Hylight

Release 1.0.0

Théo Cavignac

Jul 10, 2024

CONTENTS

CHAPTER

ONE

ABOUT

Welcome to Hylight's documentation!

Hylight is a post-processing tool written in Python to simulate the luminescence of solids based on the results of *ab initio* computations.

You may be interested in reading our first paper using it on the luminescence of $BaZrO_3$:Ti [BaZrO3:Ti].

To learn how to use Hylight, you can read the *tutorials*. Finally there is a complete *reference* that goes over all individual functions implemented in the package.

CHAPTER

TWO

ATTRIBUTION

2.1 Licenses

Hylight is written and maintained by the PyDEF team. The source code for Hylight is distributed under the EUPL license. This is a non viral copyleft license, to be compared to MPL, but, unlike the MPL it is available in all official languages of the EU, including authors own language, french. This documentation is distributed under CC-BY license.

2.2 Citing Hylight

If you ever use Hylight for a published scientific work, we ask that you cite the related paper [Hylight].

CHAPTER

THREE

TUTORIALS

3.1 Al2O3:Ti study

This section present a short overview of the use of the main features of Hylight, applied a the system Al_2O_3 : Ti,Mg. The synthesis and luminescence characterization of the doped material showed a blue luminescence that we tried to simulate. The cell used was a 2x2x1 supercell of the conventional cell of Al_2O_3 in which one Al atom was substituted with a Ti atom and another Al was substituted with a Mg atom (to compensate the charge of a Ti4+). For practical reasons, the vibration modes used in this tutorial are computed with CRYSTAL 17. Other computations (positions and energies) comes from VASP 5.4.4.

Luminescence Properties of Al2O3:Ti in the Blue and Red Regions: A Combined Theoretical and Experimental Study

3.1.1 Preliminaries

Here are some utility functions and import that will be used throughout the study.

```
%matplotlib inline
```

```
import numpy as np
import matplotlib.pyplot as plt
from hylight.constants import *
import logging

logging.getLogger().setLevel(logging.INFO)  # setting the logging level to INFO
```

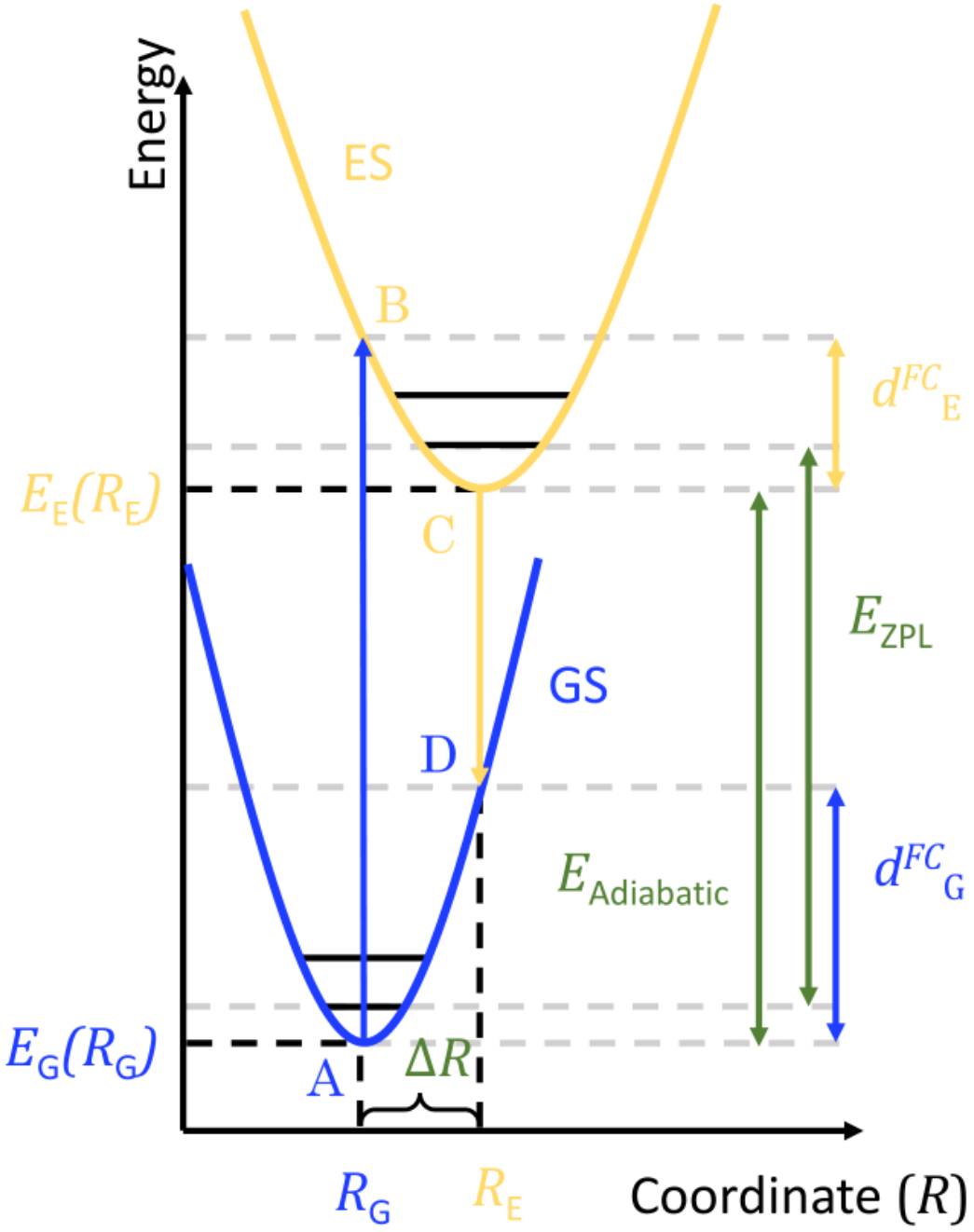
```
# Helper functions to deal with experimental spectrum


def best_max(raw_x, raw_y, f=0.95):
    "Fit a 2nd order polynom to get a position of the maximum without the noise."
    guess = np.max(raw_y)

    y = raw_y[raw_y > f * guess]
    x = raw_x[raw_y > f * guess]

    x1, x2, x3, x4 = np.mean(x**1), np.mean(x**2), np.mean(x**3), np.mean(x**4)

    x2y = np.mean(x**2 * y)
    xy = np.mean(x * y)
    my = np.mean(y)
```

(continues on next page)

(continued from previous page)

```
    A = np.array([[x4, x3, x2], [x3, x2, x1], [x2, x1, 1.0]])

    B = np.array([[x2y, xy, my]]).transpose()

    alpha, beta, gamma = np.linalg.solve(A, B)[:, 0]

    return -0.5 * beta / alpha, gamma - 0.25 * beta**2 / alpha


def load_exp(path, sep=" ", skip=0):
    exp = np.loadtxt(path, delimiter=sep, skiprows=skip)
    x, y = eV1_in_nm / exp[:, 0], exp[:, 1]
    xmax, ymax = best_max(x, y)
    y /= ymax
    return x, y, xmax


def select_interval(x, y, emin, emax, norm=True, npoints=None):
    slice_ = (x > emin) * (x < emax)
    xs, ys = x[slice_], y[slice_] / (np.max(y[slice_]) if norm else 1.0)

    if npoints is not None:
        emin = max(np.min(xs), emin)
        emax = min(np.max(xs), emax)
        xint = np.linspace(emin, emax, npoints)
        return xint, interp1d(xs, ys)(xint)

    return xs, ys
```

To perform the simulation of the spectrum of Al_2O_3 : Ti,Mg we need a few pieces of information. The following diagram shows the four points A, B, C and D that need to be computed in DFT. We used constrained DFT to compute B and C with an explicit hole in the VBM and an electron in the CBM. The cDFT and the regular DFT computations provides us with total energies and the positions R_G and R_E that we will use as inputs in the next section.



Given the geometry R_G we also perform a DFPT computation to get the eigen values and eigenvectors of the dynamical matrix, that is we get the Γ vibrational modes of the crystal.

