Soprano Documentation

Release alpha (Andrews)

Simone Sturniolo

October 26, 2016

CONTENTS

Contents:

CHAPTER

ONE

SOPRANO

soprano package

Soprano 0.5.0

A Python library to crack crystals by Simone Sturniolo

Subpackages

soprano.analyse package

Contains all modules, classes and functions relevant to analysing collections after calculations have been performed.

Subpackages

soprano.analyse.phylogen package Module containing functions and classes for phylogenetic clustering of collections.

Submodules

soprano.analyse.phylogen.genes module Definitions for the various genes used by PhylogenCluster

class `soprano.analyse.phylogen.genes.`**`Gene`**(*name*, *weight=1.0*, *params={}*, *parser=None*, *pair=False*)

Bases: `object`

A description of a property, a 'gene', characterizing a structure, to be used with a PhylogenCluster. A number of default genes is provided, but custom ones can be created as well by passing a parser. Only default genes can be used in a .genefile with the phylogen.py script though.

Args:
name (str): name of the gene. Must be one of the existing ones or a
custom one (in which case a parser must be provided as
well). Custom names can't conflict with existing ones
weight (float): weight of the gene to be applied, default is 1.0
params (dict): additional parameters to be passed to the gene parser
function; when not specified, defaults will be used
parser (function<AtomsCollection, **kwargs>
=> np.array): parser function to be used when defining custom
genes. Must return a two-dimensional Numpy array
(axis 0 for the elements of the collection,
axis 1 for the values of the gene)
is_pair (bool): False if the gene returns a multi dimensional point
for each structure, True if it only returns pair
distances. Default is False

evaluate(*c*)
Evaluate the gene on a given AtomsCollection

is_pair
Whether the gene can only compare a pair of structures or can also give an absolute value for each structure individually (required for k-means clustering)

class soprano.analyse.phylogen.genes.**GeneDictionary**
Bases: object

Container class holding gene definitions

classmethod get_gene(*g*)
Get the definition for a given gene

classmethod help(*g*)
Get an help string for a given gene

exception soprano.analyse.phylogen.genes.**GeneError**
Bases: exceptions.Exception

soprano.analyse.phylogen.genes.**load_genefile**(*gfile*)
Load a gene file and return the (validated) list of genes contained within.

Args:
gfile (file or str): file to parse

Returns:
genelist: a list of genes parsed from the given file, ready to be
passed to a PhylogenCluster

soprano.analyse.phylogen.genes.**parsegene_energy**(*c*)

soprano.analyse.phylogen.genes.**parsegene_hbonds_angle**(*c*)

soprano.analyse.phylogen.genes.**parsegene_hbonds_fprint**(*c*)

soprano.analyse.phylogen.genes.**parsegene_hbonds_length**(*c*)

soprano.analyse.phylogen.genes.**parsegene_hbonds_site_compare**(*c*)

soprano.analyse.phylogen.genes.**parsegene_hbonds_site_reference**(*c*, *ref=None*)

soprano.analyse.phylogen.genes.**parsegene_hbonds_totn**(*c*)

soprano.analyse.phylogen.genes.**parsegene_latt_abc**(*c*)

soprano.analyse.phylogen.genes.**parsegene_latt_ang**(*c*)

soprano.analyse.phylogen.genes.**parsegene_latt_cart**(*c*)

soprano.analyse.phylogen.genes.**parsegene_linkage_list**(*c*, *size=0*)

soprano.analyse.phylogen.genes.**parsegene_mol_com**(*c*, *Z=0*)

soprano.analyse.phylogen.genes.**parsegene_mol_m**(*c*, *Z=0*)

soprano.analyse.phylogen.genes.**parsegene_mol_num**(*c*)

soprano.analyse.phylogen.genes.**parsegene_mol_rot**(*c*, *Z=0*, *twist_axis=None*, *swing_plane=None*)

soprano.analyse.phylogen.mapping module 2D mapping algorithms

soprano.analyse.phylogen.mapping.**classcond_principal_component**(*p*)

soprano.analyse.phylogen.mapping.**optimal_discriminant_plane**(*p*)

soprano.analyse.phylogen.mapping.**standard_classcond_component**(*p*)

soprano.analyse.phylogen.mapping.**total_principal_component**(*p*)

soprano.analyse.phylogen.phylogenclust module Phylogenetic clustering class definitions

class soprano.analyse.phylogen.phylogenclust.**PhylogenCluster**(*coll*, *genes=None*, *norm_range=(0.0, 1.0)*, *norm_dist=1.0*)

Bases: object

An object that, given an AtomsCollection and a series of "genes" and weights, will build clusters out of the structures in the collection based on their reciprocal positions as points in a multi-dimensional space defined by those "genes".

Initialize the PhylogenCluster object.

Args:

coll (AtomsCollection): an AtomsCollection containing the
structures that should be classified.
This will be copied and frozen for the
entirety of the life of this instance;
in order to operate on a modified
collection, a new PhylogenCluster should
be created.

genes (list[tuple], str, file): list of the genes that should be
loaded immediately; each gene
comes in the form of a tuple

(name (str), weight (float),
params (dict)). A path or open
file can also be passed for a
.gene file, from which the values
will be loaded.
norm_range (list[float?]): ranges to constrain the values of
single genes in between. Default is
(0, 1). A value of "None" in either
place can be used to indicate no
normalization on one or both sides.
norm_dist (float?): value to normalize distance genes to. These
are the genes that only make sense on pairs of
structures. Their minimum value is always 0.
This number would become their maximum value,
or can be set to None to avoid normalization.

create_mapping(*method=u'total-principal'*)

Return an array of 2-dimensional points representing a reduced dimensionality mapping of the given genes using the algorithm of choice. All algorithms are described in [W. Siedlecki et al., Patt. Recog. vol. 21, num. 5, pp. 411 429 (1988)].

Args:
method (str): can be one of the following algorithms:
- total_principal (default)
- clafic
- fukunaga-koontz
- optimal-discriminant

get_distmat()

Get the distance matrix between structures in the collection, based on the genes currently in use.

Returns:
distmat (np.ndarray): a (collection.length, collection.length)
array, containing the overall distance
(the norm of all individual gene distances)
between all pairs of structures.

get_genome_matrices()

Return the genome matrices in raw form (not normalized). The matrices refer to genes that only allow to define a distance between structures. The element at i,j represents the distance between said structures. The matrix is symmetric and has null diagonal.

Returns:
genome_matrix (np.ndarray): a (collection.length,

collection.length, gene.length)
array, containing the distances for
each gene and pair of structures in
row and column
genome_legend (list[tuple]): a list of tuples containing (name,
length) of the gene fragments in the
array

get_genome_matrices_norm()
Return the genome matrices in normalized and weighted form. The matrices refer to genes that only allow to define a distance between structures. The element at i,j represents the distance between said structures. The matrix is symmetric and has null diagonal.

Returns:
genome_matrix (np.ndarray): a (collection.length,
collection.length, gene.length)
array, containing the distances for
each gene and pair of structures in
row and column
genome_legend (list[tuple]): a list of tuples containing (name,
length) of the gene fragments in the
array

get_genome_vectors()
Return the genome vectors in raw form (not normalized). The vectors refer to genes that allow to define a specific point for each structure.

Returns:
genome_vectors (np.ndarray): a (collection.length, gene.length)
array, containing the whole extent
of the gene values for each structure
in the collection on each row
genome_legend (list[tuple]): a list of tuples containing (name,
length) of the gene fragments in the
array

get_genome_vectors_norm()
Return the genome vectors in normalized and weighted form. The vectors refer to genes that allow to define a specific point for each structure.

Returns:
genome_vectors (np.ndarray): a (collection.length, gene.length)
array, containing the whole extent
of the gene values for each structure

in the collection on each row
genome_legend (list[tuple]): a list of tuples containing (name,
length) of the gene fragments in the
array

get_hier_clusters(*t*, *method=u'single'*)

Get multiple clusters (in the form of a list of collections) based on the hierarchical clustering methods and the currently set genes.

Calls scipy.cluster.hierarchy.fcluster

Args:
t (float): minimum distance of separation required to consider
two clusters separate. This controls the number of
clusters: a smaller value will produce more fine
grained clustering. At the limit, a value smaller than
the distance between the two closest structures will
return a cluster for each structure. Remember that the
'distances' in this case refer to distances between the
'gene' values attributed to each structure. In other
words they are a function of the chosen genes,
normalization conditions and weights employed.
In addition, the way they are calculated depends on the
choice of method.
method (str): clustering method to employ. Valid entries are
'single', 'complete', 'weighted' and 'average'.
Refer to Scipy documentation for further details.

