Supporting Information

`shnitsel-tools`: A Toolkit for the Full Lifecycle of Surface Hopping Trajectory Data

Kevin Höllring[a,‡], Theodor E. Röhrkasten[a,‡], Carolin Müller[a,*]

[a] Friedrich-Alexander-Universität Erlangen-Nürnberg, Computer-Chemistry-Center, Nägelsbachstraße 25, Erlangen, 91052, Germany

[*] correspondence: carolin.cpc.mueller@fau.de
[‡] These authors contributed equally to this work.

Contents

Usage of the `shnitsel-tools` package

1 Technical Details and Performance

The `shnitsel-tools` (`st`) code is available on GitHub [1] and has been archived as a static version at the time of publication on Zenodo [2]. Both repositories provide detailed installation instructions, tutorials, and example test files to facilitate use and reproducibility. Dependencies include *e.g.* Matplotlib for plotting purposes, RDKit for visualizing structural formulas, Scikit-learn for performing Principal Component Analysis (PCA) and Kernel Density Estimation (KDE) calculations, and Scipy for calculating confidence intervals. To install `st`, please follow the instructions on GitHub [1].

During the parsing or loading process, `shnitsel-tools` (`st`) generates datasets that encapsulates a variety of quantum chemical properties associated with each molecular conformation. These properties, encompassing data types such as scalars, vectors, matrices, and tensors, are stored with their corresponding dimensions. A comprehensive overview of the supported property names and their dimensionalities, as handled by the SHNITSEL databases, is provided in **Table S1**.

Table S1: Extracted and generated properties with their shapes within a single frame and standardized units in the parsed dataset depending on the input format. Within a trajectory, the listed shapes for Observables (top) are prefixed with a `time` dimension to identify the time step whereas the shape of coordinates (bottom) remains unchanged. (Abbr.: *s*: state, *a*: atom, *d*: direction, *t*: time, *sc*: state combination)

Observable	**Key**	**SHARC**	**NewtonX**	**PyRAI^{2}MD**	**unit**	**shape**
Position	`atXYZ`	✓	✓	✓	Bohr	(a×d)
Energy	`energy`	✓	✓	✓	Hartree	(s)
E_{kin}	`e_kin`	✓	✓	✓	Hartree	(scalar)
Forces	`forces`	✓	✓	✓	Hartree·Bohr^{-1}	(s×a×d)
Perm. dipole (μ_i)	`dip_perm`	✓	✓	✓	Hartree·Bohr^{-1}	(s×d)
Trans. dipole (μ_{ij})	`dip_trans`	✓	✓	✓	Hartree·Bohr^{-1}	(sc×d)
NACs	`nacs`	✓	✓	✓	Bohr^{-1}	(sc×a×d)
SOCs	`socs`	✓		✓	cm^{-1}	(sc)
Phases	`phases`				1	(2^{s-1})
State name	`state_names`	✓	✓	✓	`str`	(s)
State multiplicities	`state_types`	✓	✓	✓	$\{1,2,3\}$	(s)
State charges	`state_charges*`	✓			`int`	(s)
State combinations	`statecomb`	✓	✓	✓	`(int,int)`	(sc)
Active state (MCH)	`astate`	✓	✓	✓	`int`	(t)
Active state (diag)	`sdiag`				`int`	(t)
Atom names	`atNames`	✓	✓	✓	`str`	(a)
Atomic numbers (Z)	`atNums`	✓	✓	✓	`int`	(s)

* Charge information is only available reliably starting with SHARC 4.0

In the current implementation, complete trajectory ensembles are loaded into memory as `xarray.Dataset` objects. For the size of datasets typically encountered in nonadiabatic photochemical simulations, this is well within the capabilities of modern workstations. The memory requirements scale approximately with the number of atoms, electronic states and time steps. For example, a system with 60 atoms, 5 electronic states, and 200 time steps requires only a few megabytes per trajectory, while even much larger systems remain manageable on standard hardware (see **Tables S2** and **S3**).

