*RDKit Mol*) – Molecule.

Returns
RDKit Mol

Return type
copy of the input molecule with embedded conformers

`get_conformer_energies`(*mol*)

Calculate conformer energies.

Parameters
mol (*RDKit Mol*) – Molecule.

Returns
energies – Minimized conformer energies.

Return type
array_like

`get_molecule_force_field`(*mol*, *conf_id=None*, ***kwargs*)

Get a force field for a molecule.

Parameters

- **mol** (*RDKit Mol*) – Molecule.
- **conf_id** (*int, optional*) – ID of the conformer to associate with the force field.
- ****kwargs** (*dict, optional*) – Keyword arguments for force field constructor.

`static get_num_conformers`(*mol*)

Return ideal number of conformers from rotatable bond number in model.

Parameters
mol (*Mol*) – RDKit *Mol* object for molecule

Yields
num_conf (*int*) – Target number of conformers to accept

`minimize_conformers`(*mol*)

Minimize molecule conformers.

Parameters
mol (*RDKit Mol*) – Molecule.

`static reverse_enumerate`(*iterable*)

Enumerate, but with the last result first but still numbered last.

Parameters
iterable (*some 1-D iterable*)

Returns
Reverse of *enumerate* function

Return type
iterable

e3fp.conformer.protonation module

Functions for generating protonation states of molecules.

Author: Seth Axen E-mail: seth.axen@gmail.com

smiles_dict_to_proto_smiles_dict(*in_smiles_dict*, *max_states=3*, *pka=7.4*, *dist_cutoff=20.0*, *add_missing=False*, *parallelizer=None*, *chunk_size=100*)

Generate dict of SMILES for protonated states from SMILES dict.

smiles_list_to_proto_smiles_list(*in_smiles_list*, *max_states=3*, *pka=7.4*, *dist_cutoff=20.0*)

Generate list of SMILES for protonated states from single SMILES.

smiles_to_proto_smiles(*smiles*, *mol_name*, *max_states=3*, *pka=7.4*, *dist_cutoff=20.0*)

Generate list of SMILES for protonated states from single SMILES.

This is very inefficient in batch.

e3fp.conformer.util module

Utilities for handling SMILES strings and RDKit mols and conformers.

Author: Seth Axen E-mail: seth.axen@gmail.com

class MolItemName(*mol_name=None*, *proto_state_num=None*, *conf_num=None*, *proto_delim='-'*, *conf_delim='_'*)

Bases: `object`

Class for parsing mol item names and converting to various formats.

property conf_name

copy()

classmethod from_str(*mol_item_name*, *mol_item_regex=re.compile('(?P<mol_name>.+?)(?:-(?P<proto_state_num>\\d+))?(?:_(?P<conf_num>\\d+))?$')*, *mol_item_fields=('mol_name', 'proto_state_num', 'conf_num')*, ***kwargs*)

classmethod from_tuple(*fields_tuple*)

property mol_item_name

static mol_item_name_to_dict(*mol_item_name*, *mol_item_regex=re.compile('(?P<mol_name>.+?)(?:-(?P<proto_state_num>\\d+))?(?:_(?P<conf_num>\\d+))?$')*, *mol_item_fields=('mol_name', 'proto_state_num', 'conf_num')*)

property mol_name

property proto_name

to_conf_name(*conf_num=None*, *conf_delim='_'*)

to_mol_name(*as_proto=False*)

to_proto_name(*proto_state_num=None*, *proto_delim='-'*)

to_str()

to_tuple()

class MolItemTuple(*mol_name*, *proto_state_num*, *conf_num*)

Bases: tuple

conf_num

Alias for field number 2

mol_name

Alias for field number 0

proto_state_num

Alias for field number 1

add_conformer_energies_to_mol(*mol*, *energies*)

Add conformer energies as mol property.

See discussion at https://sourceforge.net/p/rdkit/mailman/message/27547551/

dict_to_smiles(*smiles_file*, *smiles_dict*)

Write SMILES dict to file.

get_conformer_energies_from_mol(*mol*)

Get conformer energies from mol.

iter_to_smiles(*smiles_file*, *smiles_iter*)

Write iterator of (mol_name, SMILES) to file.

mol2_generator(**filenames*)

Parse name from mol2 filename and return generator.

Parameters

files (*iterable object*) – List of mol2 files, where filename should be molecule name followed by ".mol2"

Yields

tuple – *tuple* of the format (file, name).

mol_from_mol2(*mol2_file*, *name=None*, *standardise=False*)

Read a mol2 file into an RDKit PropertyMol.

Parameters

- **mol2_file** (*str*) – path to a mol2 file
- **name** (*str, optional*) – Name of molecule. If not provided, uses file basename as name
- **standardise** (*bool*) – Clean mol through standardisation

Returns

RDKit PropertyMol

Return type

Molecule.

`mol_from_sdf`(*sdf_file*, *conf_num=None*, *standardise=False*, *mode='rb'*)

Read SDF file into an RDKit *Mol* object.

Parameters

- **sdf_file** (*str*) – Path to an SDF file
- **conf_num** (*int or None, optional*) – Maximum number of conformers to read from file. Defaults to all.
- **standardise** (*bool (default False)*) – Clean mol through standardisation
- **mode** (*str (default 'rb')*) – Mode with which to open file

Returns

RDKit Mol

Return type

Mol object with each molecule in SDF file as a conformer

`mol_from_smiles`(*smiles*, *name*, *standardise=False*)

Generate a n RDKit `PropertyMol` from SMILES string.

Parameters

- **smile** (*str*) – SMILES string
- **name** (*str*) – Name of molecule
- **standardise** (*bool*) – Clean Mol through standardisation

Returns

RDKit PropertyMol

Return type

Molecule.

`mol_to_sdf`(*mol*, *out_file*, *conf_num=None*)

Write RDKit *Mol* objects to an SDF file.

Parameters

- **mol** (*RDKit Mol*) – A molecule containing 1 or more conformations to write to file.
- **out_file** (*str*) – Path to save SDF file.
- **conf_num** (*int or None, optional*) – Maximum number of conformers to save to file. Defaults to all.

`mol_to_standardised_mol`(*mol*, *name=None*)

Standardise mol(s).

`smiles_generator`(**filenames*)

Parse SMILES file(s) and yield (name, smile).

Parameters

files (*iterable object*) – List of files containing smiles. File must contain one smile per line, followed by a space and then the molecule name.

Yields

tuple – *tuple* of the format (smile, name).

`smiles_to_dict`(*smiles_file*, *unique=False*, *has_header=False*)

Read SMILES file to dict.

Module contents

e3fp.fingerprint package

Subpackages

e3fp.fingerprint.metrics package

Submodules

e3fp.fingerprint.metrics.array_metrics module

Fingerprint array comparison metrics.

Each is fully compatible with both dense and sparse inputs.

Author: Seth Axen E-mail: seth.axen@gmail.com

`cosine`(*X*, *Y=None*, *assume_binary=False*)

Compute the Cosine similarities between *X* and *Y*.

Parameters

- **X** (*array_like or sparse matrix*) – with shape (*n_fprints_X*, *n_bits*).
- **Y** (*array_like or sparse matrix, optional*) – with shape (*n_fprints_Y*, *n_bits*).
- **assume_binary** (*bool, optional*) – Assume data is binary (results in efficiency boost). If data is not binary, the result will be incorrect.

Returns
cosine

Return type
array of shape (*n_fprints_X*, *n_fprints_Y*)

See also:

dice, *soergel*, *tanimoto*

`dice`(*X*, *Y=None*)

Compute the Dice coefficients between *X* and *Y*.

Data must be binary. This is not checked.

Parameters

- **X** (*array_like or sparse matrix*) – with shape (*n_fprints_X*, *n_bits*).
- **Y** (*array_like or sparse matrix, optional*) – with shape (*n_fprints_Y*, *n_bits*).

Returns
dice

Return type
array of shape (*n_fprints_X*, *n_fprints_Y*)

See also:

cosine, *soergel*, *tanimoto*, *pearson*

pearson(*X*, *Y=None*)

Compute the Pearson correlation between *X* and *Y*.

Parameters

- **X** (*array_like or sparse matrix*) – with shape (*n_fprints_X*, *n_bits*).
- **Y** (*array_like or sparse matrix, optional*) – with shape (*n_fprints_Y*, *n_bits*).

Returns
pearson

Return type
array of shape (*n_fprints_X*, *n_fprints_Y*)

See also:

soergel
Soergel similarity for non-binary data

cosine, *dice*, *tanimoto*

soergel(*X*, *Y=None*)

Compute the Soergel similarities between *X* and *Y*.

Soergel similarity is the complement of Soergel distance and can be thought of as the analog of the Tanimoto coefficient for count/float-based data. For binary data, it is equivalent to the Tanimoto coefficient.