The modes are extracted from the CRYSTAL output file and stored in the `vib_ti_mg_edge_crys.log.npz` with the command line tool `hylight-modes`:

```
$ hylight-modes convert --from crystal vib_ti_mg_edge_crys.log vib_ti_mg_edge_crys.log.
↪npz
Loaded 360 modes from vib_ti_mg_edge_crys.log.
Wrote vib_ti_mg_edge_crys.log.npz.
```

3.1.2 Electronic and vibrational parameters

Here we declare the inputs that will be used later.

The energy differences are computed with VASP from DFT ground state and cDFT excited state computations.

```
fc_shift_gs = 0.5846914  # D-A
fc_shift_es = 0.5312436  # B-C (only used for the width approximation)
e_adia = 3.49839858  # C-A

e_vert = e_adia - fc_shift_gs  # C-D

outcar = "vib_ti_mg_edge_crys.log.npz"  # vibrations
poscar_gs = "POSCAR_GS"  # R_G
poscar_es = "POSCAR_S1"  # R_E
T = 300  # measure temperature (K)

print(f"Adiabatic energy difference: {e_adia:0.3f} eV")
print(f"Vertical energy difference: {e_vert:0.3f} eV")
```

```
Adiabatic energy difference: 3.498 eV
Vertical energy difference: 2.914 eV
```

3.1.3 Vibrational properties investigation

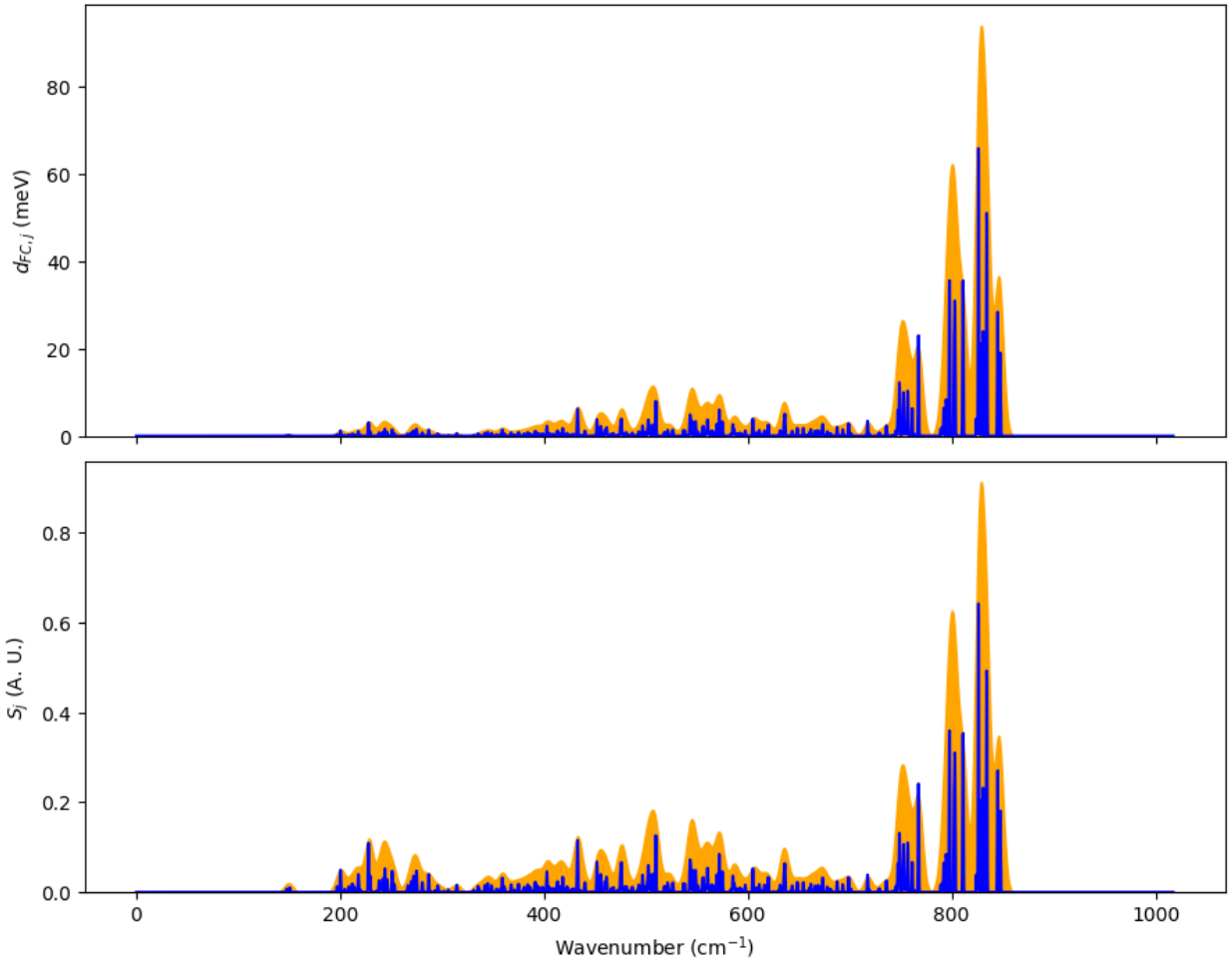
We now plot the spectral function to identify important modes.

```
from hylight.multi_modes import plot_spectral_function

fig, (ax_fc, ax_s) = plot_spectral_function(
    outcar,
    poscar_gs,
    poscar_es,
    use_cm1=True,
    disp=5e-1,
    mpl_params={
        "S_stack": {"color": "orange"},
        "FC_stack": {"color": "orange"},
        "S_peaks": {"color": "blue", "lw": 1.5},
        "FC_peaks": {"color": "blue", "lw": 1.5},
    },
)

ax_fc.set_ylabel("$d_{FC,j}$ (meV)")
ax_s.set_ylabel("$S_j$ (A. U.)")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
fig.set_size_inches((9, 7))
plt.savefig("spectral_func.png")
```



There are a few dominant modes around 830 cm^{-1}. We can export them to Jmol files to visualize it.

```
from hylight.loader import load_phonons
from hylight.multi_modes import compute_delta_R

delta_R = (
    compute_delta_R(poscar_gs, poscar_es) * 1e-10
)  # careful, Mode.huang_rhys expects SI units
modes, _, _ = load_phonons(outcar)

big_modes = [m for m in modes if m.huang_rhys(delta_R) > 0.4]
print("Important modes:", len(big_modes))

from hylight.jmol import export

mode = big_modes[0]
export(
```

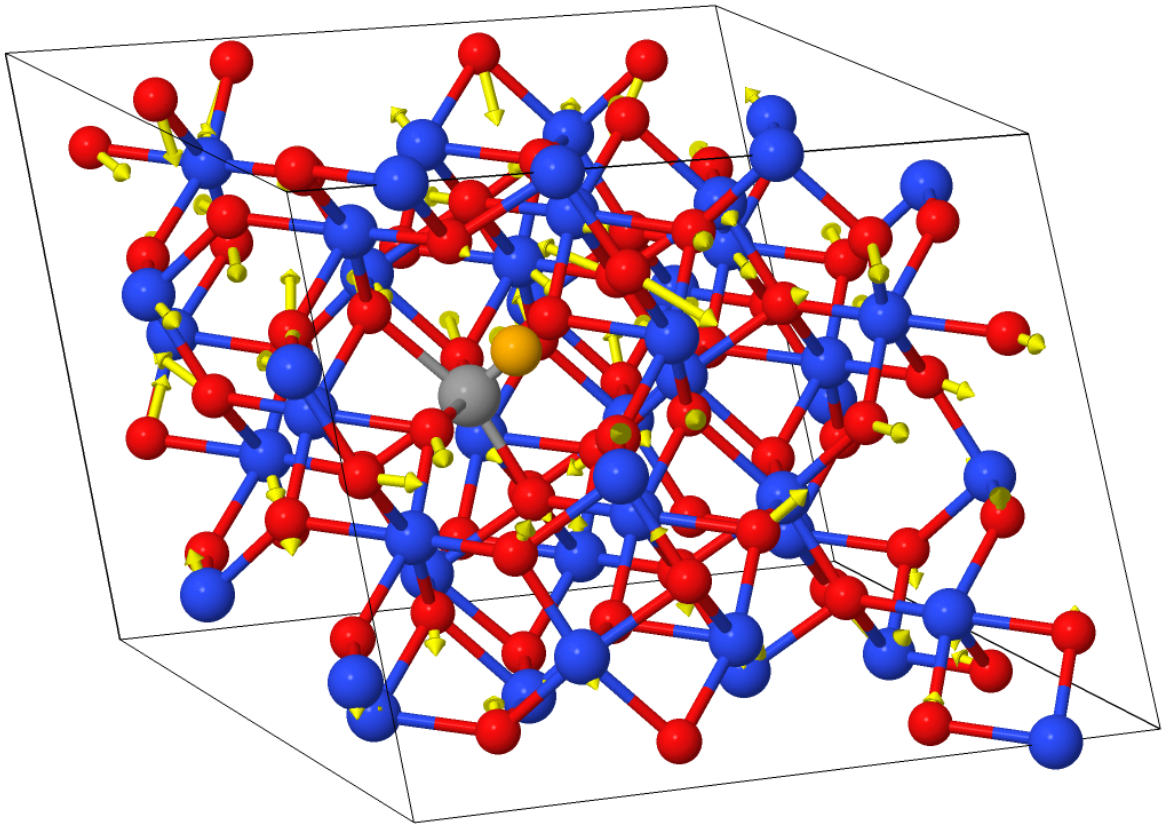
(continues on next page)