Table S2: Benchmark of sequential and parallel `read()` performance for different input formats. Reported times are averages over 10 runs and include the overhead associated with parallel setup. The initialization of `xarray.Dataset` objects introduces a noticeable cost, particularly for initial-condition files with low information density, and for the tested trajectory counts the overhead of parallelization outweighs its performance benefits. Overall, input performance is high across all formats, with processing times on the order of seconds per 10,000 frames for most cases.

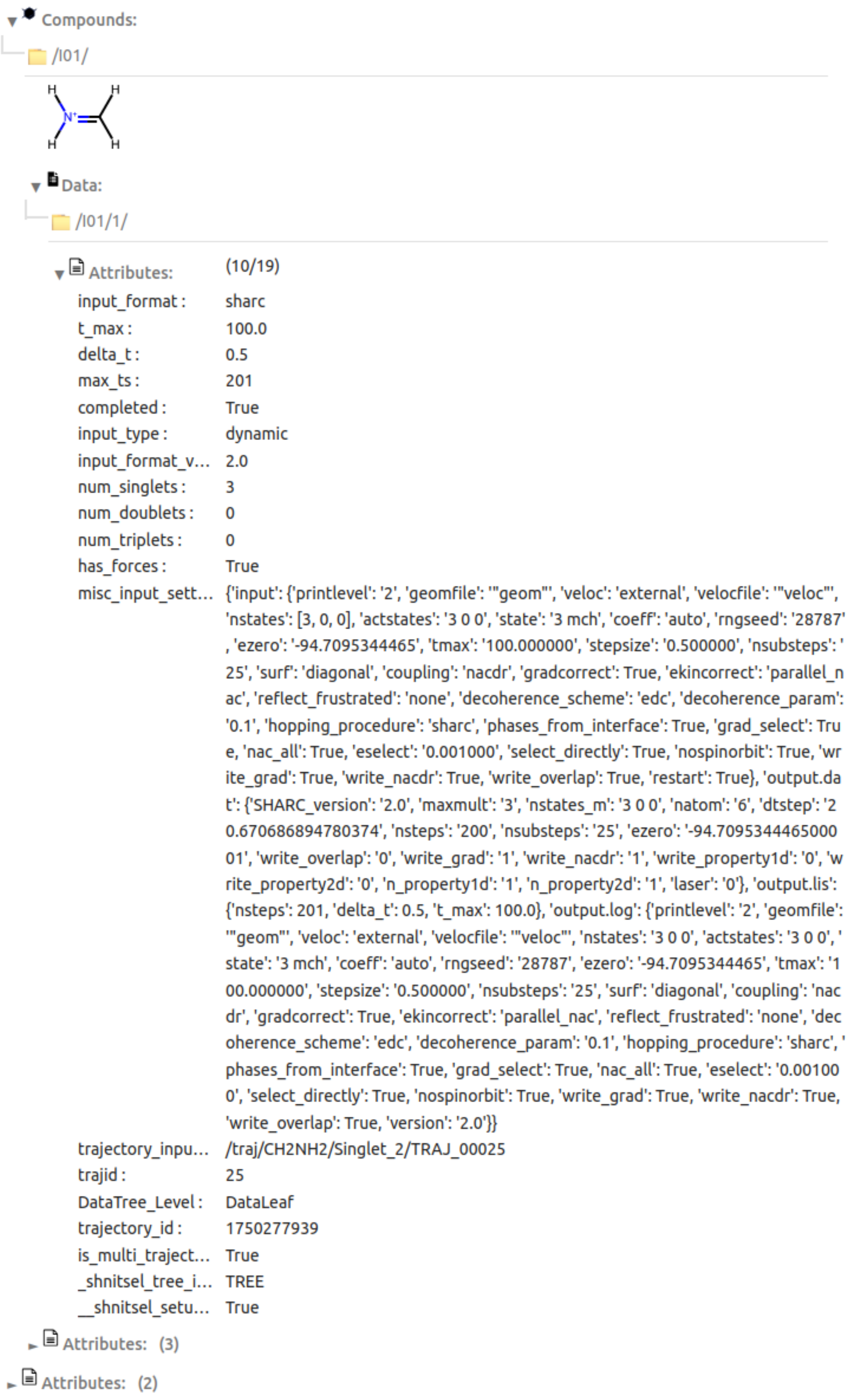
	format	#trajectories / rep	#frames / rep	time / rep	time / rep / 100 frames
			in s		in s
serial	Newton-X	5.0	5605.0	0.5932	0.0105
serial	SHARC (icond)	4.0	4.0	0.1435	3.5887
serial	SHARC (traj)	11.0	2185.0	1.0511	0.0481
serial	PyRAI^{2}MD	3.0	600.0	0.1314	0.0219
parallel	Newton-X	5.0	5605.0	0.5933	0.0105
parallel	SHARC (icond)	4.0	4.0	0.1993	4.9841
parallel	SHARC (traj)	11.0	2185.0	1.4670	0.0671
parallel	PyRAI^{2}MD	3.0	600.0	0.2498	0.0416