Parameters

- **X** (*array_like or sparse matrix*) – with shape (*n_fprints_X*, *n_bits*).
- **Y** (*array_like or sparse matrix, optional*) – with shape (*n_fprints_Y*, *n_bits*).

Returns
soergel

Return type
array of shape (*n_fprints_X*, *n_fprints_Y*)

Notes

If Numba is available, this function is jit-compiled and much more efficient.

See also:

tanimoto
A fast version of this function for binary data.

pearson
Pearson correlation, also appropriate for non-binary data.

cosine, *dice*

tanimoto(*X*, *Y=None*)

Compute the Tanimoto coefficients between *X* and *Y*.

Data must be binary. This is not checked.

Parameters

- **X** (*array_like or sparse matrix*) – with shape (*n_fprints_X*, *n_bits*).

- **Y** (*array_like or sparse matrix, optional*) – with shape (*n_fprints_Y*, *n_bits*).

Returns
tanimoto

Return type
array of shape (*n_fprints_X*, *n_fprints_Y*)

See also:

soergel
Analog to Tanimoto for non-binary data.

cosine, *dice*, *pearson*

e3fp.fingerprint.metrics.fprint_metrics module

Fingerprint comparison metrics.

Author: Seth Axen E-mail: seth.axen@gmail.com

cosine(*fp1*, *fp2*)

Calculate cosine similarity between fingerprints.

Parameters

- **fp1** (*Fingerprint*) – Fingerprint 1
- **fp2** (*Fingerprint*) – Fingerprint 2

Returns
float

Return type
Cosine similarity.

dice(*fp1*, *fp2*)

Calculate Dice coefficient between fingerprints.

Parameters

- **fp1** (*Fingerprint*) – Fingerprint 1
- **fp2** (*Fingerprint*) – Fingerprint 2

Returns
float

Return type
Dice coefficient.

distance(*fp1*, *fp2*)

Calculate Euclidean distance between fingerprints.

Parameters

- **fp1** (*Fingerprint*) – Fingerprint 1
- **fp2** (*Fingerprint*) – Fingerprint 2

Returns
float

Return type
Euclidian distance.

`hamming`(*fp1*, *fp2*)

Calculate Hamming distance between fingerprints.

Parameters

- **fp1** (*Fingerprint*) – Fingerprint 1
- **fp2** (*Fingerprint*) – Fingerprint 2

Returns
float

Return type
Hamming distance.

`pearson`(*fp1*, *fp2*)

Calculate Pearson correlation between fingerprints.

Parameters

- **fp1** (*Fingerprint*) – Fingerprint 1
- **fp2** (*Fingerprint*) – Fingerprint 2

Returns
float

Return type
Pearson correlation.

`soergel`(*fp1*, *fp2*)

Calculate Soergel similarity between fingerprints.

Soergel similarity is the complement of Soergel distance and can be thought of as the analog of the Tanimoto coefficient for count/float-based fingerprints. For *Fingerprint*, it is equivalent to the Tanimoto coefficient.

Parameters

- **fp1** (*Fingerprint*) – Fingerprint 1
- **fp2** (*Fingerprint*) – Fingerprint 2

Returns

- **float** (*Soergel similarity.*)
- *Reference*
- ——-

`tanimoto`(*fp1*, *fp2*)

Calculate Tanimoto coefficient between fingerprints.

Parameters

- **fp1** (*Fingerprint*) – Fingerprint 1
- **fp2** (*Fingerprint*) – Fingerprint 2

Returns
float

Return type
Tanimoto coefficient.

Module contents

Efficient comparison metrics for fingerprints and their databases.

Author: Seth Axen E-mail: seth.axen@gmail.com

`cosine`(*A*, *B=None*)

Compute cosine similarities between fingerprints.

Fingerprints must have same number of bits. If only one fingerprint/database is provided, it is compared to self.

Parameters
A, B (*Fingerprint or FingerprintDatabase*) – Fingerprint(s) to be compared

Returns
cosine

Return type
float or ndarray [shape (num_fps_A, num_fps_B)]

See also:

dice, *pearson*, *soergel*, *tanimoto*

`dice`(*A*, *B=None*)

Compute Dice coefficients between fingerprints.

Fingerprints must have same number of bits. If not bit-fingerprints, arrays will be cast to binary. If only one fingerprint/database is provided, it is compared to self.

Parameters
A, B (*Fingerprint or FingerprintDatabase*) – Fingerprint(s) to be compared

Returns
dice

Return type
float or ndarray [shape (num_fps_A, num_fps_B)]

See also:

cosine, *pearson*, *soergel*, *tanimoto*

`pearson`(*A*, *B=None*)

Compute Pearson correlation between fingerprints.

Fingerprints must have same number of bits. If only one fingerprint/database is provided, it is compared to self.

Parameters
A, B (*Fingerprint or FingerprintDatabase*) – Fingerprint(s) to be compared

Returns
pearson

Return type
float or ndarray [shape (num_fps_A, num_fps_B)]

See also:

cosine, *dice*, *soergel*, *tanimoto*

soergel(*A*, *B=None*)

Compute Soergel similarities between fingerprints.

Soergel similarity is the complement of the Soergel distance and is analogous to the Tanimoto coefficient for count/float fingerprints. For binary data, it is equivalent to *tanimoto*.

Parameters
A, B (*Fingerprint or FingerprintDatabase*) – Fingerprint(s) to be compared

Returns
soergel

Return type
float or ndarray [shape (num_fps_A, num_fps_B)]

See also:

cosine, *dice*, *pearson*, *tanimoto*

tanimoto(*A*, *B=None*)

Compute Tanimoto coefficients between fingerprints.

Fingerprints must have same number of bits. If not bit-fingerprints, arrays will be cast to binary. For non-binary data, use *soergel*. If only one fingerprint/database is provided, it is compared to self.

Parameters
A, B (*Fingerprint or FingerprintDatabase*) – Fingerprint(s) to be compared

Returns
tanimoto – Pairwise tanimoto(s) between fingerprint(s) in *A* and *B*.

Return type
float or ndarray [shape (num_fps_A, num_fps_B)]

See also:

cosine, *dice*, *pearson*, *soergel*

Submodules

e3fp.fingerprint.array_ops module

Various array operations.

Author: Seth Axen E-mail: seth.axen@gmail.com

as_unit(*v*, *axis=1*)

Return array of unit vectors parallel to vectors in *v*.

Parameters

- **v** (*ndarray of float*)
- **axis** (*int, optional*) – Axis along which to normalize length.

Returns
ndarray of float – magnitude along *axis*.

Return type
Unit vector of *v*, i.e. *v* divided by its

`calculate_angles`(*vec_arr*, *ref*, *ref_norm=None*)

Calculate angles between vectors in *vec_arr* and *ref* vector.

If *ref_norm* is not provided, angle ranges between 0 and pi. If it is provided, angle ranges between 0 and 2pi. Note that if *ref_norm* is orthogonal to *vec_arr* and *ref*, then the angle is rotation around the axis, but if a non-orthogonal axis is provided, this may not be the case.

Parameters

- **vec_arr** (*Nx3 array of float*) – Array of N 3D vectors.
- **ref** (*1x3 array of float*) – Reference vector
- **ref_norm** (*1x3 array of float*) – Normal vector.

Returns
Array of N angles

Return type
1-D array

`make_distance_matrix`(*coords*)

Build pairwise distance matrix from coordinates.

Parameters
coords (*ndarray of float*) – an Mx3 array of cartesian coordinates.

Returns
ndarray of float

Return type
square symmetrical distance matrix

`make_rotation_matrix`(*v0*, *v1*)

Create 3x3 matrix of rotation from *v0* onto *v1*.

Should be used by dot(R, v0.T).T.

Parameters

- **v0** (*1x3 array of float*) – Initial vector before alignment.
- **v1** (*1x3 array of float*) – Vector to which to align *v0*.

`make_transform_matrix`(*center*, *y=None*, *z=None*)

Make 4x4 homogenous transformation matrix.

Given Nx4 array A where A[:, 4] = 1., the transform matrix M should be used with dot(M, A.T).T. Order of operations is 1. translation, 2. align *y* x *z* plane to yz-plane 3. align *y* to y-axis.

Parameters

- **center** (*1x3 array of float*) – Coordinate that should be centered after transformation.
- **y** (*None or 1x3 array of float*) – Vector that should lie on the y-axis after transformation
- **z** (*None or 1x3 array of float*) – Vector that after transformation should lie on yz-plane in direction of z-axis.

Returns
4x4 homogenous transformation matrix.

Return type
4x4 array of float

pad_array(*a*, *n=1.0*, *axis=1*)

Return *a* with row of *n* appended to *axis*.