(continued from previous page)

```
    "big_mode.jmol",
    mode,
    scale=20.0,
    offset=np.array([0.5, 0.5, 0.5]),  # move the origin of half b
    bonds=[  # bonds to show, with length between 0 and 2.2 A
        ("Al", "O", 0, 2.2),
        ("Ti", "O", 0, 2.2),
    ],
    atom_colors=[  # colors to show atoms
        ("Al", "#3050FF"),  # blue
        ("Mg", "#FFB000"),  # orange
        ("Ti", "#A0A0A0"),  # grey
        ("O", "#FF1010"),  # red
    ],
)
```

```
Important modes: 2
```

Here is the result in Jmol:



This mode is mostly composed of bulk movement of the oxygen sublattice.

3.1.4 Line width approximation

We now will use a semi-classical model to approximate the width of the band. The specific model for the width is a 1D reduction of the system, and the effective vibration frequency is computed as a mean over FC shifts.

```
from hylight.guess_width import guess_width, OmegaEff, WidthModel

width = guess_width(
    outcar,
    (poscar_gs, poscar_es),
    fc_shift_gs,
    fc_shift_es,
    T,
    omega_eff_type=OmegaEff.FC_MEAN,
    width_model=WidthModel.ONED,
)
print(f"Guessed line FWHM = {width*1e3:0.1f} meV")
```

```
S = 8.055327600850317
d_fc^g,v = 0.6477615566764704 eV
d_fc^e,v = 0.5885483886207532 eV
Effective phonon energy (GS) = 0.088054437806992 eV / 710.2069370108776 cm1
Effective phonon energy (ES) = 0.08393338746477763 eV / 676.9684244077196 cm1
ex_fwhm = 0.6267042207345681 eV
```

```
Using a Gaussian line shape.
Guessed line FWHM = 561.6 meV
```

3.1.5 Spectrum simulation

Finally, taking vibrational and electronic parameters into account we simulate the spectrum.

```
from hylight.multi_modes import compute_spectrum

e, sp = compute_spectrum(  # simulate the spectrum
    outcar,
    (poscar_gs, poscar_es),
    e_adia,
    width,
)

_, max_th = max(zip(sp, e))  # extract the maximum of emission energy
print(f"Computed maximum of emission: {max_th:.3} eV")
```

```
Mode 359 has a reference position somewhat far from GS position. (atom 47 moved by 0.
↪02393033937295447)
Total Huang-Rhys factor 8.055327600850317.
```

```
Using a Gaussian line shape.
Computed maximum of emission: 2.98 eV
```

3.1.6 Plotting

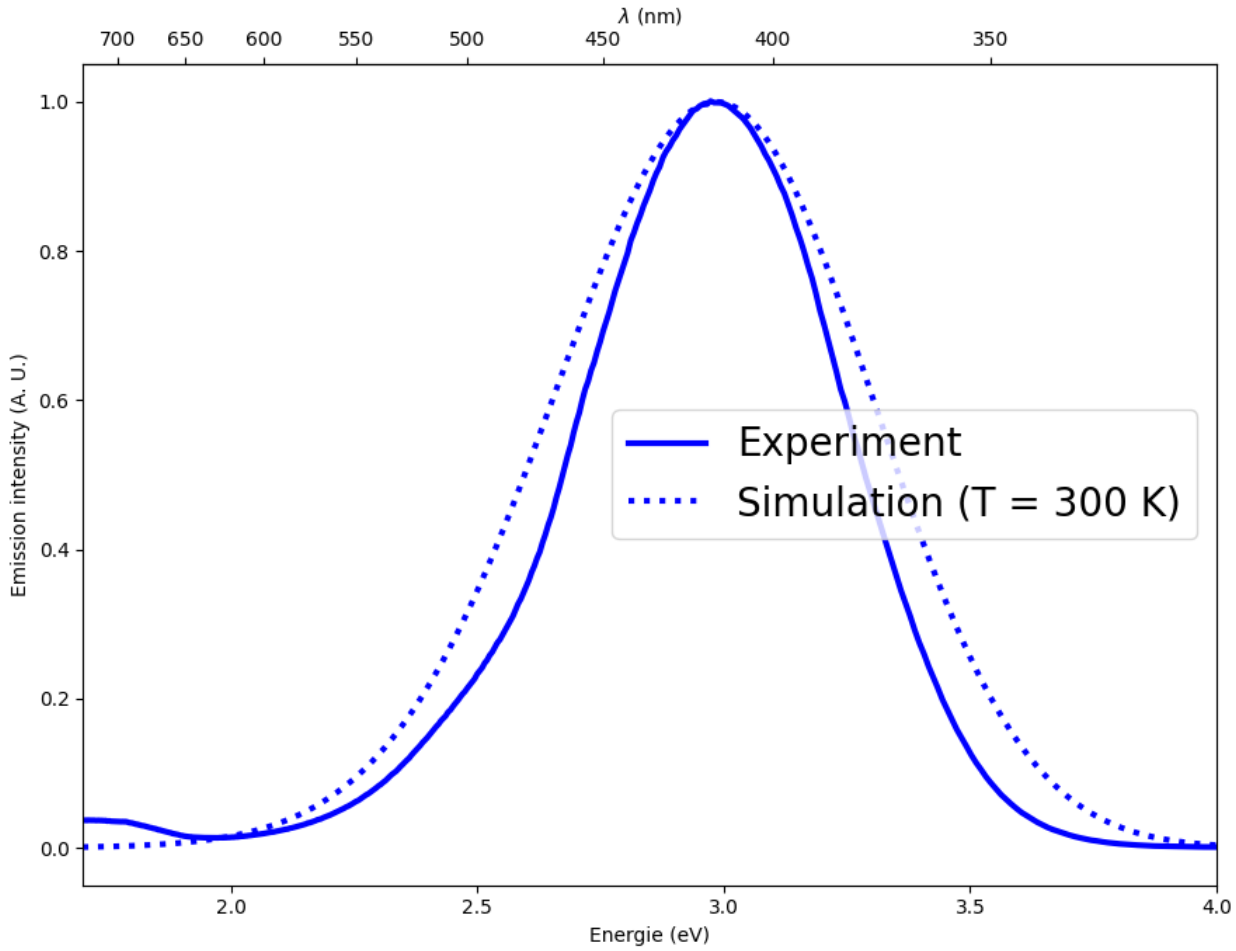
Here we load the experimental data that will be used for comparision with the simulation.

```
exp_e, exp_i, max_emission = load_exp("recorded_emission.txt")

print(f"Measured maximum of emission: {max_emission:.3} eV")
```

```
Measured maximum of emission: 2.98 eV
```

Finally we setup a plot to show the result of the simulation against the measurement.

```
plt.figure(figsize=(9, 7))

e, sp = e[1:], sp[1:]
exp_e, exp_i = exp_e[1:], exp_i[1:]
plt.plot(exp_e, exp_i, "b", lw=3, label="Experiment")
plt.plot(e, sp, ":", color="b", lw=3, label=f"Simulation (T = {T} K)")

plt.xlim(1.7, 4)
plt.xlabel("Energie (eV)")
plt.ylabel("Emission intensity (A. U.)")

ax = plt.gca()
np.seterr(
    divide="ignore"
)  # there is a division by zero occuring in the next line but it is irrelevant for our␣
↪xlim
secax = ax.secondary_xaxis(
    "top", functions=(lambda x: eV1_in_nm / x, lambda x: eV1_in_nm / x)
)
secax.set_xlabel("$\\lambda$ (nm)")

plt.tight_layout()
leg = plt.legend(prop={"size": 20})
plt.savefig("al2o3_ti_spectra.png")
```



We can see that, while the width is overestimated by our rough approximation, the positioning of the band is very well reproduced.