Table S 3: Comparison of storage requirements for surface hopping trajectory data. The table reports the disk usage of raw surface hopping outputs (ASCII files, excluding orbital data) and the corresponding `shnitsel-tools` datasets stored in two formats: an unstacked tree, where each trajectory is represented as an individual leaf, and a stacked tree, where trajectories are aggregated into dataset-level leaves. Both, the stacked and unstacked tree files are located on Zenodo [3].

Molecule	Original / MB*	shnitsel (tree) / MB*	shnitsel (stacked tree) / MB*
A01	1709.6	69.6	41.6
A02	2169.8	81.7	53.1
A03	861.1	57.7	42.8
I01	358.1	13.4	7.9
R02**	1116.2	174.9	163.6

* Decimal megabytes, *i.e.* 1 MB = 10^6 bytes.

** The original size given for **R02** combines the **R02a** and **R02b** subsets.



```
▾ Compounds:
  └─ /I01/

     ▾ Data:
       └─ /I01/1/

          ▾ Attributes:       (10/19)
            input_format :    sharc
            t_max :           100.0
            delta_t :         0.5
            max_ts :          201
            completed :       True
            input_type :      dynamic
            input_format_v…   2.0
            num_singlets :    3
            num_doublets :    0
            num_triplets :    0
            has_forces :      True
            misc_input_sett…  {'input': {'printlevel': '2', 'geomfile': '"geom"', 'veloc': 'external', 'velocfile': '"veloc"',
                              'nstates': [3, 0, 0], 'actstates': '3 0 0', 'state': '3 mch', 'coeff': 'auto', 'rngseed': '28787'
                              , 'ezero': '-94.7095344465', 'tmax': '100.000000', 'stepsize': '0.500000', 'nsubsteps': '
                              25', 'surf': 'diagonal', 'coupling': 'nacdr', 'gradcorrect': True, 'ekincorrect': 'parallel_n
                              ac', 'reflect_frustrated': 'none', 'decoherence_scheme': 'edc', 'decoherence_param':
                              '0.1', 'hopping_procedure': 'sharc', 'phases_from_interface': True, 'grad_select': Tru
                              e, 'nac_all': True, 'eselect': '0.001000', 'select_directly': True, 'nospinorbit': True, 'wr
                              ite_grad': True, 'write_nacdr': True, 'write_overlap': True, 'restart': True}, 'output.da
                              t': {'SHARC_version': '2.0', 'maxmult': '3', 'nstates_m': '3 0 0', 'natom': '6', 'dtstep': '2
                              0.670686894780374', 'nsteps': '200', 'nsubsteps': '25', 'ezero': '-94.7095344465000
                              01', 'write_overlap': '0', 'write_grad': '1', 'write_nacdr': '1', 'write_property1d': '0', 'w
                              rite_property2d': '0', 'n_property1d': '1', 'n_property2d': '1', 'laser': '0'}, 'output.lis':
                              {'nsteps': 201, 'delta_t': 0.5, 't_max': 100.0}, 'output.log': {'printlevel': '2', 'geomfile':
                              '"geom"', 'veloc': 'external', 'velocfile': '"veloc"', 'nstates': '3 0 0', 'actstates': '3 0 0', '
                              state': '3 mch', 'coeff': 'auto', 'rngseed': '28787', 'ezero': '-94.7095344465', 'tmax': '1
                              00.000000', 'stepsize': '0.500000', 'nsubsteps': '25', 'surf': 'diagonal', 'coupling': 'nac
                              dr', 'gradcorrect': True, 'ekincorrect': 'parallel_nac', 'reflect_frustrated': 'none', 'dec
                              oherence_scheme': 'edc', 'decoherence_param': '0.1', 'hopping_procedure': 'sharc', '
                              phases_from_interface': True, 'grad_select': True, 'nac_all': True, 'eselect': '0.00100
                              0', 'select_directly': True, 'nospinorbit': True, 'write_grad': True, 'write_nacdr': True,
                              'write_overlap': True, 'version': '2.0'}}
            trajectory_inpu…  /traj/CH2NH2/Singlet_2/TRAJ_00025
            trajid :          25
            DataTree_Level :  DataLeaf
            trajectory_id :   1750277939
            is_multi_traject… True
            _shnitsel_tree_i… TREE
            __shnitsel_setu…  True

       ▸ Attributes:  (3)

  ▸ Attributes:  (2)
```