Parameters

- **a** (*ndarray*) – Array to pad
- **n** (*float or int, optional*) – Value to pad *a* with
- **axis** (*int, optional*) – Axis of *a* to pad with *n*.

Returns
Padded array.

Return type
ndarray

project_to_plane(*vec_arr*, *norm*)

Project array of vectors to plane with normal *norm*.

Parameters

- **vec_arr** (*Nx3 array*) – Array of N 3D vectors.
- **norm** (*1x3 array*) – Normal vector to plane.

Returns
Array of vectors projected onto plane.

Return type
Nx3 array

quaternion_to_transform_matrix(*quaternion*, *translation=array([0., 0., 0.])*)

Convert quaternion to homogenous 4x4 transform matrix.

Parameters

- **quaternion** (*4x1 array of float*) – Quaternion describing rotation after translation.
- **translation** (*3x1 array of float, optional*) – Translation to be performed before rotation.

rotate_angles(*angles*, *amount*)

Rotate angles by *amount*, keeping in 0 to 2pi range.

Parameters

- **angles** (*1-D array of float*) – Angles in radians
- **amount** (*float*) – Amount to rotate angles by

Returns
1-D array of float

Return type
Rotated angles

transform_array(*transform_matrix*, *a*)

Pad an array with 1s, transform, and return with original dimensions.

Parameters

- **transform_matrix** (*4x4 array of float*) – 4x4 homogenous transformation matrix
- **a** (*Nx3 array of float*) – Array of 3-D coordinates.

Returns
Nx3 array of float

Return type
Transformed array

`transform_matrix_to_quaternion`(*transform_matrix*, *dtype=<class 'float'>*)

Convert homogenous 4x4 transform matrix to quaternion.

Parameters

- **transform_matrix** (*4x4 array of float*) – Homogenous transformation matrix.
- **dtype** (*numpy dtype, optional*) – Datatype for returned quaternion.

`unpad_array`(*a*, *axis=1*)

Return *a* with row removed along *axis*.

Parameters

- **a** (*ndarray*) – Array from which to remove row
- **axis** (*int, optional*) – Axis from which to remove row

Returns
Unpadded array.

Return type
ndarray

e3fp.fingerprint.db module

Database for accessing and serializing fingerprints.

Author: Seth Axen E-mail: seth.axen@gmail.com

`class FingerprintDatabase`(*fp_type=<class 'e3fp.fingerprint.fprint.Fingerprint'>*, *level=-1*, *name=None*)

Bases: `object`

Efficiently build, access, compare, and save fingerprints.

Fingerprints must have the same values of *bits* and *level*. Additionally, all fingerprints will be cast to the type of fingerprint passed to the database upon instantiation.

Parameters

- **fp_type** (*type, optional*) – Type of fingerprint (`Fingerprint`, `CountFingerprint`, `FloatFingerprint`).
- **level** (*int, optional*) – Level, or number of iterations used during fingerprinting.
- **name** (*str, optional*) – Name of database.

Variables

- **array** (*scipy.sparse.csr_matrix*) – Sparse matrix with dimensions N x M, where M is *bits*, and M is *fp_num*.
- **bits** (*int*) – Number of bits (length) of fingerprints.
- **fp_names** (*list of str*) – Names of fingerprints.
- **fp_names_to_indices** (*dict*) – Map from fingerprint name to row indices of `array`.
- **fp_num** (*int*) – Number of fingerprints in database.

- **fp_type** (*type*) – Type of fingerprint (Fingerprint, CountFingerprint, FloatFingerprint)
- **level** (*int*) – Level, or number of iterations used during fingerprinting.
- **name** (*str*) – Name of database
- **props** (*dict*) – Dict with keys specifying names of fingerprint properties and values corresponding to array of values.

Notes

Since most fingerprints are very sparse length-wise, *FingerprintDatabase* is implemented as a wrapper around a scipy.sparse.csr_matrix for efficient memory usage. This provides easy access to underlying data for tight integration with NumPy/SciPy and machine learning packages while simultaneously providing several fingerprint-specific features.

See also:

e3fp.fingerprint.fprint.Fingerprint
A fingerprint that stores indices of "on" bits

Examples

```
>>> from e3fp.fingerprint.db import FingerprintDatabase
>>> from e3fp.fingerprint.fprint import Fingerprint
>>> import numpy as np
>>> np.random.seed(2)
>>> db = FingerprintDatabase(fp_type=Fingerprint, name="TestDB")
>>> print(db)
FingerprintDatabase[name: TestDB, fp_type: Fingerprint, level: -1, bits: None, fp_
↪num: 0]
>>> bvs = (np.random.uniform(size=(3, 1024)) > .9).astype(bool)
>>> fps = [Fingerprint.from_vector(bvs[i, :], name="fp" + str(i))
...         for i in range(bvs.shape[0])]
>>> db.add_fingerprints(fps)
>>> print(db)
FingerprintDatabase[name: TestDB, fp_type: Fingerprint, level: -1, bits: 1024, fp_
↪num: 3]
```

The contained fingerprints may be accessed by index or name.

```
>>> db[0]
Fingerprint(indices=array([40, ..., 1012]), level=-1, bits=1024, name=fp0)
>>> db['fp2']
[Fingerprint(indices=array([0, ..., 1013]), level=-1, bits=1024, name=fp2)]
```

Alternatively, the underlying scipy.sparse.csr_matrix may be accessed.

```
>>> db.array
<3x1024 sparse matrix of type '<... 'numpy.bool_'>'
...with 327 stored elements in Compressed Sparse Row format>
>>> db.array.toarray()
array([[False, False, False, ..., False, False, False],
```

(continues on next page)

(continued from previous page)

```
       [False, False, False, ..., False, False, False],
       [ True, False, False, ..., False, False, False]])
```

Fingerprint properties may be stored in the database.

```
>>> db.set_prop("prop", np.arange(3))
```

The database can be efficiently stored and loaded.

```
>>> db.savez("/tmp/test_db.fpz")
>>> db = FingerprintDatabase.load("/tmp/test_db.fpz")
>>> print(db)
FingerprintDatabase[name: TestDB, fp_type: Fingerprint, level: -1, bits: 1024, fp_
↪num: 3]
```

Various comparison metrics in *e3fp.fingerprint.metrics* can operate efficiently directly on databases

```
>>> from e3fp.fingerprint.metrics import tanimoto, dice, cosine
>>> tanimoto(db, db)
array([[1.        , 0.0591133 , 0.04245283],
       [0.0591133 , 1.        , 0.0531401 ],
       [0.04245283, 0.0531401 , 1.        ]])
>>> dice(db, db)
array([[1.        , 0.11162791, 0.08144796],
       [0.11162791, 1.        , 0.10091743],
       [0.08144796, 0.10091743, 1.        ]])
>>> cosine(db, db)
array([[1.        , 0.11163878, 0.08145547],
       [0.11163878, 1.        , 0.10095568],
       [0.08145547, 0.10095568, 1.        ]])
```

add_fingerprints(*fprints*)

Add fingerprints to database.

Parameters

fprints (*iterable of Fingerprint*) – Fingerprints to add to database

as_type(*fp_type*, *copy=False*)

Get database with fingerprint type *fp_type*.

Parameters

- **fp_type** (*type*) – Type of fingerprint (`Fingerprint`, `CountFingerprint`, `FloatFingerprint`)
- **copy** (*bool, optional*) – Force copy of database. If False, if database is already of requested type, no copy is made.

Returns

Database coerced to fingerprint type of *fp_type*.

Return type

FingerprintDatabase

property bits

`fold`(*bits*, *fp_type=None*, *name=None*)

Get copy of database folded to specified bit length.

Parameters

- **bits** (*int*) – Number of bits to which to fold database.
- **fp_type** (*type or None, optional*) – Type of fingerprint (Fingerprint, CountFingerprint, FloatFingerprint). Defaults to same type.
- **name** (*str, optional*) – Name of database

Returns
Database folded to specified length.

Return type
FingerprintDatabase

Raises
BitsValueError – If `bits` is greater than the length of the database or database cannot be evenly folded to length `bits`.

`property fp_num`

`classmethod from_array`(*array*, *fp_names*, *fp_type=None*, *level=- 1*, *name=None*, *props={}*)

Instantiate from array.

Parameters

- **array** (*numpy.ndarray or scipy.sparse.csr_matrix*) – Sparse matrix with dimensions *N* x *M*, where *M* is the number of bits in the fingerprints.
- **fp_names** (*list of str*) – *N* names of fingerprints in `array`.
- **fp_type** (*type, optional*) – Type of fingerprint (Fingerprint, CountFingerprint, FloatFingerprint).
- **level** (*int, optional*) – Level, or number of iterations used during fingerprinting.
- **name** (*str or None, optional*) – Name of database.
- **props** (*dict, optional*) – Dict with keys specifying names of fingerprint properties and values corresponding to length *N* array of values.