CHAPTER

FOUR

API REFERENCE

4.1 hylight package

4.1.1 Subpackages

hylight.crystal package

Submodules

hylight.crystal.common module

Common utilities to read CRYSTAL files.

class hylight.crystal.common.**CrystalOut**(*lattice*, *species*, *species_names=None*)

Bases: *Struct*

Struct from a CRYSTAL output file.

classmethod from_file(*filename*)

Read a structure from a CRYSTAL log file.

Parameters
filename – path to the file to read

Returns
a Poscar object.

hylight.crystal.loader module

Read vibrational modes from CRYSTAL log.

hylight.crystal.loader.**load_phonons**(*path: str*) → tuple[list[*Mode*], list[int], list[float]]

Load phonons from a CRYSTAL17 logfile.

Returns

`(phonons, pops, masses)`

- *phonons*: list of hylight.mode.Mode instances
- *pops*: population for each atom species
- *masses*: list of SI masses

hylight.crystal.loader.**normalize**(*name*)

Normalize an atom name (e.g. ZR -> Zr).

Module contents

CRYSTAL related utilities.

hylight.findiff package

Submodules

hylight.findiff.collect module

Grab forces from a collection of single point computations and compute hessian.

hylight.findiff.collect.**dropwhile_err**(*pred*, *it*, *else_err*)

itertools.dropwhile wrapper that raise else_err if it reach the end of the file.

hylight.findiff.collect.**get_forces**(*n*, *path*)

Extract n forces from an OUTCAR.

hylight.findiff.collect.**get_forces_and_pos**(*n*, *path*)

Extract n forces and atomic positions from an OUTCAR.

hylight.findiff.collect.**get_ref_info**(*path*)

Load system infos from a OUTCAR.

This is an ad hoc parser, so it may fail if the OUTCAR changes a lot.

Returns

`(atoms, ref, pops, masses)`

- *atoms*: list of species names
- *pos*: positions of atoms
- *pops*: population for each atom species
- *masses*: list of SI masses

hylight.findiff.collect.**process_phonons**(*outputs*, *ref_output*, *basis_source=None*, *amplitude=0.01*, *nproc=1*, *symm=True*, *asr_force=False*)

Process a set of OUTCAR files to compute some phonons using Force based finite differences.

Parameters

- **outputs** – list of OUTCAR paths corresponding to the finite displacements.
- **ref_output** – path to non displaced OUTCAR.
- **basis_source** – (optional) read a displacement basis from a path. The file is a npy file from numpy's save. If None, the basis is built from the displacements. If not None, the order of outputs *must* match the order of the displacements in the array.
- **amplitude** – (optional) amplitude of the displacement, only used if basis_source *is not* None.
- **nproc** – (optional) number of parallel processes used to load the files.

- **symm** – (optional) If True, use symmetric differences. OUTCARs *must* be ordered as `[+delta_1, -delta_1, +delta_2, -delta_2, ...]`.
- **asr_force** – (optional, `False`) enforce the accoustic sum rule by projecting the dynamic matrix on the subspace where homogeneous translation of all modes leads to a null frequency.

Returns
the same tuple as the load_phonons functions.

Note: When using non canonical basis (displacements are not along a single degree of freedom of a single atom at a time) it may be important to provide the basis excplicity because it will avoid important rounding errors found in the OUTCAR.

---

hylight.findiff.gen module

Generate a collection of positions for finite differences computations.

hylight.findiff.gen.**gen_disp**(*ref*, *basis*, *amplitude=0.01*, *symm=False*)

Iterate over displaced Poscar instances from a given set of directions.

Takes a reference position and displaces it into directions from basis to produce a set of positions for finite differences computations.

Parameters

- **ref** – reference Poscar instancce
- **basis** – nupy array where each row is a direction.
- **amplitude** – (optional, 0.01) amplitude of the displacements in Angstrom.
- **symm** – (optional, False) when True, generate both ref+delta and ref-delta POSCARs for each direction.

hylight.findiff.gen.**random_basis**(*n*, *seed=0*)

Generate a random basis matrix or rank n.

Parameters

- **n** – Rank of the basis
- **seed** – (optional, 0) randomness seed.

Returns
a (n, n) orthonormal numpy array.

hylight.findiff.gen.**save_disp**(*ref*, *basis*, *disp_dest='.'*, *amplitude=0.01*, *symm=False*)

Produce displaced POSCARs from a given set of directions.

Takes a reference position and displaces it into directions from basis to produce a set of positions for finite differences computations.

Parameters

- **ref** – reference Poscar instancce
- **basis** – nupy array where each row is a direction.
- **disp_dest** – (optional, ".") directory name or callable that determine the destination of the POSCARs.

- **amplitude** – (optional, 0.01) amplitude of the displacements in Angstrom.
- **symm** – (optional, False) when True, generate both ref+delta and ref-delta POSCARs for each direction.

hylight.findiff.gen.**save_disp_from_basis**(*source*, *basis_source*, *disp_dest='.'*, *amplitude=0.01*, *symm=False*)

Produce displaced POSCARs from a given set of directions.

Takes a reference position and displaces it into directions from basis_source to produce a set of positions for finite differences computations.

Parameters

- **source** – reference POSCAR
- **basis_source** – Name of the basis set file. It is a npy file made with numpy's save function. Each row is a direction.
- **disp_dest** – (optional, ".") directory name or callable that determine the destination of the POSCARs.
- **amplitude** – (optional, 0.01) amplitude of the displacements in Angstrom.
- **symm** – (optional, False) when True, generate both ref+delta and ref-delta POSCARs for each direction.

hylight.findiff.gen.**save_random_disp**(*source*, *disp_dest='.'*, *basis_dest='basis.npy'*, *amplitude=0.01*, *seed=0*, *symm=False*)

Produce displaced POSCARs in random directions.

Takes a reference position and displaces it into random directions to produce a set of positions for finite differences computations. It also save the set of displacements in basis_dest.

Parameters

- **source** – reference POSCAR
- **disp_dest** – (optional, ".") directory name or callable that determine the destination of the POSCARs.
- **basis_dest** – (optional, "basis.npy") Name of the basis set file. It is a npy file made with numpy's save function.
- **amplitude** – (optional, 0.01) amplitude of the displacements in Angstrom.
- **seed** – (optional, 0) random seed to use to generate the displacements.
- **symm** – (optional, False) when True, generate both ref+delta and ref-delta POSCARs for each direction.

Module contents

A module for finite differences computations.

hylight.jmol package

Module contents

Write Jmol files.

hylight.jmol.**export**(*dest*, *mode*, *, *displacement=True*, *scale=1.0*, *compression=8*, *offset=None*, *recenter=True*, ***opts*)

Export a mode to JMol zip format.