Figure S1: Jupyter notebook tree view of the I01 dataset loaded in stacked format from Zenodo [3]. The view illustrates the metadata stored with the dataset, including the surface hopping engine (here SHARC), trajectory identifiers, maximum simulation time, and the full set of simulation parameters (*e.g.*, time step, decoherence scheme, and other SHARC input keywords).

2 Data structure

To best accommodate the multidimensional nature of surface hopping data, we have chosen `xarray` [4] as the core data structure. Although `xarray` is more commonly used in fields such as geosciences, it provides an efficient, NumPy-compatible data model that closely mirrors hierarchical NetCDF storage and is well suited for large multidimensional scientific datasets.

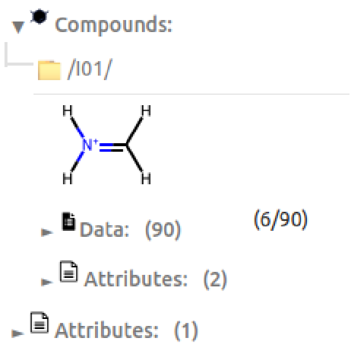
The core `xarray` data structures used in `st` are:

- `DataArray`: a labeled, multi-dimensional array that wraps a `numpy.ndarray`. Labeled dimensions enable Pandas-like indexing and automatic alignment of variables with matching dimension names, and NumPy-style functions can be applied using `xarray.apply_ufunc()`. Surface-hopping trajectories are often *ragged*, meaning that trajectories have different lengths. Since `xarray` requires rectangular arrays, stacking multiple trajectories along a new `traj` dimension (referred to as *layering*) pads missing frames with `NaN` values. While this enables efficient vectorized operations over trajectories or time steps, it can increase memory usage and requires careful handling of missing values. Alternatively, trajectories can be *appended* along a single `time` dimension, such that frames from different trajectories are concatenated sequentially and indexed by a `(traj, time)` MultiIndex. This representation is well suited for operations over all frames, such as constructing histograms. In `xarray`, conversion between layered and appended forms is performed using `stack` and `unstack`.

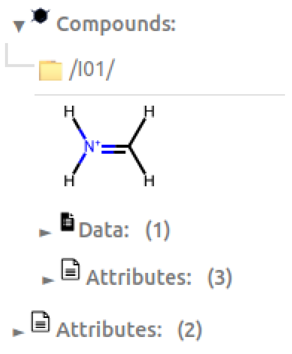
- `Dataset`: a collection of multiple `DataArray`s that share common dimensions. Conceptually, a `Dataset` extends the idea of a `pandas.DataFrame` to higher dimensions, with each `DataArray` corresponding to a column. Variables can be accessed like dictionary entries while remaining aligned across all dimensions.