Returns
Database containing fingerprints in `array`.

Return type
FingerprintDatabase

`get_density`(*index=None*)

Get percentage of fingerprints with 'on' bit at position.

Parameters
index (*int or None, optional*) – Index to bit for which to return positional density. If None, density for whole database is returned.

Returns
Density of 'on' position in database

Return type
float

`get_prop`(*key*)

Get property.

Raises
`KeyError` – If *key* not in *props*.

`get_subset`(*fp_names*, *name=None*)

Get database with subset of fingerprints.

Parameters

- **fp_names** (*list of str*) – List of fingerprint names to include in new db.
- **name** (*str, optional*) – Name of database

`classmethod load`(*fn*)

Load database from file.

The extension is used to determine how database was serialized (`save` vs `savez`).

Parameters
fn (*str*) – Filename

Returns
Database

Return type
FingerprintDatabase

`save`(***kwargs*)

Note: Deprecated in e3fp 1.2. `save` will be removed in e3fp 1.3. Use `savez` instead.

Save database to file.

fn
[str, optional] Filename or basename if extension does not include '.fps'

`savetxt`(*fn*, *with_names=True*)

Save bitstring representation to text file.

Only implemented for *fp_type* of `Fingerprint`. This should not be attempted for large numbers of bits.

Parameters

- **fn** (*str or filehandle*) – Out file. Extension is automatically parsed to determine whether compression is used.
- **with_names** (*bool, optional*) – Include name of fingerprint in same row after bitstring.

Raises

- ***E3FPInvalidFingerprintError*** – If *fp_type* is not `Fingerprint`.
- ***E3FPEfficiencyWarning*** – If `bits` is over 2^14 = 16384.

`savez`(*fn='fingerprints.fpz'*)

Save database to file.

Database is serialized using `numpy.savez_compressed`.

Parameters
fn (*str, optional*) – Filename or basename if extension is not '.fpz'

`set_prop`(*key*, *vals*, *check_length=True*)

Set values of property for fingerprints.

Parameters

- **key** (*str*) – Name of property
- **vals** (*array_like*) – Values of property.
- **check_length** (*bool, optional*) – Check to ensure number of properties match number of fingerprints already in database. This should only be set to False for temporary iterative updating.

`update_names_map`(*new_names=None*, *offset=0*)

Update map of fingerprint names to row indices of *self.array*.

Parameters

- **new_names** (*iterable of str, optional*) – Names to add to map. If None, map is completely rebuilt.
- **offset** (*int, optional*) – Number of rows before new rows.

`update_props`(*props_dict*, *append=False*, *check_length=True*)

Set multiple properties at once.

Parameters

- **props_dict** (*dict*) – Dict of properties. Values must be array-like of length *fp_num*.
- **append** (*bool, optional*) – Append values to those already in database. By default, properties are overwritten if already present.
- **check_length** (*bool, optional*) – Check to ensure number of properties match number of fingerprints already in database. This should only be set to False for temporary iterative updating.

`concat`(*dbs*)

Efficiently concatenate *FingerprintDatabase* objects.

The databases must be of the same type with the same number of bits, level, and property names.

Parameters
dbs (*iterable of FingerprintDatabase*) – Fingerprint databases

Returns
Database with all fingerprints from provided databases.

Return type
FingerprintDatabase

See also:

FingerprintDatabase

Examples

```
>>> from e3fp.fingerprint.db import FingerprintDatabase, concat
>>> from e3fp.fingerprint.fprint import Fingerprint
>>> import numpy as np
>>> np.random.seed(2)
>>> db1 = FingerprintDatabase(fp_type=Fingerprint, name="TestDB1", level=5)
>>> db2 = FingerprintDatabase(fp_type=Fingerprint, name="TestDB2", level=5)
>>> bvs = (np.random.uniform(size=(6, 1024)) > .9).astype(bool)
>>> fps = [Fingerprint.from_vector(bvs[i, :], name="fp" + str(i), level=5)
...         for i in range(bvs.shape[0])]
>>> db1.add_fingerprints(fps[:3])
>>> db2.add_fingerprints(fps[3:])
>>> print(concat([db1, db2]))
FingerprintDatabase[name: None, fp_type: Fingerprint, level: 5, bits: 1024, fp_num:␣
↪6]
```

e3fp.fingerprint.fprint module

Classes and methods for chemical fingerprint storage and comparison.

Author: Seth Axen E-mail: seth.axen@gmail.com

class CountFingerprint(*indices=None*, *counts=None*, *bits=4294967296*, *level=- 1*, *name=None*, *props={}*, ***kwargs*)

Bases: *Fingerprint*

A fingerprint that stores number of occurrences of each index.

Parameters

- **indices** (*array_like of int, optional*) – log2(`bits`)-bit indices in a sparse vector, corresponding to positions with counts greater than 0. If not provided, *counts* must be provided.
- **counts** (*dict, optional*) – Dict matching each index in `indices` to number of counts. All counts default to 1 if not provided.
- **bits** (*int, optional*) – Number of bits in bitvector.
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Variables

- **bits** (*int*) – Number of bits in bitvector, length of fingerprint.
- **counts** (*dict*) – Dict matching each index in `indices` to number of counts.
- **indices** (*numpy.ndarray of int*) – Indices of fingerprint with counts greater than 0.
- **level** (*int*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **mol** (*RDKit Mol*) – Mol to which fingerprint corresponds (stored in `props`).
- **name** (*str or None*) – Name of fingerprint (stored in `props`).

- **props** (*dict*) – Custom properties of fingerprint, consisting of a string keyword and some value.
- **vector_dtype** (*numpy.dtype*) – NumPy data type associated with fingerprint values (e.g. bits)

See also:

Fingerprint
A fingerprint that stores indices of "on" bits

FloatFingerprint
A fingerprint that stores float counts

Examples

```
>>> import e3fp.fingerprint.fprint as fp
>>> from e3fp.fingerprint.metrics import soergel
>>> import numpy as np
>>> np.random.seed(1)
>>> bits = 1024
>>> indices = np.random.randint(0, bits, 30)
>>> print(indices)
[ 37 235 908  72 767 905 715 645 847 960 144 129 972 583 749 508 390 281
 178 276 254 357 914 468 907 252 490 668 925 398]
>>> counts = dict(zip(indices,
...                   np.random.randint(1, 100, indices.shape[0])))
>>> print(sorted(counts.items()))
[(37, 51), (72, 88), (129, 62), ..., (925, 50), (960, 8), (972, 23)]
>>> f = fp.CountFingerprint(indices, counts=counts, bits=bits, level=0)
>>> f_folded = f.fold(bits=32)
>>> print(sorted(f_folded.counts.items()))
[(0, 8), (1, 62), (5, 113), ..., (29, 50), (30, 14), (31, 95)]
>>> print(f_folded.to_vector(sparse=False, dtype=int))
[  8  62   0   0   0 113  61  58  88  97  71 228 111   2  58  10  64   0
  82   0 120   0   0   0   0  82   0   0  27  50  14  95]
>>> fp.Fingerprint.from_fingerprint(f_folded)
Fingerprint(indices=array([0, 1, ...]), level=0, bits=32, name=None)
>>> indices2 = np.random.randint(0, bits, 30)
>>> counts2 = dict(zip(indices2,
...                    np.random.randint(1, 100, indices.shape[0])))
>>> f_folded2 = fp.CountFingerprint.from_indices(indices2, counts=counts2,
...                                              bits=bits).fold(bits=32)
>>> print(sorted(f_folded2.counts.items()))
[(0, 93), (2, 33), (3, 106), ..., (25, 129), (26, 89), (30, 53)]
>>> print(soergel(f_folded, f_folded2))
0.17492946392...
```

property counts

fold(**args*, ***kwargs*)

Fold fingerprint while considering counts.

Optionally, provide a function to reduce colliding counts.

Parameters

- **bits** (*int, optional*) – Length of new bitvector, ideally multiple of 2.
- **method** (*{0, 1}, optional*) – Method to use for folding.

  0
  partitioning (array is divided into equal sized arrays of length `bits` which are bitwise combined with *counts_method*)

  1
  compression (adjacent bits pairs are combined with *counts_method* until length is `bits`)
- **linked** (*bool, optional*) – Link folded and unfolded fingerprints for easy referencing. Set to False if intending to save and want to reduce file size.
- **counts_method** (*function, optional*) – Function for combining counts. Default is summation.

Returns
CountFingerprint

Return type
Fingerprint of folded vector

`classmethod from_counts`(*counts*, *bits=4294967296*, *level=- 1*, ***kwargs*)

Initialize from an array of indices.