Returns:
clusters (tuple(list[int],
list[slices])): list of cluster index for each
structure (counting from 1) and
list of slices defining the
clusters as formed by hierarchical
algorithm.

get_hier_tree(*method=u'single'*)

Get a tree data structure describing the clustering order of based on the hierarchical clustering methods and the currently set genes.

Calls scipy.cluster.hierarchy.to_tree

Args:
method (str): clustering method to employ. Valid entries are
'single', 'complete', 'weighted' and 'average'.
Refer to Scipy documentation for further details.

Returns:
root_node (ClusterNode): the root node of the tree. Access child
members with .left and .right, while .id
holds the number of the corresponding
cluster. Refer to Scipy documentation for
further details.

get_kmeans_clusters(*n*)
Get a given number of clusters (in the form of a list of collections) based on the k-means clustering methods and the currently set genes. Warning: this method only works if there are no genes that work only with pairs of structures - as specific points, and not just distances between them, are required for this algorithm.

Calls scipy.cluster.vq.kmeans

Args:
n (int): the desired number of clusters.

Returns:
clusters (tuple(list[int],
list[slices])): list of cluster index for each
structure (counting from 1) and
list of slices defining the
clusters as formed by k-means
algorithm.

get_linkage(*method=u'single'*)
Get the linkage matrix between structures in the collection, based on the genes currently in use. Only used in hierarchical clustering.

Calls scipy.cluster.hierarchy.linkage.

Args:
method (str): clustering method to employ. Valid entries are
'single', 'complete', 'weighted' and 'average'.
Refer to Scipy documentation for further details.

Returns:
Z (np.ndarray): linkage matrix for the structures in the
collection. Refer to Scipy documentation for
details about the method

get_max_cluster_dist()
Return the maximum possible distance between two clusters

static load(*filename*)
Load a pickled copy from a given file path

save (*filename*)
Simply save a pickled copy to a given file path

save_collection (*filename*)
Save as pickle the collection bound to this PhylogenCluster. The calculated genes are also stored in it as arrays for future use.

set_genes (*genes*, *load_arrays=False*)
Calculate, store and set a list of genes as used for clustering.

Args:
genes (list[soprano.analyse.phylogen.Gene],
file, str): a list of Genes to calculate and store. A path
or open file can also be passed for a .gene
file, from which the values will be loaded.
load_arrays (bool): try loading the genes as arrays from the
collection before generating them. Warning:
if there are arrays named like genes but with
different contents this can lead to
unpredictable results.

soprano.calculate package

Contains all modules, classes and functions relevant to calculating properties of existing structures and collections, from basic ones to energetic and spectroscopic properties.

Subpackages

soprano.calculate.gulp package Classes and functions to carry out calculations using the bindings to GULP (General Utility Lattice Program), a software providing a lot of useful calculations with empirical force fields, partial charge calculations, Ewald summation of Coulombic interactions and more. GULP can be found at:

http://nanochemistry.curtin.edu.au/gulp/

It needs to be installed on your system to use any of the functionality provided here.

Submodules

soprano.calculate.gulp.charges module Get charges using GULP

soprano.calculate.gulp.charges.**get_gulp_charges** (*s*, *charge_method=u'eem'*, *save_charges=True*, *gulp_command=u'gulp'*, *gulp_path=None*)
Calculate the atomic partial charges using GULP.

Parameters:
s (ase.Atoms): the structure to calculate the energy of
charge_method (Optional[str]): which method to use for atomic partial
charge calculation. Can be any of

'eem', 'qeq' and 'pacha'.
Default is 'eem'.
save_charges (Optional[bool]): whether to save or not the charges in
the given ase.Atoms object. Default is
True.
gulp_command (Optional[str]): command required to call the GULP
executable.
gulp_path (Optional[str]): path where the GULP executable can be
found. If not present, the GULP command
will be invoked directly (assuming the
executable is in the system PATH).

Returns:
charges(np.array(float)): per-atom partial charges

soprano.calculate.gulp.w99 module Classes and functions for using the W99 force field in GULP. This force field only applies to organic molecules. More information can be found in the original paper by Donald E. Williams:

D.E. Williams, *Improved Intermolecular Force Field for Molecules Containing H, C, N, and O Atoms, with Application to Nucleoside and Peptide Crystals* - Journal of Computational Chemistry, Vol. 22, No. 11, 1154-1166 (2001)

exception soprano.calculate.gulp.w99.**W99Error**
Bases: exceptions.Exception

soprano.calculate.gulp.w99.**find_w99_atomtypes**(*s*, *force_recalc=False*)
Calculate the W99 force field atom types for a given structure.

Parameters:
s (ase.Atoms): the structure to calculate the atomtypes on
force_recalc (bool): whether to recalculate the molecules even if
already present. Default is False.

soprano.calculate.gulp.w99.**get_w99_energy**(*s*, *charge_method=u'eem'*, *Etol=1e-06*, *gulp_command=u'gulp'*, *gulp_path=None*, *save_charges=False*)
Calculate the W99 force field energy using GULP.

Parameters:
s (ase.Atoms): the structure to calculate the energy of
charge_method (Optional[str]): which method to use for atomic partial
charge calculation. Can be any of
'eem', 'qeq' and 'pacha'.
Default is 'eem'.
Etol (Optional[float]): tolerance on energy for intermolecular
potential cutoffs (relative to single
interaction energy). Default is 1e-6 eV.
gulp_command (Optional[str]): command required to call the GULP

executable.
gulp_path (Optional[str]): path where the GULP executable can be
found. If not present, the GULP command
will be invoked directly (assuming the
executable is in the system PATH).
save_charges (Optional[bool]): whether to retrieve also the charges
and save them in the Atoms object.
False by default.

Returns:
energy (float): the calculated energy

soprano.calculate.xrd package Classes and functions for simulating X-ray diffraction spectroscopic results from structures.

Submodules

soprano.calculate.xrd.sel_rules module Providing an interface to selection rules for XRD peaks and various space-groups.

soprano.calculate.xrd.sel_rules.**get_sel_rule_from_hall**(*h*)
Generate a function object that acts as a selection rule for XRD lines for the given symmetry group expressed in Hall number notation

Args:
h (int): Hall number of the required spacegroup

Returns:
rule_func (function< list<int> >
=> <bool>): a function that can be used to test triples of
Miller indices h,k,l to verify whether the
related plane gives rise or not to a peak

Raises:
RuntimeError: if the database of XRD selection rules or that of
Hall numbers was not properly loaded
ValueError: if the passed argument is invalid

soprano.calculate.xrd.sel_rules.**get_sel_rule_from_international**(*n*, *o=u'all'*)
Generate a function object that acts as a selection rule for XRD lines for the given symmetry group expressed in international number notation

Args:
n (int): International number of the required spacegroup

o (Optional[int]): Sub-option of the required spacegroup

Returns:
rule_func (function< list<int> >
=> <bool>): a function that can be used to test triples of
Miller indices h,k,l to verify whether the
related plane gives rise or not to a peak

Raises:
RuntimeError: if the database of XRD selection rules was not properly
loaded
ValueError: if some of the passed arguments are invalid

soprano.calculate.xrd.xrd module Classes and functions for simulating X-ray diffraction spectroscopic results from structures.

class soprano.calculate.xrd.xrd.**XRDCalculator**(*lambdax=1.54056*, *theta2_digits=6*, *baseline=0.0*, *peak_func=None*, *peak_f_args=None*)

Bases: object

A class implementing methods for XRD simulations, comparisons and fittings.

Initialize the XDRCalculator object's main parameters

Args:
lambdax (Optional[float]): X-ray wavelength in Angstroms
(default is 1.54056 Ang)
theta2_digits (Optional[int]): Rounding within which
two theta angles (in degrees)
are considered to be equivalent
(default is 6 digits) when
calculating theoretical peaks
baseline (Optional[float]): baseline to use as starting point for
simulated spectra
peak_func (Optional[function<float, float, *kargs>
=> <np.ndarray>]): the function used to
simulate peaks. Should take
th2 as its first argument,
peak centre as its second,
and any number of optional
arguments. Returns a numpy
array containing the peak
shape. Should be able to
work with numpy arrays as
input
peak_f_args (Optional[list<float>]): optional arguments for

peak_func. If no peak_func
has been supplied by the
user, the first value will
be used as the Gaussian width

dataset_range(*xpeaks*, *theta2_range=(None*, *None)*)

Restrict the given dataset (XraySpectrum or XraySpectrumData) to only the values that lie within a certain theta2 range.