- `DataTree`: a hierarchical container for organizing multiple `Dataset`s. In `st`, a `DataTree` can represent either a *stacked* tree, where each leaf contains a dataset of stacked trajectories, or an *unstacked* tree, where each leaf contains an individual trajectory (see **Figure S2**). Trees can also be structured to reflect different molecules, electronic-structure methods, or simulation settings, enabling systematic cross-system and cross-method analyses within a single hierarchy. Numerical operations are vectorized across tree nodes, and different trees can be broadcast against each other. `st` further extends the `DataTree` model by allowing additional object types at different hierarchy levels to encode chemically and computationally meaningful groupings. Example trees for compounds **A01**, **A02**, **A03**, **I01**, and **R02** are provided in the `shnitsel-data` repository on Zenodo [3, 5].

`st` is designed with the assumption that all quantum chemical properties are consistently available for every geometry in a dataset. While it is generally preferable to maintain completeness, missing properties can occasionally arise in certain cases. To ensure flexibility, `st` allows for incomplete datasets, meaning that some geometries may lack specific quantum chemical properties and, more importantly, that trajectories within an ensemble can have different lengths. The default `DataTree` structure allows for this, as trajectories of different lengths can be kept each as a separate node. Even multiple ensembles compounds with different numbers of atoms or states can be stored unambiguously in the same object.

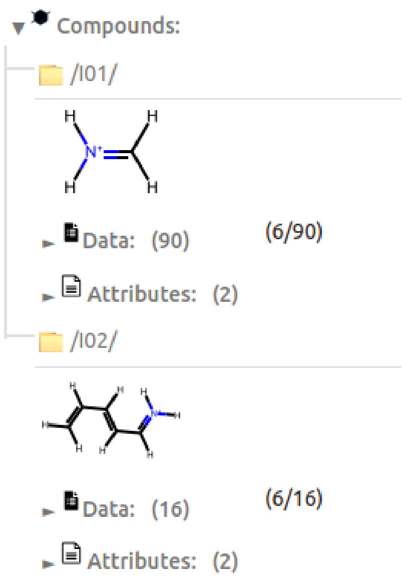


(a)

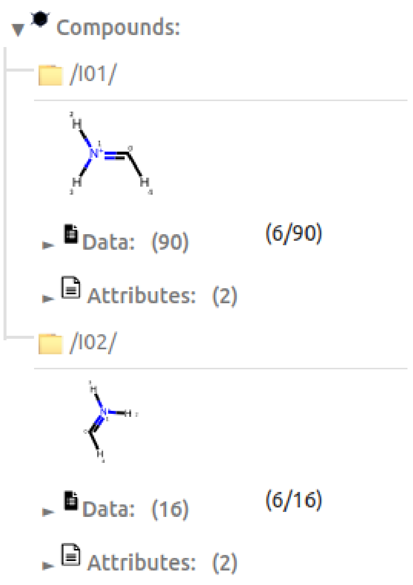


(b)

Figure S2: Comparison of the Jupyter notebook tree view for the I01 dataset loaded in unstacked (a) and stacked (b) formats from Zenodo [3]. Both representations contain 90 trajectories; in the unstacked tree each trajectory appears as an individual leaf, whereas in the stacked tree all trajectories are combined into a single dataset.



(a) full-structure



(b) maximum common substructure

Figure S3: Comparison of the tree representation for the unstacked I01 and I02 datasets before and after extraction of their maximum common substructure (MCS). The initial tree contains **I01** and **I02** as separate branches, which are transformed into substructure aligned datasets after MCS selection. The **I01** data (here *unstacked* was used) are available on Zenodo [3], and the **I02** data are provided in the tutorial section of the `shnitsel-tools` GitHub repository [1].

3 Key features and methodologies

3.1 Parse, write and load data

The `st` package provides dedicated functionality for parsing trajectory data from standard output files generated by the widely used surface hopping program suits SHARC [6, 7] (versions 2.0 to 4.0), Newton-X [8, 9] (versions 2.0 to 2.6), and PyRAI^{2}MD [10, 11] (version 2.4) *via* the `st.io` module. Furthermore, this module supports direct parsing of initial condition files from SHARC simulations, i.e., `QM.out` files generated by SHARC, through the same interface. The resulting `DataArrays` can be stored in NetCDF4 format. Examples of both data parsing and loading are provided below.