Parameters

- **counts** (*dict*) – Dictionary mapping sparse indices to counts.
- **bits** (*int, optional*) – Number of bits in array. Indices will be log2(`bits`)-bit integers.
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Returns
fingerprint

Return type
CountFingerprint

`classmethod from_fingerprint`(*fp*, ***kwargs*)

Initialize by copying existing fingerprint.

Parameters

- **fp** (*Fingerprint*) – Existing fingerprint.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Returns
fingerprint

Return type
Fingerprint

classmethod from_indices(*indices*, *counts=None*, *bits=4294967296*, *level=- 1*, ***kwargs*)

Initialize from an array of indices.

Parameters

- **indices** (*array_like of int, optional*) – Indices in a sparse bitvector of length `bits` which correspond to 1.
- **counts** (*dict, optional*) – Dictionary mapping sparse indices to counts.
- **bits** (*int, optional*) – Number of bits in array. Indices will be log2(`bits`)-bit integers.
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Returns
fingerprint

Return type
CountFingerprint

get_count(*index*)

Return count index in fingerprint.

Returns
int

Return type
Count of index in fingerprint

mean()

Return mean of counts.

Returns
float

Return type
Mean

reset(**args*, ***kwargs*)

Reset all values.

std()

Return standard deviation of fingerprint.

Returns
float

Return type
Standard deviation

vector_dtype

alias of `uint16`

class Fingerprint(*indices*, *bits=4294967296*, *level=- 1*, *name=None*, *props={}*, ***kwargs*)

Bases: `object`

A fingerprint that stores indices of "on" bits.

Parameters

- **indices** (*array_like of int, optional*) – log2(*bits*)-bit indices in a sparse bitvector of *bits* which correspond to 1.
- **bits** (*int, optional*) – Number of bits in bitvector.
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Variables

- **bits** (*int*) – Number of bits in bitvector, length of fingerprint.
- **counts** (*dict*) – Dict matching each index in *indices* to number of counts (1 for bits).
- **indices** (*numpy.ndarray of int*) – Indices of "on" bits
- **level** (*int*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **mol** (*RDKit Mol*) – Mol to which fingerprint corresponds (stored in *props*).
- **name** (*str or None*) – Name of fingerprint (stored in *props*).
- **props** (*dict*) – Custom properties of fingerprint, consisting of a string keyword and some value.
- **vector_dtype** (*numpy.dtype*) – NumPy data type associated with fingerprint values (e.g. bits)

See also:

CountFingerprint
A fingerprint that stores number of occurrences of each index

FloatFingerprint
A fingerprint that stores indices of "on" bits

e3fp.fingerprint.db.FingerprintDatabase
Efficiently store fingerprints

Examples

```
>>> import e3fp.fingerprint.fprint as fp
>>> from e3fp.fingerprint.metrics import tanimoto
>>> import numpy as np
>>> np.random.seed(0)
>>> bits = 1024
>>> indices = np.random.randint(0, bits, 30)
>>> print(indices)
[684 559 629 192 835 763 707 359   9 723 277 754 804 599  70 472 600 396
 314 705 486 551  87 174 600 849 677 537 845  72]
>>> f = fp.Fingerprint(indices, bits=bits, level=0)
>>> f_folded = f.fold(bits=32)
>>> print(f_folded.indices)
[ 0  1  3  4  5  6  7  8  9 12 13 14 15 17 18 19 21 23 24 25 26 27]
>>> print(f_folded.to_vector(sparse=False, dtype=int))
[1 1 0 1 1 1 1 1 1 1 0 0 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
>>> print(f_folded.to_bitstring())
11011111110011110111010111110000
>>> print(f_folded.to_rdkit())
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> f_folded2 = fp.Fingerprint.from_indices(np.random.randint(0, bits, 30),
...                                         bits=bits).fold(bits=32)
>>> print(f_folded2.indices)
[ 0  1  3  5  7  9 10 14 15 16 17 18 19 20 23 24 25 29 30 31]
>>> print(tanimoto(f_folded, f_folded2))
0.5
```

property bit_count

property bits

clear()

Clear temporary (and possibly large) values.

property counts

property density

fold(*bits=1024*, *method=0*, *linked=True*)

Return fingerprint for bitvector folded to size `bits`.

Parameters

- **bits** (*int, optional*) – Length of new bitvector, ideally multiple of 2.
- **method** (*{0, 1}, optional*) – Method to use for folding.

  0
  partitioning (array is divided into equal sized arrays of length `bits` which are bitwise combined with OR)

  1
  compression (adjacent bits pairs are combined with OR until length is `bits`)
- **linked** (*bool, optional*) – Link folded and unfolded fingerprints for easy referencing. Set to False if intending to save and want to reduce file size.

Returns
Fingerprint

Return type
Fingerprint of folded bitvector

classmethod from_bitstring(*bitstring*, *level=- 1*, ***kwargs*)

Initialize from bitstring (e.g. '10010011').

Parameters

- **bitstring** (*str*) – String of 1s and 0s.
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Returns
fingerprint

Return type
Fingerprint

`classmethod from_fingerprint`(*fp*, ***kwargs*)

Initialize by copying existing fingerprint.

Parameters
fp (*Fingerprint*) – Existing fingerprint.

Returns
fingerprint

Return type
Fingerprint

`classmethod from_indices`(*indices*, *bits=4294967296*, *level=- 1*, ***kwargs*)

Initialize from an array of indices.

Parameters

- **indices** (*array_like of int*) – Indices in a sparse bitvector of length `bits` which correspond to 1.
- **bits** (*int, optional*) – Number of bits in array. Indices will be log2(`bits`)-bit integers.
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Returns
fingerprint

Return type
Fingerprint

`classmethod from_rdkit`(*rdkit_fprint*, ***kwargs*)

Initialize from RDKit fingerprint.

If provided fingerprint is of length 2^32 - 1, assumes real fingerprint is of length 2^32.

Parameters

- **rdkit_fprint** (*RDKit ExplicitBitVect or SparseBitVect*) – Existing RDKit fingerprint.
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Returns
fingerprint

Return type
Fingerprint

classmethod from_vector(*vector*, *level=- 1*, ***kwargs*)

Initialize from vector.

Parameters

- **vector** (*numpy.ndarray or scipy.sparse.csr_matrix*) – Array of bits/counts/floats
- **level** (*int, optional*) – Level of fingerprint, corresponding to fingerprinting iterations.
- **name** (*str, optional*) – Name of fingerprint.
- **props** (*dict, optional*) – Custom properties of fingerprint, consisting of a string keyword and some value.

Returns
fingerprint

Return type
Fingerprint

get_count(*index*)

Return count index in fingerprint.

Defaults to 1 if index in *self.indices*

Returns
int

Return type
Count of bit in fingerprint

get_folding_index_map()

Get map of sparse indices to folded indices.

Returns
dict

Return type
Map of sparse index (keys) to corresponding folded index.

get_prop(*key*)

Get property. If not set, raise KeyError.

get_unfolding_index_map()

Get map of sparse indices to unfolded indices.

Returns
dict – indices.

Return type
Map of sparse index (keys) to set of corresponding unfolded

property index_id_map

property indices

property level

mean()

Return mean, i.e. proportion of "on" bits in fingerprint.

Returns
float

Return type
Mean

`property mol`

`property name`

`property props`

`reset()`
Reset all values.

`set_prop(key, val)`
Set property.

`std()`
Return standard deviation of fingerprint.

Returns
float

Return type
Standard deviation

`to_bitstring()`
Get bitstring as string of 1s and 0s.

Returns
str

Return type
bitstring

`to_bitvector(sparse=True)`
Get full bitvector.

Returns
numpy.ndarray or scipy.sparse.csr_matrix of bool

Return type
Bitvector

`to_rdkit()`
Convert to RDKit fingerprint.

If number of bits exceeds 2^31 - 1, fingerprint will be folded to length 2^31 - 1 before conversion.

Returns
rdkit_fprint – Convert to bitvector used for RDKit fingerprints. If *self.bits* is less than 10^5, `ExplicitBitVect` is used. Otherwise, `SparseBitVect` is used.

Return type
RDKit ExplicitBitVect or SparseBitVect

`to_vector(sparse=True, dtype=None)`
Get vector of bits/counts/floats.

Returns
Vector of bits/counts/floats

Return type
numpy.ndarray or scipy.sparse.csr_matrix

unfold()

Return unfolded parent fingerprint for bitvector.

Returns
Fingerprint – None.

Return type
Fingerprint of unfolded bitvector. If None, return

update_props(*props_dict*)

Set multiple properties at once.

vector_dtype

alias of bool_

class FloatFingerprint(*indices=None*, *counts=None*, *bits=4294967296*, *level=- 1*, *name=None*, *props={}*, ***kwargs*)

Bases: *CountFingerprint*

A Fingerprint that stores float counts.