Args:
xpeaks (XraySpectrum or XraySpectrumData): the dataset to modify
theta2_range (tuple<int>): a tuple indicating minimum and maximum
of the desired theta2 range (degrees).
A value of None means no boundary

Returns:
xpeaks_restrict (XraySpectrum or XraySpectrumData): the restricted
dataset

Raises:
ValueError: if some of the values passed are invalid

exp_dataset(*th2_axis*, *int_axis*)
Build an experimental dataset as an XraySpectrumData object.

Args:
th2_axis (np.ndarray): array containing the values for 2*theta
int_axis (np.ndarray): array containing the values for intensity

Returns:
exp_spec (XraySpectrumData): named tuple containing the
experimental dataset

Raises:
ValueError: if some of the values passed are invalid

lebail_fit(*xpeaks*, *exp_spec*, *rwp_tol=0.01*, *max_iter=100*)
Perform a refining on an XraySpectrum object's intensities based on experimental data with leBail's method.

Args:

xpeaks (XraySpectrum): object containing the details of the XRD
peaks
exp_spec (XraySpectrumData): experimental data, dataset built
using xrd_exp_dataset
rwp_tol (Optional[float]): tolerance on the Rwp error value
between two iterations that stops the
calculation. Default is 1e-2
max_iter (Optional[int]): maximum number of iterations to perform

Returns:
xpeaks_scaled (XraySpectrum): a new XraySpectrum object, with
intensities properly scaled to match
the experimental data
simul_spec (np.ndarray): final simulated XRD spectrum
simul_peaks (np.ndarray): final simulated spectrum broken by peak
contribution along axis 1
rwp (float): the final value of Rwp (fitness of simulated to
experimental data)

Raises:
ValueError: if some of the arguments are invalid

peak_f_args
Additional arguments to be passed to peak_func

peak_func
The function used to build peaks in simulated spectra

Should be of form peak_func(theta2, peak_position, *peak_f_args)

powder_peaks(*atoms=None*, *latt_abc=None*, *n=1*, *o=u'all'*)
Calculate the peaks (without intensities) of a powder XRD spectrum given either an Atoms object or the lattice in ABC form and the spacegroup indices to apply the selection rules

Args:
atoms (Optional[soprano.Atoms]): atoms object to gather lattice
and spacegroup information from
latt_abc (Optional[np.ndarray]): periodic lattice in ABC form,
Angstroms and radians
n (Optional[int]): International number of the required spacegroup
o (Optional[int]): Sub-option of the required spacegroup

Returns:
xpeaks (XraySpectrum): a named tuple containing the peaks
with theta2, corresponding hkl indices,
a unique hkl tuple for each peak,

inverse reciprocal lattice distances,
intensities and wavelength

Raises:
ValueError: if some of the arguments are invalid

set_peak_func(*peak_func=None*, *peak_f_args=None*)
Set a new peak_func for this XDRCalculator. If no new function is passed, reset the default Gaussian function.

Args:
peak_func (Optional[function<float, float, *kargs>
=> <np.ndarray>]): the function used to
simulate peaks. Should
take th2 as its first
argument, peak centre as
its second, and any
number of optional
arguments. Returns a
numpy array containing
the peak shape. Should
be able to work with
numpy arrays as input
peak_f_args (Optional[list<float>]): optional arguments for
peak_func. If no peak_func
has been supplied by the
user, the first value will
be used as the Gaussian
width

spec_simul(*xpeaks*, *th2_axis*)
Simulate an XRD spectrum given positions of peaks, intensities, baseline, and a peak function (a Gaussian by default).

Args:
xpeaks (XraySpectrum): object containing the details of the XRD
peaks
th2_axis (np.ndarray): theta2 axis points on which the
spectrum should be simulated

Returns:
simul_spec (XraySpectrumData): simulated XRD spectrum
simul_peaks (np.ndarray): simulated spectrum intensities broken by
peak contribution along axis 1

Raises:
ValueError: if some of the arguments are invalid

class soprano.calculate.xrd.xrd.**XraySpectrum**(*theta2*, *hkl*, *hkl_unique*, *invd*, *intensity*, *lambdax*)

Bases: tuple

hkl
Alias for field number 1

hkl_unique
Alias for field number 2

intensity
Alias for field number 4

invd
Alias for field number 3

lambdax
Alias for field number 5

theta2
Alias for field number 0

class soprano.calculate.xrd.xrd.**XraySpectrumData**(*theta2*, *intensity*)

Bases: tuple

intensity
Alias for field number 1

theta2
Alias for field number 0

soprano.collection package

Contains all modules, classes and functions relevant to handling, loading or randomly generating collections of structures.

Subpackages

soprano.collection.generate package This module contains generators meant to produce AtomsCollections based on different criteria.

Submodules

soprano.collection.generate.airss module Bindings for AIRSS Buildcell program for random structure generation

soprano.collection.generate.airss.**airssGen**(*input_file*, *n=100*, *buildcell_command=u'buildcell'*, *buildcell_path=None*)

Generator function binding to AIRSS' Buildcell.

This function searches for a buildcell executable and uses it to generate multiple new Atoms structures for a collection.

Args:
input_file (str or file): the .cell file with appropriate comments
specifying the details of buildcell's
construction work.
n (int): number of structures to generate. If set to None the
generator goes on indefinitely.
buildcell_command (str): command required to call the buildcell
executable.
buildcell_path (str): path where the buildcell executable can be
found. If not present, the buildcell command
will be invoked directly (assuming the
executable is in the system PATH).

Returns:
airssGenerator (generator): an iterable object that yields structures
created by buildcell.

soprano.collection.generate.linspace module Generator producing structures interpolated between two extremes

soprano.collection.generate.linspace.**linspaceGen**(*struct_0*, *struct_1*, *steps=10*, *periodic=False*)
Generator function to create multiple structures with positions interpolated linearly between two extremes.

Args:
struct_0 (ase.Atoms): the starting structure
struct_1 (ase.Atoms): the final structure. The atoms should be in the
same order as the ones in struct_0
steps (Optional[int]): number of interpolated steps to produce
(extremes included). Default is 10
periodic (Optional[bool]): if True the interpolation will take into
account periodic boundaries and interpolate
between positions in struct_0 and the
closest periodic copy of positions in
struct_1. By default set to False

Returns:
linspaceGenerator (generator): an iterator object that yields
structures created by linear
interpolation.

soprano.collection.generate.rattle module Generator producing structures rattled of a given amount

soprano.collection.generate.rattle.**rattleGen**(*struct*, *amplitude=0.01*, *n=100*, *method=u'uniform'*)
Generator function to create multiple structures by randomly displacing atoms of a given amount.

Args:

struct (ase.Atoms): the starting structure to randomize

amplitude (float or np.ndarray): the amplitude of the random displacement. Can be a single float for all atoms, a 1D numpy array of length N (N being the number of atoms, one value each) or a 2D numpy array of shape (N,3) (one value for each dimension). These values are used as interval for uniform random numbers and as stdev for normal random numbers

n (int): maximum number of structures to generate. If set to None will generate infinite structures

method (str): must be either 'uniform' or 'normal'. In the first case the rattling will be a uniform random number between +amplitude and -amplitude. In the second case it will be a gaussian random number with +amplitude standard deviation.

Returns:

rattleGenerator (generator): an iterator that yields copies of the base structure with randomly displaced atoms.

Submodules

soprano.collection.collection module Definition of the Collection class.

It handles multiple Atoms ASE objects and mirrors in this sense the structure of the Atoms object itself.

class soprano.collection.collection.**AtomsCollection**(*structures=[]*, *info={}*, *cell_reduce=False*, *progress=False*)

Bases: object

AtomsCollection object.

An AtomsCollection represents a group of ASE Atoms objects. It handles them together, can perform mass operations on them, and stores arrays of informations related to them.