Parameters

- **dest** – path to the JMol zip file.
- **mode** – the mode to export.
- **displacement** – (kw only, default True) choose between eigendisplacements and eigenvectors
- **scale** – (kw only, default 1.0) a scale factor for the displacements/eigenvectors
- **compression** – (kw only) zipfile compression algorithm.
- **offset** – (kw only, *np.array([0, 0, 0])*) offset vector (in fractional coordinates) to shift the atoms in the unit cell to have different atoms at the center
- **recenter** – (kw only, *True*) wether atoms should be moved according to periodic conditions to fit in the unit cell
- ****opts** – see *write_jmol_options()*

hylight.jmol.**export_disp**(*dest*, *struct*, *disp*, *, *compression=8*, ***opts*)

Export a difference between two positions to JMol zip format.

Parameters

- **dest** – path to the JMol zip file.
- **struct** – the reference position (a *hylight.struct.Struct* instance).
- **disp** – an array of displacements.
- **compression** – (optional) zipfile compression algorithm.
- ****opts** – see *write_jmol_options()*

hylight.jmol.**format_v**(*v*)

Format a vector in Jmol syntax.

hylight.jmol.**write_jmol_options**(*f*, *opts*)

Write options in the form of a JMol script.

Parameters

- **f** – a file like object.
- **opts** – a dictionary of options
  - *unitcell*: lattice vectors as a 3x3 matrix where vectors are in rows.
  - ***bonds*: a list of `(sp_a, sp_b, min_dist, max_dist)` where species** names are strings of names of species and **_dist* are interatomic distances in Angstrom.

- ***atom_colors*: a list of (sp, color) where sp is the name of a** species and color is the name of a color or an HTML hex code (example `"#FF0000"` for pure red).
- *origin*: the origin of the unitcell box (in fractional coordinates)

`hylight.jmol.`**`write_xyz`**(*f*, *atoms*, *ref*, *delta*)

Write the coordinates and displacements of in the JMol xyz format.

Parameters

- **`f`** – a file-like object to write to.
- **`atoms`** – the list of atom names
- **`ref`** – an array of atomic positions
- **`delta`** – an array of displacements

hylight.phonopy package

Submodules

hylight.phonopy.loader module

Module to load phonons frequencies and eigenvectors from phonopy output files.

It always uses PyYAML, but it may also need h5py to read hdf5 files.

`class` `hylight.phonopy.loader.`**`PPStruct`**(*pops: list[int]*, *lattice: ndarray*, *masses: list[float]*, *atoms: list[str]*, *ref: ndarray*)

Bases: `object`

Crystal structure as described by phonopy.

`atoms: list[str]`

`classmethod from_yaml_cell`(*cell: dict*) → *PPStruct*

Create a PPStruct from a cell dictionary found in phonopy files.

`lattice: ndarray`

`masses: list[float]`

`pops: list[int]`

`ref: ndarray`

`hylight.phonopy.loader.`**`get_struct`**(*phyaml: str*) → *PPStruct*

Get a structure from the phonopy.yaml file.

`hylight.phonopy.loader.`**`load_phonons`**(*dir_: str*) → tuple[list[*Mode*], list[int], list[float]]

Load vibrational modes from phonopy output files.

This function takes the directory where the files are stored and tries to detect the right file to process.

See also:

- *load_phonons_qpointsh5()*
- *load_phonons_bandsh5()*

- *load_phonons_qpointsyaml()*
- *load_phonons_bandyaml()*

Parameters
dir – directory containing phonopy output files

Returns
(modes, frequencies, eigenvectors) tuple

hylight.phonopy.loader.**load_phonons_bandsh5**(*bandh5: str*, *phyaml: str*, *op=<built-in function open>*) → tuple[list[*Mode*], list[int], list[float]]

Load vibrational modes from phonopy HDF5 output files.

Parameters

- **bandh5** – path to band.hdf5 or band.hdf5.gz file
- **op** – (optional, open) open function, pass `gzip.open()` when dealing with compressed file.

Returns
(modes, frequencies, eigenvectors) tuple

hylight.phonopy.loader.**load_phonons_bandyaml**(*bandy: str*) → tuple[list[*Mode*], list[int], list[float]]

Load vibrational modes from phonopy YAML output files.

Parameters
bandy – path to band.yaml file

Returns
(modes, frequencies, eigenvectors) tuple

hylight.phonopy.loader.**load_phonons_qpointsh5**(*qph5: str*, *phyaml: str*, *op=<built-in function open>*) → tuple[list[*Mode*], list[int], list[float]]

Load vibrational modes from phonopy HDF5 output files.

Parameters

- **qph5** – path to qpoints.hdf5 or qpoints.hdf5.gz file
- **phyaml** – path to phonopy.yaml file
- **op** – (optional, open) open function, pass `gzip.open()` when dealing with compressed file.

Returns
(modes, frequencies, eigenvectors) tuple

hylight.phonopy.loader.**load_phonons_qpointsyaml**(*qpyaml: str*, *phyaml: str*) → tuple[list[*Mode*], list[int], list[float]]

Load vibrational modes from phonopy YAML output files.

Parameters

- **qpyaml** – path to qpoints.yaml file
- **phyaml** – path to phonopy.yaml file

Returns
(modes, frequencies, eigenvectors) tuple

Module contents

phonopy related utils.

hylight.vasp package

Submodules

hylight.vasp.common module

Common utilities to read and write VASP files.

class hylight.vasp.common.**Poscar**(*lattice*, *species*, *species_names=None*)

Bases: *Struct*

A crystal cell from a VASP POSCAR file.

classmethod from_file(*filename*)

Load a POSCAR file

Parameters
filename – path to the file to read

Returns
a Poscar object.

to_file(*path='POSCAR'*, *cartesian=True*)

Write to a POSCAR file.

The property system_name may be set to change the comment at the top of the file.

Parameters

- **path** – path to the file to write
- **cartesian** –
  - if True, write the file in cartesian representation,
  - if False, write in fractional representation

to_stream(*out*, *cartesian=True*)

Write a POSCAR content to a stream.

The property system_name may be set to change the comment at the top of the file.

Parameters

- **path** – path to the file to write
- **cartesian** –
  - if True, write the file in cartesian representation,
  - if False, write in fractional representation

hylight.vasp.loader module

Read vibrational modes from VASP files.

hylight.vasp.loader.**load_phonons**(*path: str*) → tuple[list[*Mode*], list[int], list[float]]

Load phonons from a OUTCAR.

Note: This function is a bit heavy because of text parsing. You may want to use hyligh-modes to parse the file once and later load that preparsed file using *hylight.npz.load_phonons()* instead.

Returns

`(phonons, pops, masses)`

- *phonons*: list of *hylight.mode.Mode* instances
- *pops*: population for each atom species
- *masses*: list of SI masses

hylight.vasp.loader.**load_poscar**(*path*)

Read the positions from a POSCAR.

Returns

a `numpy.ndarray((natoms, 3), dtype=float)`

hylight.vasp.loader.**load_poscar_latt**(*path*)

Read the positions and the lattice parameters from a POSCAR.

Returns

a `(np.ndarray((natoms, 3), dtype=float), nd.array((3, 3), dtype=float))`

- first element is the set of positions
- second element is the lattice parameters

hylight.vasp.utils module

Pervasive utilities for hylight.vasp submodule.

hylight.vasp.utils.**make_finite_diff_poscar**(*outcar*, *poscar_gs*, *poscar_es*, *A=0.01*, ***, *load_phonons=<function load_phonons>*, *bias=0*, *mask=None*)

Compute positions for evaluation of the curvature of the ES PES.

Parameters

- **outcar** – the name of the file where the phonons will be read.
- **poscar_gs** – the path to the ground state POSCAR
- **poscar_es** – the path to the excited state POSCAR, it will be used as a base for the generated Poscars
- **A** – (optional, 0.01) the amplitude of the displacement in A.
- **load_phonons** – (optional, vasp.loader.load_phonons) the procedure use to read outcar
- **bias** – an energy under which modes are ignored, 0 by default

- **mask** – a `mode.Mask` to select the modes to consider, override the bias.

Returns

`(mu, pes_left, pes_right)`

- mu: the effective mass in kg
- pes_left: a Poscar instance representing the left displacement
- pes_right: a Poscar instance representing the right displacement

Module contents

VASP related utilities.

4.1.2 Submodules

4.1.3 hylight.constants module

A collection of useful physics constants.