Nearly identical to *CountFingerprint*. Mainly a naming convention, but count values are stored as floats.

See also:

Fingerprint
A fingerprint that stores indices of "on" bits

CountFingerprint
A fingerprint that stores number of occurrences of each index

property counts

vector_dtype

alias of float64

add(*fprints*, *weights=None*)

Add fingerprints by count to new *CountFingerprint*.

If any of the fingerprints are *FloatFingerprint*, resulting fingerprint is likewise a *FloatFingerprint*. Otherwise, resulting fingerprint is *CountFingerprint*.

Parameters

- **fprints** (*iterable of Fingerprint*) – Fingerprints to be added by count.
- **weights** (*iterable of float*) – Weights for weighted sum. Results in *FloatFingerprint* output.

Returns
Fingerprint with counts as sum of counts in *fprints*.

Return type
CountFingerprint or *FloatFingerprint*

See also:

mean

coerce_to_valid_dtype(*dtype*)

Coerce provided NumPy data type to closest fingerprint data type.

If provided *dtype* cannot be read, default corresponding to bit `Fingerprint` is returned.

Parameters
dtype (*numpy.dtype or str*) – Input NumPy data type.

Returns
Output NumPy data type.

Return type
numpy.dtype

diff_counts_dict(*fp1*, *fp2*, *only_positive=False*)

Given two fingerprints, returns difference of their counts dicts.

Parameters

- **fp1, fp2** (*Fingerprint*) – `Fingerprint` objects, *fp2* subtracted from *fp1*.
- **only_positive** (*bool, optional*) – Return only positive counts, negative being thresholded to 0.

Returns
counts_diff – Count indices in either *fp1* or *fp2* with value as diff of counts.

Return type
dict

See also:

`sum_counts_dict`

dtype_from_fptype(*fp_type*)

Get NumPy data type from fingerprint type.

Parameters
fp_type (*class or Fingerprint*) – Class of fingerprint

Returns
NumPy data type

Return type
numpy.dtype

fptype_from_dtype(*dtype*)

Get corresponding fingerprint type from NumPy data type.

Parameters
dtype (*numpy.dtype or str*) – NumPy data type.

Returns
class – Class of fingerprint

Return type
{Fingerprint, *CountFingerprint*, FloatFingerprint}

load(*f*, *update_structure=True*)

Load `Fingerprint` object from file.

Parameters

- **f** (*str or File*) – File name or file-like object to load file from.

- **update_structure** (*bool, optional*) – Attempt to update the class structure by initializing a new, shiny fingerprint from each fingerprint in the file. Useful for guaranteeing that old, dusty fingerprints are always upgradeable.

Returns
Fingerprint

Return type
Pickled fingerprint.

See also:

`loadz`, `save`

`loadz`(*f*, *update_structure=True*)

Load `Fingerprint` objects from file.

Parameters

- **f** (*str or File*) – File name or file-like object to load file from.
- **update_structure** (*bool, optional*) – Attempt to update the class structure by initializing a new, shiny fingerprint from each fingerprint in the file. Useful for guaranteeing that old, dusty fingerprints are always upgradeable. If this doesn't work, falls back to the original saved fingerprint.

Returns
list of Fingerprint

Return type
Fingerprints in pickle.

See also:

`load`, `savez`

`mean`(*fprints*, *weights=None*)

Average fingerprints to generate `FloatFingerprint`.

Parameters

- **fprints** (*iterable of Fingerprint*) – Fingerprints to be added by count.
- **weights** (*array_like of float, optional*) – Weights for weighted mean. Weights are normalized to a sum of 1.

Returns
FloatFingerprint – *fprints*.

Return type
Fingerprint with float counts as average of counts in

`save`(*f*, *fp*, ***kwargs*)

Save `Fingerprint` object to file.

Parameters

- **f** (*str or File*) – filename *str* or file-like object to save file to
- **fp** (*Fingerprint*) – Fingerprint to save to file
- **protocol** (*{0, 1, 2, None}, optional*) – Pickle protocol to use. If None, highest available protocol is used. This will not affect fingerprint loading.

Returns
bool

Return type
Success or fail

See also:

savez, *load*

savez(*f*, **fps*, ***kwargs*)

Save multiple *Fingerprint* objects to file.

Parameters

- **f** (*str or File*) – filename *str* or file-like object to save file to
- **fps** (*list of Fingerprint*) – List of Fingerprints to save to file
- **protocol** (*{0, 1, 2, None}, optional*) – Pickle protocol to use. If None, highest available protocol is used. This will not affect fingerprint loading.

Returns
bool

Return type
Success or fail

See also:

save, *loadz*

sum_counts_dict(**fprints*, ***kwargs*)

Given fingerprints, return sum of their counts dicts.

If an optional *weights* iterable of the same length as *fprints* is provided, the weighted sum is returned.

Parameters

- ***fprints** – One or more *Fingerprint* objects
- **weights** (*iterable of float, optional*) – Weights for weighted mean. Weights are normalized to a sum of 1.

Returns
dict – as sum of counts.

Return type
Dict of non-zero count indices in any of the *fprints* with value

See also:

diff_counts_dict

e3fp.fingerprint.fprinter module

Tools for generating E3FP fingerprints.

Author: Seth Axen E-mail: seth.axen@gmail.com

class Fingerprinter(*bits=4294967296*, *level=5*, *radius_multiplier=1.718*, *stereo=True*, *counts=False*, *include_disconnected=True*, *rdkit_invariants=False*, *exclude_floating=True*, *remove_duplicate_substructs=True*)

Bases: `object`

E3FP fingerprint generator.

Parameters

- **bits** (*int or None, optional*) – Maximum number of bits to which to fold returned fingerprint. Multiple of 2 is strongly recommended.
- **level** (*int or None, optional*) – Maximum number of iterations for fingerprint generation. If None or -1, run until no new substructures are identified. Because this could produce a different final level number for each conformer, it is recommended to manually specify a level.
- **radius_multiplier** (*float, optional*) – Multiple by which to increase shell size. At iteration 0, shell radius is 0*`radius_multiplier`, at iteration 2, radius is 2*`radius_multiplier`, etc.
- **counts** (*bool, optional*) – Instead of simple bit-based `Fingerprint` object, generate `CountFingerprint` that tracks number of times each bit appears in a fingerprint.
- **stereo** (*bool, optional*) – Differentiate based on stereography. Resulting fingerprints are not comparable to non-stereo fingerprints.
- **remove_duplicate_substructs** (*bool, optional*) – If a substructure arises that corresponds to an identifier already in the fingerprint, then the identifier for the duplicate substructure is not added to fingerprint.
- **include_disconnected** (*bool, optional:*) – Include disconnected atoms from hashes and substructure. E3FP's advantage over ECFP relies on disconnected atoms, so the option to turn this off is present only for testing/comparison.
- **rdkit_invariants** (*bool, optional*) – Use the atom invariants used by RDKit for its Morgan fingerprint.
- **exclude_floating** (*bool, optional:*) – Mask atoms with no bonds (usually floating ions) from the fingerprint. These are often placed arbitrarily and can confound the fingerprint.

Variables

- **current_level** (*int*) – The maximum level/iteration to which the fingerprinter has been run on the current conformer.
- **level_shells** (*dict*) – Dict matching level to set of all shells accepted at that level.

property current_level

get_fingerprint_at_level(*level=- 1*, *bits=None*, *exact=False*, *atom_mask={}*)

Get the fingerprint at the specified level.

Parameters

- **level** (*int or None, optional*) – Level/iteration

- **bits** (*int or None, optional*) – Return fingerprints folded to this number of bits. If unspecified, defaults to bits set when instantiated.
- **exact** (*bool, optional*) – Exact level
- **atom_mask** (*int or set of int, optional*) – Don't return shells whose substructures contain these atoms.

Returns
Fingerprint

Return type
Fingerprint at level

`get_shells_at_level`(*level=- 1*, *exact=False*, *atom_mask={}*)

Get set of shells at the specified level.

Parameters

- **level** (*int or None, optional*) – Level/iteration
- **exact** (*bool, optional*) – Exact level
- **atom_mask** (*int or set of int, optional*) – Don't return shells whose substructures contain these atoms.

Returns
set of Shell

Return type
Shells at level

`initialize_conformer`(*conf*)

Retrieve atom coordinates and instantiate shells generator.

Parameters
conf (*RDKit Conformer*) – Conformer to fingerprint

`initialize_identifiers`()

Set initial identifiers for atoms.

`initialize_mol`(*mol*)

Set general properties of *mol* that apply to all its conformers.

Parameters
mol (*RDKit Mol*) – Input molecule *Mol* object.

`next`()

Run next iteration of fingerprinting.

`reset`()

Clear all variables associated with the last run.