Initialize the AtomsCollection

Args:

structures (list[str] or list[ase.Atoms]): list of file names or Atoms that will form the collection

info (dict): dictionary of general information to attach to this collection

cell_reduce (bool): if True, perform a Niggli cell reduction on
all loaded structures
progress (bool): visualize a progress bar for the loading process

all

chunkify (*chunk_size=None*, *chunk_n=None*)
Split this collection into multiple collections based on either size or number of chunks.

Args:
chunk_size (Optional[int]): maximum size of a generated chunk
chunk_n (Optional[int]): number of chunks to generate

Returns:
chunks (list[AtomsCollection]): a list of the generated chunks

filter (*filter_func*)
Return a collection composed only of the elements for which a given filter function returns True.

Args:
filter_func (function<Atoms>
=> bool): filter function. Should take an
Atoms object and return a boolean

Returns:
filtered (AtomsCollection): the filtered version of the collection

get_array (*name*, *copy=True*)
Get a copy of an array of given name (or a reference if copy=False)

Args:
name (str): name of the array to retrieve.
copy (bool): if the array should be copied or a reference should
be returned instead.

Returns:
array (np.ndarray): the requested array

has (*name*)
Check if array of given name exists

length

static **load**(*filename*)
Load a pickled copy from a given file path

run_calculators(*properties=None*, *system_changes=None*)
Run all previously set ASE calculators.

Args:
properties (list[str]): list of properties to calculate (depends
on type of Calculator used)
system_changes (list[str]): list of changes to the structure
since the last calculation. Can be
any combination of these five:
'positions', 'numbers', 'cell',
'pbc', 'initial_charges' and
'initial_magmoms'.

save(*filename*)
Simply save a pickled copy to a given file path

set_array(*name*, *a*, *dtype=None*, *shape=None*, *args={}*)
Add or modify an array of data related to the Atoms objects in this collection.

Args:
name (str): name of the array to operate on.
a (np.ndarray or function<Atoms, **kwargs>
=> Any): the data to assign to the array (must
be same length as the collection) or
a function that takes an Atoms object
as the first argument and returns a
value. This will be mapped over the
structures to create the array.
dtype (type): type to cast the values of the array to.
shape (tuple [int]): shape of each entry of the array. Will be
checked if provided.
args (dict): named arguments to pass to the function provided
as a. Will be ignored if an array is passed instead.

set_calculators(*calctype*, *labels=None*, *params={}*)
Set an ASE calculator on each structure in the collection, and set said calculator's parameters.

Args:
calctype (ASE Calculator type): the type of calculator
to instantiate.
labels (Optional[list[str]]): names to use for the calculators'
files. If not present, random
generated names are used.

params (Optional[dict]): parameters of the calculator to set.

sorted_byarray(*name*, *reverse=False*)
Return a copy of this collection sorted by a given array.

Args:
name (str): name of the array to use for the sorting
reverse (Optional[bool]): reverse order of sorting (max to min)

Returns:
sorted (AtomsCollection): a sorted copy of the collection

soprano.hpc package

Classes and functions useful to run calculations on huge cluster systems (High Performance Computation). To be used with care - these are liable to fail if some specific architecture has quirks that are not accounted for!

Subpackages

soprano.hpc.submitter package Classes and functions required for processes that automatically submit jobs to a queueing system working in the background.

These can be launched interactively from the command line. In order to do that:

1. write your own implementation of a submitter class by inheriting from soprano.hpc.submitter.Submitter or use one of the provided ones;
2. write an input file in which you simply create an instance of said class and set up its parameters (ideally by calling the set_parameters method);
3. launch that submitter from the command line with the following command:

```
python -m soprano.hpc.submitter start <filename>
```

You can have multiple submitter instances, even of different types, defined in the same file: in that case you will need to use the -n option to specify which one you want to launch (the name you need to use is the name of the *variable* you stored the instance in). If you are working on remote login and you want to prevent the submitter from being terminated upon exiting your session use the -nohup option. To list which submitters from a given file are running, and how long have they been running for, just use:

```
python -m soprano.hpc.submitter list <filename>
```

Similarly, you can stop a running submitter with:

```
python -m soprano.hpc.submitter stop <filename>
```

Submitters have a 'name' property and will save a <name>.log file in which any output from their run can be stored.

Subpackages

soprano.hpc.submitter.debug package Functions useful for debugging QueueInterface and Submitters. These provide a 'fake' queue that executes basic jobs with artificial delays in order to simulate an environment similar to what can be found on an HPC machine.

Submodules

soprano.hpc.submitter.debug.debugqueue module Definition of a fake QueueInterface class, useful for debugging Submitters.

class soprano.hpc.submitter.debug.debugqueue.**DebugQueueInterface**(*dt=0.1*)

Bases: *soprano.hpc.submitter.queues.QueueInterface*

DebugQueueInterface object

A class meant to emulate a QueueInterface while doing absolutely nothing of what it does. Jobs are simply stored locally, there's a fixed waiting time, and are then executed. Ideally they should be simple, quick stuff (like an echo command). No guarantees for actually long jobs.

In the submitted script a syntax for additional variables is allowed, similar to real queue systems. These follow the convention of many engines of having to start with #$. For example

#$ WAIT 10

means the job will be put in a "wait" state for 10 seconds. The currently available variables are:

WAIT - specify how long the job has to stay in a "wait" state. If two values are provided, these are considered bounds for a random number RUN - same as above, but for the running state. This has no bearing on the *actual* running time (it's suggested that it's something very quick)

Initialize the DebugQueueInterface.

Args:
dt (float): frequency with which the queue status is updated

kill(*job_id*)
Kill the job with the given ID

Args:
job_id (str): ID of the job to kill

list()
List all jobs found in the queue

Returns:
jobs (dict): a dict of jobs classified by ID containing all info
that can be matched through list_outre

submit(*script*, *cwd=None*)
Submit a job to the queue.

Args:
script (str): content of the submission script
cwd (Optional[str]): path to the desired working directory

Returns:
job_id (str): the job ID assigned by the queue system and parsed
with sub_outre

Submodules

soprano.hpc.submitter.castep module Definition of CastepSubmitter class.

A basic "rolling" submitter for Castep calculations, grabbing from one folder and depositing results in another.

class soprano.hpc.submitter.castep.**CastepSubmitter**(*name*, *queue*, *submit_script*, *max_jobs=4*, *check_time=10*, *max_time=3600*, *temp_folder=None*)
Bases: *soprano.hpc.submitter.submit.Submitter*

Initialize the Submitter object

Args:
name (str): name to be used for this Submitter (two Submitters
with the same name can't be launched in the same
working directory)
queue (QueueInterface): object describing the properties of the
interface to the queue system in use
submit_script (str): text of the script to use when submitting a
job to the queue. All tags of the form <name>
will be replaced with the job's name, and all
similar tags of the form <[arg]> will be
replaced if the argument name is present in
the job's args dictionary
max_jobs (Optional[int]): maximum number of jobs to submit at a
given time. Default is 4
check_time (Optional[float]): time in seconds between consecutive
checks for the queue status and
attempts to submit new jobs. Default
is 10
max_time (Optional[float]): time in seconds the Submitter will run
for before shutting down. If set to
zero the thread won't stop until
killed with Submitter.stop.
temp_folder (Optional[str]): where to store the temporary folders

for the calculations. By default it's
the system's tmp/ folder, but might
be changed if there's a need because
of writing permissions.

finish_job(*name*, *args*, *folder*)
Save required output files to the output folder

finish_run()
Try removing the temporary keywords directory

next_job()
Grab the next job from folder_in

set_parameters(*folder_in, folder_out, castep_command, castep_path=None, copy_extensions=[u'.castep'], pspot_files=[], dryrun_test=False*)
Set the parameters of the CASTEP Submitter

Args:
folder_in (str): path of the folder to extract cell files from
folder_out (str): path of the folder where the results will be
saved
castep_command (str): command used to call the CASTEP executable
on this system
castep_path (Optional[str]): folder where the CASTEP executable is
located (if not part of the system
PATH)
pspot_files (Optional[list[str]]): additional pseudopotential
files to be copied in the input
temporary folders
copy_extensions (Optional[list[str]]): extensions of output files
to copy to the output
folder (by default only
.castep file)
dryrun_test (Optional[bool]): run a dryrun test on files before
actually running the calculation.
Off by default.

setup_job(*name*, *args*, *folder*)
Copy files to temporary folder to prepare for execution

start_run()

soprano.hpc.submitter.queues module Definition of QueueInterface class.

class soprano.hpc.submitter.queues.**QueueInterface**(*sub_cmd*, *list_cmd*, *kill_cmd*, *sub_outre*, *list_outre*)
Bases: object

QueueInterface object

A class meant to simplify interfacing in a basic way with a Queue system. Contains commands to submit to the queue, list the job IDs, and kill them if necessary. Will contain Regexps to parse for IDs and additional information as returned upon submission and listing. It is important that the regular expressions used employ NAMED GROUPS to parse the various fields. In particular, a job_id group must ALWAYS be present. The class also provides some static variables implementing standard interfaces for common queueing systems. These can be retrieved by using QueueInterface.<NAME>. The currently implemented names are the following:

- LSF (IBM's managing system, using the command bsub)
- **GridEngine (Sun's managing system, also available in an open version,** using the command qsub)

Initialize the QueueInterface.