4.1.4 hylight.guess_width module

Semi classical guess of linewidth.

class hylight.guess_width.**OmegaEff**(*wm*)

Bases: `object`

Mode of computation of a effective frequency.

FC_MEAN:

$$\Omega = \frac{\sum_j \omega_j d_j^{\mathrm{FC}}}{\sum_j d_j^{\mathrm{FC}}}$$

Should be used with *WidthModel.ONED* because it is associated with the idea that all the directions are softened equally in the excited state.

HR_MEAN:

$$\Omega = \frac{d^{\mathrm{FC}}}{S_{\mathrm{tot}}} = \frac{\sum_j \omega_j S_j}{\sum_j S_j}$$

HR_RMS:

$$\Omega = \sqrt{\frac{\sum_j \omega_j^2 S_j}{\sum_j S_j}}$$

FC_RMS:

$$\Omega = \sqrt{\frac{\sum_j \omega_j^2 d_j^{\mathrm{FC}}}{\sum_j d_j^{\mathrm{FC}}}}$$

Should be used with *WidthModel.SINGLE_ES_FREQ* because it makes sense when we get only one Omega_eff for the excited state (this single effective frequency should be computed beforehand)

GRAD:

The effective frequency of GS is computed in the direction of the gradiant of GS PES at the ES position. The effective frequency for the ES is provided by the user (and should be computed in the same direction). Should be used with *WidthModel.ONED*.

DISP:

The effective frequency of GS is computed in the direction of the displacement. The effective frequency for the ES is provided by the user (and should be computed in the same direction). Should be used with *WidthModel.ONED*.

DISP: *OmegaEff* = <hylight.guess_width.OmegaEff object>

FC_MEAN: *OmegaEff* = <hylight.guess_width.OmegaEff object>

FC_RMS: *OmegaEff* = <hylight.guess_width.OmegaEff object>

GRAD: *OmegaEff* = <hylight.guess_width.OmegaEff object>

HR_MEAN: *OmegaEff* = <hylight.guess_width.OmegaEff object>

HR_RMS: *OmegaEff* = <hylight.guess_width.OmegaEff object>

class hylight.guess_width.**WidthModel**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Mode of approximation of the band width.

ONED: The witdth is computed in a 1D mode for both ES and GS.

SINGLE_ES_FREQ: We suppose all modes of the ES have the same frequency (that should be provided).

FULL_ND: (Not implemented) We know the frequency of each ES mode.

FULL_ND = 2

ONED = 0

SINGLE_ES_FREQ = 1

hylight.guess_width.**duschinsky**(*phonons_a*, *phonons_b*)

Dushinsky matrix from b to a $S_{a \leftarrow b}$.

hylight.guess_width.**effective_phonon_energy**(*omega_eff_type*, *hrs*, *es*, *masses*, ***, *delta_R=None*, *modes=None*, *mask=None*)

Compute an effective phonon energy in eV following the strategy of omega_eff_type.

Parameters

- **omega_eff_type** – The mode of evaluation of the effective phonon energy.
- **hrs** – The array of Huang-Rhyes factor for each mode.
- **es** – The array of phonon energy in eV.
- **masses** – The array of atomic masses in atomic mass unit.
- **delta_R** – The displacement between GS and ES in A. It is only required if omega_eff_type is ONED_FREQ.

Returns

The effective energy in eV.

hylight.guess_width.**expected_width**(*phonons*, *delta_R*, *fc_shift_gs*, *fc_shift_es*, *T*, *mask=None*, *omega_eff_type=<hylight.guess_width.OmegaEff object>*, *width_model=WidthModel.ONED*)

Compute a spectrum without free parameters.

Parameters

- **phonons** – list of modes
- **delta_R** – displacement in A
- **fc_shift_gs** – Ground state/absorption Franck-Condon shift in eV
- **fc_shift_es** – Exceited state/emmission Franck-Condon shift in eV
- **T** – temperature in K
- **resolution_e** – energy resolution in eV
- **bias** – ignore low energy vibrations under bias in eV
- **window_fn** – windowing function in the form provided by numpy (see numpy.hamming)
- **pre_convolve** – (float, optional, None) if not None, standard deviation of the pre convolution gaussian
- **shape** – ZPL line shape.
- **omega_eff_type** – mode of evaluation of effective frequency.
- **result_store** – a dictionary to store some intermediate results.
- **ex_pes** – mode of evaluation of the ES PES curvature.

Returns

`(sig(T=0), sig(T))`

hylight.guess_width.**gradiant_at**(*modes*, *masses*, *delta_R*, ***, *mask=None*)

Compute the gradient of the PES at the position of delta_R.

Parameters

- **modes** – list of modes
- **masses** – list of atomic masses
- **delta_R** – displacement of each atoms in A
- **mask** – (optional, None) a *hylight.mode.Mask* used to discard irrelevant modes

Returns

the gradient of the PES at the position of delta_R

hylight.guess_width.**guess_width**(*phonons*, *delta_R*, *fc_shift_gs*, *fc_shift_es*, *T*, *resolution_e=0.001*, *mask=None*, *shape=LineShape.GAUSSIAN*, *omega_eff_type=<hylight.guess_width.OmegaEff object>*, *width_model=WidthModel.ONED*, *window_fn=<function hamming>*, *trial_line_sig=0.02*)

Try to guess the width of the line from a 1D semi-classical model.

Parameters

- **phonons** – list of modes (see `load_phonons()`) or path to load the modes
- **delta_R** – displacement in A in a numpy array (see `compute_delta_R()`) or tuple of two paths `(pos_gs, pos_es)`

- **fc_shift_gs** – Ground state/absorption Franck-Condon shift in eV
- **fc_shift_es** – Exceited state/emmission Franck-Condon shift in eV
- **T** – temperature in K
- **resolution_e** – energy resolution in eV
- **mask** – a mask used in other computations to show on the plot.
- **shape** – the lineshape (a `LineShape` instance)
- **omega_eff_type** – mode of evaluation of effective frequency.
- **ex_pes** – mode of evaluation of the ES PES curvature.
- **window_fn** – windowing function in the form provided by numpy (see numpy.hamming)

Returns
a guess width in eV

hylight.guess_width.**integrate**(*x*, *y*)

Integrate a function over x.

Parameters

- **x** – `numpy.ndarray` of x values
- **y** – `y = f(x)`

Returns
$\int f(x)dx$

hylight.guess_width.**sigma**(*T*, *S_em*, *e_phonon_g*, *e_phonon_e*)

Temperature dependant standard deviation of the lineshape.

Parameters

- **T** – temperature in K
- **S_em** – emmission Huang-Rhys factor
- **e_phonon_g** – energy of the GS PES vibration (eV)
- **e_phonon_e** – energy of the ES PES vibration (eV)

hylight.guess_width.**sigma_full_nd**(*T*, *delta_R*, *modes_gs*, *modes_es*, *mask=None*)

Compute the width of the ZPL for the `ExPES.FULL_ND` mode.

Parameters

- **T** – temperature in K.
- **delta_R** – distorsion in A.
- **modes_gs** – list of Modes of the ground state.
- **modes_es** – list of Modes of the excited state.

Returns
`numpy.ndarray` with only the width for the modes that are real in ground state.

hylight.guess_width.**sigma_hybrid**(*T*, *S*, *e_phonon*, *e_phonon_e*)

Compute the width of the ZPL for the `ExPES.SINGLE_ES_FREQ` mode.

hylight.guess_width.**variance**(*x*, *y*)

Compute the variance of a random variable of distribution y.

4.1.5 hylight.loader module

Wrapper for the npz loader.

4.1.6 hylight.mode module

Vibrational mode and related utilities.

class hylight.mode.**CellMismatch**(*reason*, *details*)

Bases: object

A falsy value explaining how the cell are not matching.

class hylight.mode.**Mask**(*intervals: list[tuple[float, float]]*)

Bases: object

An energy based mask for the set of modes.

accept(*value: float*) → bool

Return True if value is not under the mask.

add_interval(*interval: tuple[float, float]*) → None

Add a new interval to the mask.

as_bool(*ener: ndarray*) → ndarray

Convert to a boolean *np.ndarray* based on *ener*.

classmethod from_bias(*bias: float*) → *Mask*

Create a mask that reject modes of energy between 0 and *bias*.