`reset_conf`()

Clear only conformer-specific variables.

`reset_mol`()

Clear all variables associated with the molecule.

`run`(*conf=None*, *mol=None*, *return_substruct=False*)

Generate fingerprint from provided conformer or mol and conf id.

Parameters

- **conf** (*RDKit Conformer or int, optional*) – Input conformer or conformer in *mol*.
- **mol** (*RDKit Mol, optional*) – Input molecule object, with at least one conformer. If `conf` not specified, first conformer is used.
- **return_substruct** (*bool, optional*) – Return dict mapping substructure to fingerprint indices. Keys are indices, values are list of substructures, represented as a tuple of atom indices where the first index is the central atom and the remaining indices (within the sphere) are sorted.

`substructs_to_pdb`(*level=None*, *bits=None*, *out_dir='substructs'*, *reorient=True*, *exact=False*)

Save all accepted substructs from current level to PDB.

Parameters

- **level** (*int or None, optional*) – Level of fingerprinting/number of iterations
- **bits** (*int or None, optional*) – Folding level of identifiers
- **out_dir** (*str, optional*) – Directory to which to save PDB files.
- **reorient** (*bool, optional*) – Reorient substructure to match stereo quadrants.

`class ShellsGenerator`(*conf*, *atoms*, *radius_multiplier=0.5*, *include_disconnected=True*, *atom_coords=None*, *bound_atoms_dict=None*)

Bases: `object`

Generate nested `Shell` objects from molecule upon request.

`back`()

Back up one iteration.

`get_match_atoms`(*rad*)

Get atoms within shell at radius *rad*.

Parameters
rad (*float*) – Radius of shell.

Returns
dict – shell

Return type
Dict matching atom id to set of ids for other atoms within

`get_shells_at_level`(*level*)

Get *dict* of atom shells at specified level/iteration.

If not run to *level*, raises *IndexError*.

Parameters
level (*int*) – Level/iteration from which to retrieve shells *dict*.

Returns
dict

Return type
Dict matching atom ids to that atom's `Shell` at that level.

`next`()

Get next iteration's *dict* of atom shells.

atom_tuples_from_shell(*shell*, *atom_coords*, *connectivity*, *stereo*)

Generate sorted atom tuples for neighboring atoms.

Parameters

- **shell** (*Shell*) – Shell for which to build atom tuples
- **atom_coords** (*dict*) – Dict matching atom ids to coords.
- **connectivity** (*dict*) – Dict matching atom id pair tuples to their bond order (5 for unbound).
- **stereo** (*bool*) – Add stereo indicators to tuples

bound_atoms_from_mol(*mol*, *atoms*)

Build *dict* matching atom id to ids of bounded atoms.

Bound atoms not in *atoms* are ignored.

Parameters

- **mol** (*RDKit Mol*) – Input mol
- **atoms** (*list of int*) – List of atom IDs

Returns
dict

Return type
Dict matching atom id to set of bound atom ids.

coords_from_atoms(*atoms*, *conf*)

Build *dict* matching atom id to coordinates.

Parameters

- **atoms** (*list of int*) – Atom ids
- **conf** (*RDKit Conformer*) – Conformer from which to fetch coordinates

Returns
dict

Return type
Dict matching atom id to 1-D array of coordinates.

get_first_unique_tuple_inds(*tuples_list*, *num_ret*, *ignore=[]*, *assume_sorted=True*)

Return indices of first *num_ret* unique tuples in a list.

Only first 2 values of each tuple are considered.

Parameters

- **tuples_list** (*list of tuple*) – List of tuples. Only first two unique values are considered.
- **num_ret** (*int*) – Maximum number of first unique tuples to return.
- **ignore** (*list, optional*) – Indices for tuples not be considered as unique.
- **assume_sorted** (*bool, optional*) – If True, assume list is already sorted by tuples.

Returns
tuple of int – unique tuples in list.

Return type
List of at most *num_ret* ints indicating index of

hash_int64_array(*array*, *seed=0*)

Hash an int64 array into a 32-bit integer.

Parameters

- **array** (*ndarray of int64*) – Numpy array containing integers
- **seed** (*any, optional*) – Seed for MurmurHash3.

Returns
int

Return type
32-bit integer

identifier_from_shell(*shell*, *atom_coords*, *connectivity*, *level*, *stereo*)

Determine new identifier for a shell at a specific level.

Parameters

- **shell** (*Shell*) – Shell for which to determine identifier
- **atom_coords** (*dict*) – Dict matching atom ids to coords.
- **connectivity** (*dict*) – Dict matching atom id pair tuples to their bond order (5 for unbound).
- **level** (*int*) – Level/iteration
- **stereo** (*bool*) – Add stereo indicators

identifiers_from_invariants(*mol*, *atoms*, *rdkit_invariants=False*)

Initialize ids according to Daylight invariants.

Parameters

- **mol** (*RDKit Mol*) – Input molecule
- **atoms** (*list of int*) – IDs for atoms in mol for which to generate identifiers.
- **rdkit_invariants** (*bool, optional*) – Use the atom invariants used by RDKit for its Morgan fingerprint.

Returns
ndarray of int64

Return type
initial identifiers for atoms

invariants_from_atom(*atom*)

Get seven invariants from atom.

Invariants used are the six Daylight invariants, plus an indicator of whether the atom is in a ring, as detailed in [1].

References

1. D Rogers, M Hahn. J. Chem. Inf. Model., 2010, 50 (5), pp 742-754 https://doi.org/10.1021/ci100050t

Parameters
atom (*RDKit Atom*) – Input atom

Returns
1-D array if int64

Return type
Array of 7 invariants

`pick_y`(*atom_tuples*, *cent_coords*, *y_precision=0.1*)

Pick a y-coordinate from atom tuples or mean coordinate.

Parameters

- **atom_tuples** (*list of tuple*) – Sorted list of atom tuples
- **cent_coords** (*Nx3 array of float*) – Coordinates of atoms with center atom at origin.
- **y_precision** (*str, optional*) – For mean to be chosen for y-coordinate, it must be at least this distance from the origin. Useful when atoms are symmetrical around the center atom where a slight shift in any atom results in a very different y.

Returns

- **1x3 array of float or None** (*y-coordinate*)
- **int or None** (*index to y-atom, if y was chosen from the atoms.*)

`pick_z`(*connectivity*, *identifiers*, *cent_coords*, *y*, *long_angle*, *z_precision=0.01*)

Pick a z-coordinate orthogonal to *y*.

Parameters

- **connectivity** (*dict*) – Dict matching atom id pair tuples to their bond order (5 for unbound).
- **identifiers** (*iterable of int*) – Atom identifiers
- **cent_coords** (*Nx3 array of float*) – Coordinates of atoms with center atom at origin.
- **y** (*1x3 array of float*) – y-coordinate
- **long_angle** (*Nx1 array of float*) – Absolute angle of atoms from orthogonal to *y*.
- **z_precision** (*float, optional*) – Minimum difference in *long_angle* between two potential z-atoms. Used as a tie breaker to prevent small shift in one atom resulting in very different z.

Returns
1x3 array of float or None

Return type
z-coordinate

`quad_indicators_from_coords`(*cent_coords*, *y*, *y_ind*, *z*, *long_sign*)

Create angle indicators for four quadrants in each hemisphere.

Parameters

- **cent_coords** (*Nx3 array of float*) – Array of centered coordinates.
- **y** (*1-D array of float*) – Vector lying along y-axis.

- **y_ind** (*int*) – Index of *cent_coords* corresponding to *y*.
- **z** (*1-D array of float*) – Vector lying along z-axis
- **long_sign** (*Nx1 array of int*) – Array of signs of vectors in *cent_coords* indicating whether they are above (+1) or below (-1) the xz-plane.

Returns
Nx1 array of int – indicators are 2, 3, 4, 5 for vectors above the xz-plane and -2, -3, -4, -5 for vectors below the xz-plane.

Return type
Quadrant indicators. Clockwise from *z* around *y*,

`rdkit_invariants_from_atom`(*atom*)

Get the 6 atom invariants RDKit uses for its Morgan fingerprints.

Parameters
atom (*RDKit Atom*) – Input atom

Returns
1-D array if int64

Return type
Array of 6 invariants

`signed_to_unsigned_int`(*a*, *bits=4294967296*)

Convert *int* between +/-*bits* to an int between 0 and *bits*.

Parameters

- **a** (*int or ndarray of int*) – Integer
- **bits** (*int, optional*) – Maximum size of int. E.g. 32-bit is 2^32.

Returns
int

Return type
unsigned integer

`stereo_indicators_from_shell`(*shell*, *atom_tuples*, *atom_coords_dict*, *add_transform_to_shell=True*)

Get *list* of *int* indicating location of atoms on unit sphere.