Args:

sub_cmd (str): command used to submit a script to the queue
list_cmd (str): command used to list all queued jobs for the user
kill_cmd (str): command used to kill a job given its id
sub_outre (str): regular expression used to parse the output of
sub_cmd. Must contain at least a job_id named
group
list_outre (str): regular expression used to parse the output of
list_cmd. Must contain at least a job_id named
group

classmethod GridEngine()

classmethod LSF()

kill(*job_id*)

Kill the job with the given ID

Args:

job_id (str): ID of the job to kill

list()

List all jobs found in the queue

Returns:

jobs (dict): a dict of jobs classified by ID containing all info
that can be matched through list_outre

submit(*script*, *cwd=None*)

Submit a job to the queue.

Args:

script (str): content of the submission script
cwd (Optional[str]): path to the desired working directory

Returns:
job_id (str): the job ID assigned by the queue system and parsed
with sub_outre

soprano.hpc.submitter.submit module Definition of Submitter class

Base class for all Submitters to inherit from.

class soprano.hpc.submitter.submit.**Submitter**(*name*, *queue*, *submit_script*, *max_jobs=4*, *check_time=10*, *max_time=3600*, *temp_folder=None*)

Bases: object

Submitter object

Template to derive all specialised Submitters. These are meant to generate, submit and post-process any number of jobs on a queueing system in the form of a background process running on a head node. It implements methods that should be mostly overridden by the child classes. Six methods define its core behaviour:

1. next_job is the function that outputs the specification for each new job to submit. The specification should be a dict with two members, 'name' (a string) and 'args' (ideally a dict). If no more jobs are available it should return None;
2. setup_job takes as arguments name, args and folder (a temporary one created independently) and is supposed to generate the input files for the job before submission. It returns a boolean, confirming that the setup went well; if False, the job will be skipped;
3. check_job takes as arguments job ID, name, args and folder and should return a bool confirmation of whether the job has finished or not. By default it simply checks whether the job is still listed in the queue, however other checks can be implemented in its place;
4. finish_job takes as arguments name, args and folder and takes care of the post processing once a job is complete. Here meaningful data should be extracted and useful files copied to permament locations, as the temporary folder will be deleted immediately afterwards. It returns nothing;
5. start_run takes no arguments, executes at the beginning of a run;
6. finish_run takes no arguments, executes at the end of a run.

In addition, the Submitter takes a template launching script which can be tagged with keywords, mainly <name> for the job name or any other arguments present in args. These will be replaced with the appropriate values when the script is submitted.

Initialize the Submitter object

Args:
name (str): name to be used for this Submitter (two Submitters
with the same name can't be launched in the same
working directory)
queue (QueueInterface): object describing the properties of the
interface to the queue system in use
submit_script (str): text of the script to use when submitting a
job to the queue. All tags of the form <name>

will be replaced with the job's name, and all
similar tags of the form <[arg]> will be
replaced if the argument name is present in
the job's args dictionary
max_jobs (Optional[int]): maximum number of jobs to submit at a
given time. Default is 4
check_time (Optional[float]): time in seconds between consecutive
checks for the queue status and
attempts to submit new jobs. Default
is 10
max_time (Optional[float]): time in seconds the Submitter will run
for before shutting down. If set to
zero the thread won't stop until
killed with Submitter.stop.
temp_folder (Optional[str]): where to store the temporary folders
for the calculations. By default it's
the system's tmp/ folder, but might
be changed if there's a need because
of writing permissions.

check_job(*job_id*, *name*, *args*, *folder*)
Checks if given job is complete or not

finish_job(*name*, *args*, *folder*)
Performs completiion operations on the job. At this point any relevant output files should be copied from 'folder' to their final destination as the temporary folder itself will be deleted immediately after

finish_run()
Operations to perform after the daemon thread stops running

static list()

log(*logtxt*)

next_job()
Return a dictionary definition of the next job in line

set_parameters()
Set additional parameters. In this generic example class it has no arguments, but in specific implementations it will be used to add more variables without overriding __init__.

setup_job(*name*, *args*, *folder*)
Perform preparatory operations on the job

start()

start_run()
Operations to perform when the daemon thread starts running

static stop(*fname*, *subname*)
Stop Submitter process from filename and name, return False if failed

soprano.properties package

Contains classes, modules and functions relevant to Properties, a catch-all term for things we might want to extract or calculate from Atoms and AtomsCollections. Some will require running an external ASE calculator first, some will just work on their own, some will require some calculations and parameters.

Subpackages

soprano.properties.basic package Module containing very basic AtomsProperties (the kind that only require a couple of lines of code but are still pretty convenient to have at hand)

Submodules

soprano.properties.basic.basic module Implementation of some basic AtomsProperty classes

class soprano.properties.basic.basic.**CalcEnergy**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Property representing the energy calculated by an ASE calulator

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'calc_energy'

default_params = {}

static extract(*s*)

class soprano.properties.basic.basic.**LatticeABC**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Property representing the axis-angles form of a structure's lattice

Parameters:
shape (tuple): the shape to give to the array
deg (bool): whether to give the angles in degrees instead of radians

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the

property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'lattice_abc'

default_params = {u'shape': (2, 3), u'deg': False}

static extract(*s*, *shape*, *deg*)

class soprano.properties.basic.basic.**LatticeCart**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Property representing the Cartesian form of a structure's lattice

Parameters:
shape (tuple): the shape to give to the array

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'lattice_cart'

default_params = {u'shape': (3, 3)}

static extract(*s*, *shape*)

class soprano.properties.basic.basic.**NumAtoms**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Property representing the number of atoms in a structure

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'num_atoms'

default_params = {}

static extract(*s*)

soprano.properties.castep package Module containing AtomsProperties related specifically to CASTEP calculations. Some of these can be looked up only in a CASTEP Calculator; others require passing the path of the .castep file as a parameter and actually parsing its contents.

Submodules

soprano.properties.castep.castep module Implementation of some CASTEP related AtomsProperties

class soprano.properties.castep.castep.**CastepEnthalpy**(*name=None*, ***params*)

Bases: *soprano.properties.atomsproperty.AtomsProperty*

Enthalpy as found in the .castep file of a GeometryOptimization calculation. If not present, this will fall back on the final free energy.

Parameters:

castep_path (str): the path in which the .castep file is to be found.

seedname_info (str): the Atoms.info key that contains the seedname of the .castep file. By default is 'name'.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:

name (str): a name to give to this specific instance of the property (will be used to store it as array if requested)

params: named arguments specific to this type of property

default_name = u'castep_enthalpy'

default_params = {u'seedname_info': u'name', u'castep_path': u'.'}

static extract(*s*, *castep_path*, *seedname_info*)

soprano.properties.labeling package Module containing AtomsProperties that relate to labeling a system's atoms, molecules, hydrogen bonds and such based on their chemical properties.

Submodules

soprano.properties.labeling.labeling module Implementation of AtomsProperties that relate to labeling of systems

class soprano.properties.labeling.labeling.**HydrogenBondTypes**(*name=None*, ***params*)

Bases: *soprano.properties.atomsproperty.AtomsProperty*

Assign MoleculeSites labels to atoms, then characterise existing hydrogen bonds based on them, and return a list of such bonds detected in a system. The bonds come in the form '{0}<{1},{2}>..{3}<{4}>', where {0} is the name of the molecule containing the hydrogen, {2} is the hydrogen, {1} the atom to which the hydrogen is bonded, {3} the name of the other molecule and {4} the atom to which the hydrogen is hydrogen bonded.