Parameters
bias – minimum of accepted energy (eV)

Returns
a fresh instance of *Mask*.

plot(*ax*, *unit*)

Add a graphical representation of the mask to a plot.

Parameters

- **ax** – a matplotlib *Axes* object.
- **unit** – the unit of energy to use (ex: `hylight.constant.eV_in_J` if the plot uses eV)

Returns
a function that must be called without arguments after resizing the plot.

reject(*value: float*) → bool

Return True if *value* is under the mask.

class hylight.mode.**Mode**(*lattice: ndarray*, *atoms: list[str]*, *n: int*, *real: bool*, *energy: float*, *ref: ndarray*, *eigenvector: ndarray*, *masses: Iterable[float]*)

Bases: object

The representation of a vibrational mode.

It stores the eigenvector and eigendisplacement. It can be used to project other displacement on the eigenvector.

energies()

Return the energy participation of each atom to the mode.

property energy_cm1

Energy of the mode in cm^{-1}.

property energy_eV

Energy of the mode in eV.

property energy_meV

Energy of the mode in meV.

property energy_si

Energy of the mode in J.

huang_rhys(*delta_R: ndarray*) → float

Compute the Huang-Rhyes factor.

$$S_i = 1/2\frac{\omega}{\hbar}[(M^{1/2^T}\Delta R)\cdot\gamma_i]^2 = 1/2\frac{\omega}{\hbar}\sum_i {m_i^{1/2}\gamma_i\Delta R_i}^2$$

Parameters

delta_R – displacement in SI

localization_ratio()

Quantify the localization of the mode over the supercell.

See also *participation_ratio()*.

1: fully delocalized >> 1: localized

participation_ratio()

Fraction of atoms active in the mode.

R J Bell et al 1970 J. Phys. C: Solid State Phys. 3 2111 https://doi.org/10.1088/0022-3719/3/10/013

It is equal to $M_1^2/(M_2M_0)$ where

$$M_n = \sum_\alpha m_\alpha ||\eta_\alpha||^{2^n}$$

where η_α is the contribution of atom α to eigendisplacement η.

But $M_0 = N$ by definition and $M_1 = 1$ because the eigenvectors are normalized.

Note: M_n = `np.sum(self.energies()**n)`

---

per_species_n_eff(*sp*)

Compute the number of atoms participating to the mode for a given species.

per_species_pr(*sp*)

Compute the fraction of atoms participation to the mode for a given species.

See also *per_species_n_eff()*.

project(*delta_Q: ndarray*) → float

Project delta_Q onto the eigenvector.

project_coef2(*delta_Q: ndarray*) → float

Square lenght of the projection of delta_Q onto the eigenvector.

project_coef2_R(*delta_R: ndarray*) → float

Square lenght of the projection of delta_R onto the eigenvector.

set_lattice(*lattice: ndarray*, *tol=1e-06*) → None

Change the representation to another lattice.

Parameters

- **lattice** – 3x3 matrix representing lattice vectors `np.array([a, b, c])`.
- **tol** – numerical tolerance for vectors mismatch.

to_jmol(*dest*, ***opts*)

Write a mode into a Jmol file.

See *hylight.jmol.export()* for the parameters.

to_traj(*duration*, *amplitude*, *framerate=25*)

Produce a ase trajectory for animation purpose.

Parameters

- **duration** – duration of the animation in seconds
- **amplitude** – amplitude applied to the mode in A (the modes are normalized)
- **framerate** – number of frame per second of animation

hylight.mode.**angle**(*v1*, *v2*)

Compute the angle in radians between two 3D vectors.

hylight.mode.**dynamical_matrix**(*phonons: Iterable[*Mode*]*) → ndarray

Retrieve the dynamical matrix from a set of modes.

Note that if some modes are missing the computation will fail.

Parameters

phonons – list of modes

hylight.mode.**generate_basis**(*seed*)

Generate an orthonormal basis with the rows of seed as first rows.

Parameters

seed – the starting vectors, a `(m, n)` matrix of orthonormal rows. `m = 0` is valid and will create a random basis.

Returns

a `(n, n)` orthonormal basis where the first `m` rows are the rows of `seed`.

hylight.mode.**get_HR_factors**(*phonons: Iterable[*Mode*]*, *delta_R_tot: ndarray*, *mask:* Mask | *None = None*) → ndarray

Compute the Huang-Rhys factors for all the real modes with energy above bias.

Parameters

- **phonons** – list of modes
- **delta_R_tot** – displacement in SI
- **mask** – a mask to filter modes based on their energies.

hylight.mode.**get_energies**(*phonons: Iterable[*Mode*]*, *mask:* Mask | *None = None*) → ndarray

Return an array of mode energies in SI.

hylight.mode.**modes_from_dynmat**(*lattice*, *atoms*, *masses*, *ref*, *dynmat*)

Compute vibrational modes from the dynamical matrix.

Parameters

- **lattice** – lattice parameters (3 x 3 numpy.ndarray)
- **atoms** – list of atoms
- **masses** – list of atom masses
- **ref** – reference position
- **dynmat** – dynamical matrix

Returns

list of *hylight.mode.Mode*

hylight.mode.**orthonormalize**(*m*, *n_skip=0*)

Ensure that the vectors of m are orthonormal.

Change the rows from n_seed up inplace to make them orthonormal.

Parameters

- **m** – the starting vectors
- **n_seed** – number of first rows to not change. They must be orthonormal already.

hylight.mode.**project_on_asr**(*mat*, *masses*)

Enforce accoustic sum rule.

Project the input matrix on the space of ASR abiding matrices and return the projections.

hylight.mode.**rot_c_to_v**(*phonons: Iterable[*Mode*]*) → ndarray

Rotation matrix from Cartesian basis to Vibrational basis (right side).

hylight.mode.**same_cell**(*cell1: ndarray*, *cell2: ndarray*, *tol=1e-06*) → *CellMismatch* | bool

Compare two lattice vectors matrix and return True if they describe the same cell.

4.1.7 hylight.mono_mode module

Simulation of luminescence spectra in 1D model.

hylight.mono_mode.**compute_spectrum**(*e_zpl*, *S*, *sig*, *e_phonon_g*, *e=None*)

Compute a spectrum from 1D model with experimental like inputs.

Parameters

- **e_zpl** – energy of the zero phonon line, in eV
- **S** – the emission Huang-Rhys factor
- **sig** – the lineshape standard deviation
- **e** – (optional, None) a numpy array of energies to compute the spectrum at if ommited, an array ranging from 0 to 3*e_zpl will be created.

E_phonon_g

the ground state phonon energy

hylight.mono_mode.**huang_rhys**(*stokes_shift*, *e_phonon*)

Huang-Rhys factor from the Stokes shift and the phonon energy.

4.1.8 hylight.multi_modes module

Simulation of spectra in nD model.

class hylight.multi_modes.**LineShape**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Line shape type.

GAUSSIAN = 0

LORENTZIAN = 1

NONE = 2

hylight.multi_modes.**compute_delta_R**(*poscar_gs*, *poscar_es*)

Return ΔR in A.

Parameters

- **poscar_gs** – path to ground state positions file.
- **poscar_es** – path to excited state positions file.

Returns

a numpy.ndarray of shape `(n, 3)` where `n` is the number of atoms.

hylight.multi_modes.**compute_spectrum**(*phonons*, *delta_R*, *zpl*, *fwhm*, *e_max=None*, *resolution_e=0.001*, *mask=None*, *shape=LineShape.GAUSSIAN*, *pre_convolve=None*, *load_phonons=<function load_phonons>*, *window_fn=<function hamming>*, *T=0*)

Compute a luminescence spectrum with the time-dependant formulation with an arbitrary linewidth.