Parameters

- **shell** (*Shell*) – Shell for which to get stereo indicators.
- **atom_tuples** (*list of tuple*) – List of atom tuples.
- **atom_coords_dict** (*dict*) – Dict matching atom ids to coords.
- **add_transform_to_shell** (*bool, optional*) – Calculate transformation matrix to align coordinates to unit sphere, and add to shell.

Returns
list of int

Return type
stereo indicators for atoms in *atom_tuples*.

e3fp.fingerprint.generate module

Generate E3FP fingerprints.

Author: Seth Axen E-mail: seth.axen@gmail.com

fprints_dict_from_mol(*mol*, *bits=4294967296*, *level=5*, *radius_multiplier=1.718*, *first=3*, *counts=False*, *stereo=True*, *include_disconnected=True*, *rdkit_invariants=False*, *exclude_floating=True*, *remove_duplicate_substructs=True*, *out_dir_base=None*, *out_ext='.fp.bz2'*, *save=False*, *all_iters=False*, *overwrite=False*)

Build a E3FP fingerprint from a mol with at least one conformer.

Parameters

- **mol** (*RDKit Mol*) – Input molecule with one or more conformers to be fingerprinted.
- **bits** (*int*) – Set number of bits for final folded fingerprint.
- **level** (*int, optional*) – Level/maximum number of iterations of E3FP. If -1 is provided, it runs until termination, and *all_iters* is set to False.
- **radius_multiplier** (*float, optional*) – Radius multiplier for spherical shells.
- **first** (*int, optional*) – First *N* number of conformers from file to fingerprint. If -1, all are fingerprinted.
- **counts** (*bool, optional*) – Instead of bit-based fingerprints. Otherwise, generate count-based fingerprints.
- **stereo** (*bool, optional*) – Incorporate stereochemistry in fingerprint.
- **remove_duplicate_substructs** (*bool, optional*) – If a substructure arises that corresponds to an identifier already in the fingerprint, then the identifier for the duplicate substructure is not added to fingerprint.
- **include_disconnected** (*bool, optional*) – Include disconnected atoms when hashing and for stereo calculations. Turn off purely for testing purposes, to make E3FP more like ECFP.
- **rdkit_invariants** (*bool, optional*) – Use the atom invariants used by RDKit for its Morgan fingerprint.
- **exclude_floating** (*bool, optional:*) – Mask atoms with no bonds (usually floating ions) from the fingerprint. These are often placed arbitrarily and can confound the fingerprint.
- **out_dir_base** (*str, optional*) – Basename of out directory to save fingerprints. Iteration number is appended.
- **out_ext** (*str, optional*) – Extension on fingerprint pickles, used to determine compression level.
- **save** (*bool, optional*) – Save fingerprints to directory.
- **all_iters** (*bool, optional*) – Save fingerprints from all iterations to file(s).
- **overwrite** (*bool, optional*) – Overwrite pre-existing file.
- **Deleted Parameters**
- **————————**
- **sdf_file** (*str*) – SDF file path.

fprints_dict_from_sdf(*sdf_file*, ***kwargs*)

Build fingerprints dict for conformers encoded in an SDF file.

See *fprints_dict_from_mol* for description of arguments.

run(*sdf_files*, *bits=4294967296*, *first=3*, *level=5*, *radius_multiplier=1.718*, *counts=False*, *stereo=True*, *include_disconnected=True*, *rdkit_invariants=False*, *exclude_floating=True*, *remove_duplicate_substructs=True*, *params=None*, *out_dir_base=None*, *out_ext='.fp.bz2'*, *db_file=None*, *overwrite=False*, *all_iters=False*, *log=None*, *num_proc=None*, *parallel_mode=None*, *verbose=False*)

Generate E3FP fingerprints from SDF files.

e3fp.fingerprint.structs module

Class for defining 3D atom environments.

Author: Seth Axen E-mail: seth.axen@gmail.com

exception FormatError

Bases: Exception

class Shell(*center_atom*, *shells={}*, *radius=None*, *last_shell=None*, *identifier=None*)

Bases: object

A container for other Shells centered on an atom.

Shells represent all atoms explicitly within a container. Atoms are represented by their ids. If atoms are provided instead of shells, they are converted to single-atom shells. A Substruct is generated from a Shell on the fly by recursion through member shells. An optional identifier may be set.

property atoms

Get all atoms explicitly within the shell.

property center_atom

classmethod from_substruct(*substruct*)

Create shell with one shell for each atom in the substruct.

property shells

property substruct

Get substruct with all atoms implicitly within the shell.

class Substruct(*center_atom=None*, *atoms={}*)

Bases: object

A container for atoms optionally centered on an atom.

A Substruct represents all atoms implicitly within a Shell. Two Substructs are equal if they contain the same atoms.

property atoms

property center_atom

classmethod from_shell(*shell*)

shell_to_pdb(*mol*, *shell*, *atom_coords*, *bound_atoms_dict*, *out_file=None*, *reorient=True*)

Append substructure within shell to PDB.

Parameters

- **mol** (*RDKit Mol*) – Input mol
- **shell** (*Shell*) – A shell
- **atom_coords** (*dict*) – Dict matching atom id to coordinates.
- **bound_atoms_dict** (*dict*) – Dict matching atom id to id of bound atoms.
- **out_file** (*str or None, optional*) – File to which to append coordinates.
- **reorient** (*bool, optional*) – Use the transformation matrix in the shell to align by the stereo quadrants. If no transformation matrix present, centers the center atom.

Returns
list of str

Return type
list of PDB file lines, if *out_file* not specified

e3fp.fingerprint.util module

Utility methods and class for fingerprinting-related functions.

Author: Seth Axen E-mail: seth.axen@gmail.com

exception E3FPBitsValueError

Bases: *E3FPError*, ValueError

Bits value is invalid.

exception E3FPCountsError

Bases: *E3FPError*, ValueError

Index in counts is invalid.

exception E3FPInvalidFingerprintError

Bases: *E3FPError*, TypeError

Fingerprint is incorrectly formatted.

exception E3FPMolError

Bases: *E3FPError*, TypeError

Mol is of incorrect type.

exception E3FPOptionError

Bases: *E3FPError*, ValueError

Option provided is invalid.

Module contents

Submodules

e3fp.pipeline module

Functions for various pipeline use cases.

Author: Seth Axen E-mail: seth.axen@gmail.com

confs_from_smiles(*smiles*, *name*, *confgen_params={}*, *save=False*)
Generate conformations of molecule from SMILES string.

fprints_from_fprints_dict(*fprints_dict*, *level=- 1*)
Get fingerprint at *level* from dict of level to fingerprint.

fprints_from_mol(*mol*, *fprint_params={}*, *save=False*)
Generate fingerprints for all *first* conformers in mol.

fprints_from_sdf(*sdf_file*, *fprint_params={}*, *save=False*)
Generate fingerprints from conformers in an SDF file.

fprints_from_smiles(*smiles*, *name*, *confgen_params={}*, *fprint_params={}*, *save=False*)
Generate conformers and fingerprints from a SMILES string.

params_to_dicts(*params*)
Get params dicts for pipeline functions from INI format params file.

sdf_from_smiles(*smiles*, *name*, *confgen_params={}*, *out_file=None*, *out_ext='.sdf.bz2'*)
Generate conformations from SMILES string and save to SDF file.

e3fp.util module

Utility classes/methods.

Author: Seth Axen E-mail: seth.axen@gmail.com

exception E3FPDeprecationWarning
Bases: *E3FPWarning*, DeprecationWarning

A warning class for a deprecated method or class.

exception E3FPEfficiencyWarning
Bases: *E3FPWarning*, RuntimeWarning

A warning class for a potentially inefficient process.

exception E3FPError
Bases: Exception

Base class for E3FP-specific errors.

This class is provided for future E3FP-specific functionality.

exception E3FPWarning
Bases: Warning

Base E3FP warning class.

Unlike normal warnings, these are by default always set to on.

class deprecated(*deprecated_version*, *remove_version=None*, *msg=None*)

Bases: `object`

Decorator to mark a function as deprecated.

Issue a deprecation warning when a function is called, and update the documentation. A deprecation version must be provided.

Examples

```
>>> from e3fp.util import deprecated
>>> @deprecated("1.1", remove_version="1.3",
...             msg="Function no longer needed")
... def my_function():
...     pass
```

Notes

Adapted from https://wiki.python.org/moin/PythonDecoratorLibrary

deprecate_function(*f*)

Return the decorated function.

update_docstring(*obj*)

Add deprecation note to docstring.

maybe_jit(**args*, ***kwargs*)

Decorator to jit a function using Numba if available.

Usage is identical to *numba.jit*.

Module contents

PYTHON MODULE INDEX

e

INDEX

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W