Parameters:
force_recalc (bool): if True, always recalculate the molecules even if already present.
save_info (bool): if True, save the found hydrogen bond types as part of the Atoms object info. By default True.

Returns:
hydrogen_bond_types (list): A list containing info characterising the hydrogen bonds present in the system in a detailed way.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the property (will be used to store it as array if requested)
params: named arguments specific to this type of property

default_name = u'hydrogen_bond_types'

default_params = {u'force_recalc': False, u'save_info': True}

static extract(*s*, *force_recalc*, *save_info*)

class soprano.properties.labeling.labeling.**MoleculeSites**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Assigns univoque labels to atoms belonging to molecules by exploiting network topology. Atoms can have the same label, but only if they're fundamentally indistinguishable in the molecule's chemical context (for example, three hydrogen atoms on a CH3 group). The molecule will be described by a characteristic string and by a series of labels in the format [element]_[number]. These sites will be saved by default and can be used for better insight when carrying out other analysis.

Parameters:
force_recalc (bool): if True, always recalculate the molecules even if already present.
save_info (bool): if True, save the found molecular sites as part of the Atoms object info. By default True.
save_asarray (bool): if True the molecular site names are also saved as an array of the molecule selection.

Returns:
molecular_sites (dict): A dictionary containing info characterising
the molecule's chemical sites unequivocally.
These are a string representation of the
molecule itself and a dictionary linking
atomic indices (as found in the molecule in
AtomSelection form) to site labels.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'molecule_sites'

default_params = {u'force_recalc': False, u'save_info': True, u'save_asarray': False}

static extract(*s*, *force_recalc*, *save_info*, *save_asarray*)

soprano.properties.linkage package Module containing AtomsProperties that relate to linkage properties of a given system, self-correlation etc.

Submodules

soprano.properties.linkage.linkage module Implementation of AtomsProperties that relate to linkage of atoms

class soprano.properties.linkage.linkage.**Bonds**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Produces an array of tuples identifying all bonds existing within the system (calculated using Van der Waals radii). The tuples are structured as:

(atom_1, atom_2, atom_2_cell, bond_length)

with atom_1 and atom_2 being indices and atom_2_cell being an array of integers identifying the unit cell to which atom_2 belongs with respect to atom_1 (which is assumed to be in (0,0,0), the central cell). This is to account for the possibility of course that the bond exists through the periodic boundary. WARNING: the possibility of an atom bonding with another throughout two different periodic boundaries is not accounted for.

Parameters:
vdw_set({ase, jmol}): set of Van der Waals radii to use. Default is
the one extracted from JMol.
vdw_scale (float): scaling factor to apply to the base Van der Waals
radii values. Values bigger than one make for more
tolerant bonds.

default_vdw (float): default Van der Waals radius for species for
whom no data is available.

Returns:
bonds([tuple]): list of bonds in the form of 3-tuples structured as
explained above

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'bonds'

default_params = {u'default_vdw': 2.0, u'vdw_scale': 1.0, u'vdw_set': u'jmol'}

static extract(*s*, *vdw_set*, *vdw_scale*, *default_vdw*)

class soprano.properties.linkage.linkage.**CoordinationHistogram**(*name=None*, ***params*)

Bases: *soprano.properties.atomsproperty.AtomsProperty*

Produces an histogram representing, for each pair of species present in the system, how many atoms of species 1 have n bonds with species 2, n being the histogram bins. The histogram is topped at a 'maximum coordination' parameter which is 6 by default but can be user defined; the last bin represents all higher values (so by default '6 or more'). Two species or lists of species can be given if one wants to restrict the search; otherwise a full histogram for all pairs of species is returned.

Parameters:
vdw_set({ase, jmol}): set of Van der Waals radii to use. Default is
the one extracted from JMol.
vdw_scale (float): scaling factor to apply to the base Van der Waals
radii values. Values bigger than one make for more
tolerant bonds.
default_vdw (float): default Van der Waals radius for species for
whom no data is available.
species_1 (str or [str]): list of species to compute the histogram
for. By default all of them.
species_2 (str or [str]): list of species whose coordination with
species_1 should be checked. By default all
of them.
max_coord (int): what should be the largest coordination number
considered for an atom (default 6).

Returns:
coord_hist (dict): dictionary of dictionaries indexed by species_1
followed by species_2. The elements are arrays of
integers constituting the histogram.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'coord_histogram'

default_params = {u'species_2': None, u'species_1': None, u'default_vdw': 2.0, u'max_coord': 6, u'vdw_set': u'jmol'

static extract(*s*, *vdw_set*, *vdw_scale*, *default_vdw*, *species_1*, *species_2*, *max_coord*)

class soprano.properties.linkage.linkage.**HydrogenBonds**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Hydrogen Bonds

Produces a dictionary containing the atom indices defining hydrogen bonds detected in the system - if required, classified by type. By default only O and N atoms are considered for hydrogen bonds (OH..O, OH..N and so on). The type is defined as AH..B where A is the symbol of the atom directly bonded to the proton and B the one of the hydrogen bonded one.

Parameters:
vdw_set({ase, jmol}): set of Van der Waals radii to use. Default is
the one extracted from JMol.
vdw_scale (float): scaling factor to apply to the base Van der Waals
radii values. Values bigger than one make for more
tolerant molecules.
default_vdw (float): default Van der Waals radius for species for
whom no data is available.
hbond_elems ([str]): chemical symbols of elements considered capable
of forming hydrogen bonds (by default O and N)
max_length (float): maximum A-B length of the hydrogen bond in
Angstrom - default is 3.5 Ang
max_angle (float): maximum A-H/A-B angle in the hydrogen bond in
degrees - default is 45 deg
save_info (bool): if True, save the found hydrogen bonds as part of
the Atoms object info. By default True.

Returns:

hbondss ([dict]): list of hydrogen bonds detected
in the system by type (can contain empty arrays).
For each hydrogen bond we give index of the H
atom, index and unit cell of the A atom (the one
directly bonded), index and unit cell of the B atom
(the one that's hydrogen bonded), length and angle
in degrees.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'hydrogen_bonds'

default_params = {u'save_info': True, u'hbond_elems': [u'O', u'N'], u'default_vdw': 2.0, u'max_length': 3.5, u'vdw_

static extract(*s*, *vdw_set*, *vdw_scale*, *default_vdw*, *hbond_elems*, *max_length*, *max_angle*, *save_info*)

class soprano.properties.linkage.linkage.**HydrogenBondsNumber**(*name=None*, ***params*)

Bases: *soprano.properties.atomsproperty.AtomsProperty*

Number of hydrogen bonds detected in this system, classified by type. By default will use already existing hydrogen bonds if they're present as a saved array in the system.

Parameters:
force_recalc (bool): if True, always recalculate the hydrogen bonds
even if already present.

Returns:
hbonds_n (int): number of hydrogen bonds found

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'hydrogen_bonds_n'

default_params = {u'force_recalc': False}

static extract(*s*, *force_recalc*)

class soprano.properties.linkage.linkage.**LinkageList**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Produces an array containing the atomic pair distances in a system, reduced to their shortest periodic version and sorted min to max.

Parameters:
size (int): maximum number of distances to include. If not present,
all of them will be included. If present, arrays will be
cut or padded to reach this sizeber.

Returns:
link_list ([float]): sorted list of interatomic linkage distances

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'linkage_list'

default_params = {u'size': 0}

static extract(*s*, *size*)

class soprano.properties.linkage.linkage.**MoleculeCOMLinkage**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Linkage list - following the same criteria as the atomic one - calculated for the centers of mass of the molecules present in the system. By default will use already existing molecules if they're present as a saved array in the system.

Parameters:
force_recalc (bool): if True, always recalculate the molecules even if
already present.
size (int): maximum number of distances to include. If not present,
all of them will be included. If present, arrays will be
cut or padded to reach this sizeber.

Returns:
molecule_linkage ([float]): distances between all centers of mass of
molecules in the system, sorted.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'molecule_com_linkage'

default_params = {u'force_recalc': False, u'size': 0}

static extract(*s*, *force_recalc*, *size*)

class soprano.properties.linkage.linkage.**MoleculeMass**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Total mass of each of the molecules detected in this system. By default will use already existing molecules if they're present as a saved array in the system.

Parameters:
force_recalc (bool): if True, always recalculate the molecules even if
already present.
size (int): maximum number of distances to include. If not present,
all of them will be included. If present, arrays will be
cut or padded to reach this sizeber.