Parameters

- **phonons** – list of modes (see `load_phonons()`) or path to load the modes
- **delta_R** – displacement in A in a numpy array (see *compute_delta_R()*) or tuple of two paths `(pos_gs, pos_es)`
- **zpl** – zero phonon line energy in eV
- **fwhm** – ZPL lineshape full width at half maximum in eV or None
  - if `fwhm is None or fwhm == 0.0`: the raw spectrum is provided unconvoluted
  - if `fwhm > 0`: the spectrum is convoluted with a gaussian line shape
  - if `fwhm < 0`: error
- **e_max** – (optional, `2.5*e_zpl`) max energy in eV (should be at greater than `2*zpl`)
- **resolution_e** – (optional, `1e-3`) energy resolution in eV
- **load_phonons** – a function to read phonons from files.
- **mask** – a *mode.Mask* instance to select modes base on frequencies
- **shape** – the lineshape (a *LineShape* instance)

- **pre_convolve** – (float, optional, None) if not None, standard deviation of a pre convolution gaussian
- **window_fn** – windowing function in the form provided by numpy (see `numpy.hamming()`)

Returns
`(energy_array, intensity_array)`

`hylight.multi_modes.`**duschinsky**(*phonons_a*, *phonons_b*)

Dushinsky matrix from b to a S_{agetsb}.

`hylight.multi_modes.`**dynmatshow**(*dynmat*, *blocks=None*)

Plot the dynamical matrix.

Parameters

- **dynmat** – numpy array representing the dynamical matrice in SI.
- **blocks** – (optional, None) a list of coloured blocks in the form `(label, number_of_atoms, color)`.

`hylight.multi_modes.`**fc_spectrum**(*phonons*, *delta_R*, *n_points=5000*, *disp=1*)

Build arrays for plotting a spectrum energy spectral function.

`hylight.multi_modes.`**freq_from_finite_diff**(*left*, *mid*, *right*, *mu*, *A=0.01*)

Compute a vibration energy from three energy points.

Parameters

- **left** – energy (eV) of the left point
- **mid** – energy (eV) of the middle point
- **right** – energy (eV) of the right point
- **mu** – effective mass associated with the displacement from the middle point to the sides.
- **A** – amplitude (A) of the displacement between the middle point and the sides.

`hylight.multi_modes.`**hr_spectrum**(*phonons*, *delta_R*, *n_points=5000*, *disp=1*)

Build arrays for plotting a spectrum phonon spectral function.

`hylight.multi_modes.`**make_line_shape**(*t*, *sigma_si*, *shape*)

Create the lineshape function in time space.

Parameters

- **t** – the time array (in s)
- **sigma_si** – the standard deviation of the line
- **shape** – the type of lineshape (an instance of *LineShape*)

Returns
a `numpy.ndarray` of the same shape as `t`

`hylight.multi_modes.`**plot_spectral_function**(*mode_source*, *poscar_gs*, *poscar_es*, *load_phonons=<function load_phonons>*, *use_cm1=False*, *disp=1*, *mpl_params=None*, *mask=None*)

Plot a two panel representation of the spectral function of the distorsion.

Parameters

- **mode_source** – path to the mode file (by default a pickle file)

- **poscar_gs** – path to the file containing the ground state atomic positions.
- **poscar_es** – path to the file containing the excited state atomic positions.
- **load_phonons** – a function to read mode_source.
- **use_cm1** – use cm1 as the unit for frequency instead of meV.
- **disp** – standard deviation of the gaussians in background in meV.
- **mpl_params** – dictionary of kw parameters for pyplot.plot.
- **mask** – a mask used in other computations to show on the plot.

Returns
`(figure, (ax_FC, ax_S))`

`hylight.multi_modes.rect(n)`
A dummy windowing function that works like numpy.hamming, but as no effect on data.

4.1.9 hylight.npz module

Serialization of modes to numpy zipped file.

See also `numpy.savez()` and `numpy.load()`.

`hylight.npz.archive_modes(modes, dest, compress=False)`
Store modes in dest using numpy's npz format.

Parameters

- **modes** – a list of Mode objects.
- **dest** – the path to write the modes to.

Returns
the data returned by load_phonons.

`hylight.npz.load_phonons(source)`
Load modes from a Hylight archive.

`hylight.npz.pops_and_masses(modes)`

4.1.10 hylight.struct module

A generic representation of a crystal cell.

`class hylight.struct.Struct(lattice, species, species_names=None)`
Bases: `object`

A general description of a periodic crystal cell.

Store all the required infos to describe a given set of atomic positions.

`property atoms`
List the species names in an order matching *self.raw*.

`copy()` → *Struct*
Return a copy of the structure.

classmethod from_mode(*mode:* Mode) → *Struct*

Extract the cell from a *hylight.mode.Mode*.

get_offset(*sp*)

Return the index offset of the given species block in *raw*.

property raw

Return an array of atomic positions.

This can be modified overwritten, but not modified in place.

sp_at(*i*)

Return the name of the species at the given index.

Parameters
i – the index of the atom.

Returns
the name of the species.

property species_names

Names of species in the same order as found in the raw positions.

property system_name

The name of the system, eventually generated from formula.

Can be overwritten.

4.1.11 hylight.typing module

Helper module for type annotations.

Should help with variable support of typing across versions of python and libraries.

4.1.12 hylight.utils module

Pervasive utilities.

exception hylight.utils.**InputError**

Bases: ValueError

An exception raised when the input files are not as expected.

hylight.utils.**gaussian**(*e*, *sigma*, *standard=True*)

Evaluate a Gaussian function on e.

Parameters

- **e** – abscissa
- **sigma** – standard deviation
- **standard** –
  - if True the curve is normalized to have an area of 1
  - if False the curve is normalized to have a maximum of 1

hylight.utils.**gen_translat**(*lattice: ndarray*)

Generate all translations to adjacent cells

Parameters

lattice – np.ndarray([a, b, c]) first lattice parameter

hylight.utils.**measure_fwhm**(*x*, *y*)

Measure the full width at half maximum of a given spectrum.

> **Warning:** It may fail if there are more than one band that reach half maximum in the array. In this case you may want to use select_interval to make a window around a single band.

Parameters

- **x** – the energy array
- **y** – the intensity array

Returns

FWHM in the same unit as x.

hylight.utils.**parse_formatted_table**(*lines*, *format*)

Parse a table of numbers from a list of string with a regex.

Parameters

- **lines** – a list of string representing the lines of the table
- **format** – a re.Pattern or a str representing the format of the line It should fullmatch each line or a ValueError exception will be raised. All the groups defined in format will be converted to float64 by numpy.

Returns

a np.array of dimension (len(lines), {number_of_groups})

Example:

```
>>> parse_formatted_table(["a=0.56 b=0.8 c=0.9"], "a=(.*) b=(.*) c=(.*)")
np.array([[0.56, 0.8, 0.9]])
```

hylight.utils.**periodic_diff**(*lattice*, *ref*, *disp*)

Compute the displacement between ref and disp, accounting for periodic conditions.

hylight.utils.**periodic_dist**(*lattice*, *ref*, *disp*)

Compute the distance between ref and disp, accounting for periodic conditions.

hylight.utils.**select_interval**(*x*, *y*, *emin*, *emax*, *normalize=False*, *npoints=None*)

Extract an interval of a spectrum and return the windows x and y arrays.

Parameters

- **x** – x array
- **y** – y array
- **emin** – lower bound for the window
- **emax** – higher bound for the window

- **normalize** – if true, the result y array is normalized
- **npoints** – if an integer, the result arrays will be interpolated to contains exactly npoints linearly distributed between emin and emax.

Returns
(windowed_x, windowed_y)

4.1.13 Module contents

The Hylight package.

Copyright (c) 2024, Théo Cavignac <theo.cavignac+dev@gmail.com>, The PyDEF team <camille.latouche@cnrs-imn.fr> Licensed under the EUPL

hylight.**setup_logging**()

Setup module logging.

BIBLIOGRAPHY

[Hylight] (Ref comming soon)

[BaZrO3:Ti] Cavignac, T.; Jobic, S.; C. Latouche, J. Chem. Theory Comput., 2022, 18, 12, 7714-7721

PYTHON MODULE INDEX

h

INDEX

I

L

M

N

O

P

R

S

T

V

W