Returns:
molecule_m ([float]): mass of each of the molecules present, sorted.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'molecule_mass'

default_params = {u'force_recalc': False, u'size': 0}

static extract(*s*, *force_recalc*, *size*)

class soprano.properties.linkage.linkage.**MoleculeNumber**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Number of molecules detected in this system. By default will use already existing molecules if they're present as a saved array in the system.

Parameters:
force_recalc (bool): if True, always recalculate the molecules even if already present.

Returns:
molecule_n (int): number of molecules found

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the property (will be used to store it as array if requested)
params: named arguments specific to this type of property

default_name = u'molecule_n'

default_params = {u'force_recalc': False}

static extract(*s*, *force_recalc*)

class soprano.properties.linkage.linkage.**MoleculeRelativeRotation**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

A list of relative rotations between molecules. Uses the inertia tensor eigenvectors to establish a local frame for each molecule, then uses quaternions to define a rotational distance between molecules. It then produces a list of geodesic distances between these quaternions.

Parameters:
force_recalc (bool): if True, always recalculate the molecules even if already present.
size (int): maximum number of distances to include. If not present, all of them will be included. If present, arrays will be cut or padded to reach this size.
twist_axis ([float]): if present, only compare the Twist component of quaternion along the given axis. The Twist/Swing decomposition splits a quaternion in a rotation

around an axis and one around an orthogonal
direction. Only one between this and swing_plane
can be present.
swing_plane ([float]): if present, only compare the Swing component of
quaternion along the given axis. The Twist/Swing
decomposition splits a quaternion in a rotation
around an axis and one around an orthogonal
direction. Only one between this and twist_axis
can be present.

Returns:
molecule_relrot ([float]): list of relative rotations, as quaternion
distances, with the required ordering.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'molecule_rel_rotation'

default_params = {u'force_recalc': False, u'twist_axis': None, u'swing_plane': None, u'size': 0}

static extract(*s*, *force_recalc*, *size*, *swing_plane*, *twist_axis*)

class soprano.properties.linkage.linkage.**Molecules**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Produces an array containing multiple AtomSelection objects representing molecules in the system as found by connecting atoms closer than the half sum of their Van der Waals radii. It will return the entire unit cell if the system can not be split in molecules at all.

Parameters:
vdw_set({ase, jmol}): set of Van der Waals radii to use. Default is
the one extracted from JMol.
vdw_scale (float): scaling factor to apply to the base Van der Waals
radii values. Values bigger than one make for more
tolerant molecules.
default_vdw (float): default Van der Waals radius for species for
whom no data is available.
save_info (bool): if True, save the found molecules as part of the
Atoms object info. By default True.

Returns:
molecules ([AtomSelection]): list of molecules in the form of
AtomSelection objects.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'molecules'

default_params = {u'default_vdw': 2.0, u'vdw_scale': 1.0, u'save_info': True, u'vdw_set': u'jmol'}

static extract(*s*, *vdw_set*, *vdw_scale*, *default_vdw*, *save_info*)

soprano.properties.symmetry package Module containing AtomProperties that pertain to symmetry detection. Depends on having the Python bindings to SPGLIB installed on the system.

Submodules

soprano.properties.symmetry.symmetry module Implementation of AtomProperties that relate to symmetry

class soprano.properties.symmetry.symmetry.**SymmetryDataset**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Extracts SPGLIB's standard symmetry dataset from a given system, including spacegroup symbol, symmetry operations etc.

Parameters:
symprec (float): distance tolerance, in Angstroms, applied when
searching symmetry.

Returns:
symm_dataset (dict): dictionary of symmetry information

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if

requested)
params: named arguments specific to this type of property

default_name = u'symmetry_dataset'

default_params = {u'symprec': 1e-05}

static extract (*s*, *symprec*)

soprano.properties.transform package Module containing a special set of AtomsProperties that transform an Atoms object into another (by translating, rotating or mirroring all or some ions, and so on). These all accept an Atoms object and some parameters and return an Atoms object as well. Default behaviour for the .get method in most cases will be to do nothing at all, these properties are meant to be instantiated.

Submodules

soprano.properties.transform.transform module Implementation of AtomsProperties that transform the instance in some way

class soprano.properties.transform.transform.**Mirror**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Returns an Atoms object with some or all the atoms reflected with either a given center or a given plane. Absolute or scaled coordinates may be used.

Parameters:
selection (AtomSelection): selection object defining which atoms to
act on. By default, all of them.
center ([float]*3): center around which the reflection should take
place. By default the origin of the axes. Can't be
present at the same time as plane.
plane ([float]*4): plane with respect to which the reflection should
take place, in the form [a, b, c, d] parameters of
the plane equation.
By default is not used. Can't be present at the
same time as center.
scaled (bool): if True, treat the input vector as expressed in scaled,
not absolute, coordinates.

Returns:
reflected (ase.Atoms): Atoms object with the reflection performed.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the

property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'reflected'

default_params = {u'scaled': False, u'selection': None, u'center': None, u'plane': None}

static extract(*s*, *selection*, ***kwargs*)

class soprano.properties.transform.transform.**Regularise**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Regularize

Perform a translation by a vector calculated to cancel out the effect of global translational symmetry. In theory, given two copies of the same system that only differ by a translation of all atoms in the unit cell, this should produce two systems that overlap perfectly. Can be used to compare slightly different systems if they're similar enough. If a selection is given, only those atoms will be used to calculate the center, but the translation will still be applied to all atoms. The same atoms have to be used in all systems for comparisons to make sense (for example one might use all the heavy atoms and not include hydrogens).

Parameters:
selection (AtomSelection): selection object defining which atoms to
act on. By default, all of them.

Returns:
regularised (ase.Atoms): Atoms object translated by the regularizing
vector.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'regularised'

default_params = {u'selection': None}

static extract(*s*, *selection*, ***kwargs*)

class soprano.properties.transform.transform.**Rotate**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Returns an Atoms object with some or all the atoms rotated by a given quaternion and with a given center. Absolute or scaled coordinates may be used.

Parameters:
selection (AtomSelection): selection object defining which atoms to act on. By default, all of them.
center ([float]*3): center around which the rotation should take place. By default the origin of the axes.
quaternion (ase.quaternions.Quaternion): quaternion expressing the rotation that should be applied.
scaled (bool): if True, treat the input vector as expressed in scaled, not absolute, coordinates.

Returns:
rotated (ase.Atoms): Atoms object with the rotation performed.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the property (will be used to store it as array if requested)
params: named arguments specific to this type of property

default_name = u'rotated'

default_params = {u'scaled': False, u'quaternion': None, u'selection': None, u'center': [0, 0, 0]}

static extract(*s*, *selection*, ***kwargs*)

class soprano.properties.transform.transform.**Translate**(*name=None*, ***params*)
Bases: *soprano.properties.atomsproperty.AtomsProperty*

Returns an Atoms object with some or all the atoms translated by a given vector. Absolute or scaled coordinates may be used.

Parameters:
selection (AtomSelection): selection object defining which atoms to act on. By default, all of them.
vector ([float]*3): vector by which to translate the atoms.
scaled (bool): if True, treat the input vector as expressed in scaled, not absolute, coordinates.

Returns:
translated (ase.Atoms): Atoms object with the translation performed.

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'translated'

default_params = {u'vector': [0, 0, 0], u'scaled': False, u'selection': None}

static extract (*s*, *selection*, ***kwargs*)

Submodules

soprano.properties.atomsproperty module Definition of AtomsProperty class.

A generic template class that specific Properties will inherit from.

class soprano.properties.atomsproperty.**AtomsProperty** (*name=None*, ***params*)
Bases: object

Initialize an AtomsProperty and set its parameters. The AtomsProperty instance can then be called with a structure as its only argument to get the property with the given parameters.

Args:
name (str): a name to give to this specific instance of the
property (will be used to store it as array if
requested)
params: named arguments specific to this type of property

default_name = u'generic_property'

default_params = {}

static extract (*s*, ***params*)
Extract the given property with given parameters from an Atoms object.

Args:
s (ase.Atoms): the structure from which to extract the property
params: named arguments specific to this type of property

Returns:
property: the value of the property for the given structure and
parameters

classmethod `get`(*s*, *store_array=False*)

Extract the given property using the default parameters on an Atoms object s

Args:

s (ase.Atoms or AtomsCollection): the structure or collection from which to extract the property

store_array (bool): if s is a collection, whether to store the resulting data as an array in the collection using the default name for this property

Returns:

property: the value of the property for the given structure or a list of values if a collection has been passed

Submodules

soprano.selection module

selection.py

Contains the definition of an AtomSelection class, namely a group of selected atoms for a given structure, and methods to build it.

class `soprano.selection.`**`AtomSelection`**(*atoms*, *sel_indices*, *authenticate=True*)

Bases: `object`

AtomSelection object.

An AtomSelection represents a group of atoms from an ASE Atoms object. It keeps track of them and can be used to perform operations on them (for example geometrical transformation or extraction of specific properties). It does not keep track of the original Atoms object it's been created from, but can be "authenticated" to verify that it is indeed operating consistently on the same structure. It also provides a series of static methods to build selections with various criteria.

Initialize the AtomSelection.

Args:

atoms (ase.Atoms): the atoms object on which the selection is applied

sel_indices (list[int]): the list of indices of the atoms that are to be selected

authenticate (Optional[bool]): whether to use hashing to confirm the identity of the atoms object

we're operating with

static `all` (*atoms*)
Generate a selection for the given Atoms object of all atoms.

Args:
atoms (ase.Atoms): Atoms object on which to perform selection

Returns:
selection (AtomSelection)

static `from_box` (*atoms*, *abc0*, *abc1*, *periodic=False*, *scaled=False*)
Generate a selection for the given Atoms object of all atoms within a given box volume.

Args:
atoms (ase.Atoms): Atoms object on which to perform selection
abc0 ([float, float, float]): bottom corner of box
abc1 ([float, float, float]): top corner of box
periodic (Optional[bool]): if True, include periodic copies of the
atoms
scaled (Optional[bool]): if True, consider scaled (fractional)
coordinates instead of absolute ones

Returns:
selection (AtomSelection)

static `from_element` (*atoms*, *element*)
Generate a selection for the given Atoms object of all atoms of a specific element.

Args:
atoms (ase.Atoms): Atoms object on which to perform selection
element (str): symbol of the element to select

Returns:
selection (AtomSelection)

static `from_sphere` (*atoms*, *center*, *r*, *periodic=False*, *scaled=False*)
Generate a selection for the given Atoms object of all atoms within a given spherical volume.

Args:
atoms (ase.Atoms): Atoms object on which to perform selection

center ([float, float, float]): center of the sphere
r (float): radius of the sphere
periodic (Optional[bool]): if True, include periodic copies of the
atoms
scaled (Optional[bool]): if True, consider scaled (fractional)
coordinates instead of absolute ones

Returns:
selection (AtomSelection)

get_array(*name*)
Retrieve a previously stored data array.

Args:
name (str): name of the array to be set or created

Returns:
array (np.ndarray): array of data to be saved

indices

set_array(*name*, *array*)
Save an array of given name containing arbitraty information tied to the selected atoms. This must match the length of the selection and will be passed on to any Atoms objects created with .subset.

Args:
name (str): name of the array to be set or created
array (np.ndarray): array of data to be saved

subset(*atoms*)
Generate an Atoms object containing only the selected atoms.

validate(*atoms*)
Check that the given Atoms object validates with this selection.

soprano.utils module

utils.py

Contains package-wide useful routines that don't fall under any specific category. Many of these handle common operations involving periodicity, conversions between different representations etc.

soprano.utils.**abc2cart**(*abc*)
Transforms an axes and angles representation of lattice parameters into a Cartesian one

soprano.utils.**cart2abc**(*cart*)
Transforms a Cartesian representation of lattice parameters into an axes and angles one

soprano.utils.**hkl2d2_matgen**(*abc*)
Generate a matrix that turns hkl indices into inverse crystal plane distances for a given lattice in ABC form

soprano.utils.**inspect_args**(*f*)

soprano.utils.**inv_plane_dist**(*hkl*, *hkl2d2*)
Calculate inverse planar distance for a given set of Miller indices h, k, l.

soprano.utils.**is_string**(*s*)
Checks whether s is a string, with Python 2 and 3 compatibility

soprano.utils.**list_distance**(*l1*, *l2*)
Return an integer distance between two lists (number of differing elements)

soprano.utils.**minimum_periodic**(*v*, *latt_cart*)
Find the shortest periodic equivalent vector for a list of vectors and a given lattice.

Args:
v (np.ndarray): list of 3-vectors representing points or vectors to
reduce to their closest periodic version
latt_cart (np.ndarray): unit cell in cartesian form

Returns:
v_period (np.ndarray): array with the same shape as v, containing the
vectors in periodic reduced form
v_cells (np.ndarray): array of triples of ints, corresponding to the
cells from which the various periodic copies of
the vectors were taken. For an unchanged vector
will be all [0,0,0]

soprano.utils.**minimum_supcell**(*max_r, latt_cart=None, r_matrix=None, pbc=[True, True, True]*)
Generate the bounds for a supercell containing a sphere of given radius, knowing the unit cell.

Args:
max_r (float): radius of the sphere contained in the supercell
latt_cart (np.ndarray): unit cell in cartesian form
r_matrix (np.ndarray): matrix for the quadratic form returning
r^2 for this supercell.
Alternative to latt_cart, for a direct
space cell would be equal to
np.dot(latt_cart, latt_cart.T)
pbc ([bool, bool, bool]): periodic boundary conditions - if
a boundary is not periodic the
range returned will always be zero
in that dimension

Returns:
shape (tuple[int]): shape of the supercell to be built.

Raises:

ValueError: if some of the arguments are invalid

soprano.utils.**periodic_center**(*v_frac*)

Alright, how does this work? Basically, we're looking for the point, inside the unit cell, which minimizes the sum of the squared distance from all ions. Of course we need to consider the periodic boundaries. So the distance on a single axis isn't simply abs(x), but a triangular wave. Fun times! A triangular wave can be represented as a Fourier series. And we can truncate that series to the first term because the minimum basically stays the same and get: sum((x-x_i)**2) ~ sum((4/pi**2*sin(2*pi*(x-x_i-1/4))+0.5)**2) All the factors depend on the fact that we need to move the triangular wave to the interval [0,1] and center it so that it's 0 for x-x_i == 0. It gets better! We take the derivative of this thing and look for a spot where it becomes zero. The derivative is kind of a trigonometric monstrosity but we can solve the equation by setting t = e^(2*pi*1.0j*x) and then replacing cosines and sines with it. As a result, we get an equation of 4th degree in t. And then we solve that with numpy.roots, take the phase, turn that into a coordinate, find the one corresponding to the absolute minimum. All of which we can perform independently on each of the three axes because the function is just the sum of the three components: sum((r-r_i)**2) = sum((x-x_i)**2) + sum((y-y_i)**2) + sum((z-z_i)**2). And there you go! Problem solved.

soprano.utils.**progbar**(*i*, *i_max*, *bar_len=20*, *spinner=True*, *spin_rate=3.0*)

A textual progress bar for the command line

Args:

i (int): current progress index

max_i (int): final progress index

bar_len (Optional[int]): length in characters of the bar (no brackets)

spinner (Optional[bool]): show a spinner at the end

spin_rate (Optional[float]): spinner rotation speed (turns per full progress)

Returns:

bar (str): a progress bar formatted as requested

soprano.utils.**replace_folder**(*path*, *new_folder*)

Replace the folder of the given path with a new one

soprano.utils.**seedname**(*path*)

Get the filename (with no extension) from a full path

soprano.utils.**supcell_gridgen**(*latt_cart*, *shape*)

Generate a full linearized grid for a supercell with r_bounds and a base unit cell in Cartesian form.

Args:

latt_cart (np.ndarray): unit cell in cartesian form

shape (tuple[int]): shape of the supercell to be built, as returned by minimum_supcell.

Returns:

neigh_i_grid (np.ndarray): supercell grid in fractional coordinates

neigh_grid (np.ndarray): supercell grid in cartesian coordinates

Raises:

ValueError: if some of the arguments are invalid

soprano.utils.**swing_twist_decomp**(*quat*, *axis*)

Perform a Swing*Twist decomposition of a Quaternion. This splits the quaternion in two: one containing the rotation around axis (Twist), the other containing the rotation around a vector parallel to axis (Swing).

Returns two quaternions: Swing, Twist.

Soprano 0.5.0

A Python library to crack crystals by Simone Sturniolo

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

S

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

W

X