- **Example: Parsing SHARC Initial Conditions**

The following example demonstrates how to parse the quantum chemistry output from SHARC initial conditions (`ICOND`-folders) and store it in the `shnitsel-tools` database. The code automatically verifies the availability of the required quantum chemical properties, as listed in **Table S1**.

```
import shnitsel_tools as st
# parse icond data
iconds_butene = st.io.read(path='./test_data/sharc/iconds_butene')
# save the parsed data to h5netcdf file
savepath = "./test_data/sharc/iconds_butene.nc"
st.io.write_shnitsel_file(iconds_butene, savepath=savepath)
```

- **Example: Parse trajectory data from SHARC and Newton-X**

The following example demonstrates how to parse trajectory data from quantum chemistry simulations performed with SHARC [6, 7], Newton-X [8, 9] and PyRAI^{2}MD[10, 11]. The respective test data and jupyter notebook tutorials (`0_1_sharc2hdf5.ipynb` and `0_2_nx2hdf5.ipynb`) are compiled on Github [1].

```
import os
import shnitsel as st
# parse trajectory data from SHARC output files
traj_I01_sharc = st.io.read('test_data/sharc/traj_I01_v4.0', kind='sharc')

# parse trajectory data from Newton-X output files
traj_I01_nx = st.io.read('test_data/newtonx/test_I01_v2.6', kind='nx')

# parse trajectory data from PyRAI2MD output files
traj_I01_py = st.io.read('test_data/pyrai2md/traj_I01', kind='pyrai2md')
```

- **Example: Loading a `shnitsel-tools` (SHNITSEL) database**

When working with an existing `shnitsel-tools` database, *i.e.*, provided in the tutorials on Github (sample trajectories of the retinal model system, `traj_I02.nc`) [1], the data can be loaded following the subsequently given example.

```
# Read SHNITSEL data in netcdf format
import shnitsel as st
st.io.read('tutorials/test_data/shnitsel/traj_I02.nc')
```

Users should note that the trajectory datasets in the `shnitsel-data` [5] and Zenodo repository [12] have been updated to conform to the `shnitsel-tools` data format [3]. In this format, trajectories are stored as a stacked tree, where each branch corresponds to a dataset; by contrast, in an unstacked tree each branch or leaf represents an individual trajectory. The updated data trees generated by `st` [3] also include additional metadata, such as property units, electronic-structure reference methods, and charge information, which were not present in the datasets reported in Ref. [12].

- **Extending the input formats**

`Shnitsel-tools` is built with modularity and extensibility in mind. This is also the case for its input routines, which can easily be extended to support additional formats. Two key requirements are necessary for `shnitsel-tools` to support additional formats in its `st.io.read()` function:

- The format needs to be registered with the format registry by calling the corresponding registration function, i.e. *via* `st.io.format_registry.register_format_reader()`, where the name or rather *identifier* for the format (e.g. `ase`, `nx`, `pyrai2md`) is associated with an format reader instance.
- An implementation of the corresponding format reader. To facilitate this, `shnitsel.io` provides the abstract base class `FormatReader`, which defines the interface that each new format reader must implement. At a minimum, the following methods are required:
  - `find_candidates_in_directory()`: This method is called to identify candidate entries in a directory if `read()` is called with a path to a directory and multiple trajectories may be read in parallel
  - `check_path_for_format_info()`: This method is called to check a specific path for the correct format supported by the respective reader. In principle, it can be used to support the candidate detection.
  - `read_from_path()`: reads the dataset from a file that has been identified as being of the correct format by `check_path_for_format_info()`. This method performs the actual parsing of the input data.
  - `get_units_with_defaults()`: This method is supposed to set default units for the respective format that may not be detected during the parsing process. It supports user-provided unit overrides and should be provided. For information about the units, refer to the `shnitsel.units` module.

Target variable names you parse from your input formats are listed in **Table S1**. Additional data may be present at your discretion, but should adhere to the overall rule that variables with units should have the unit specified in their `units` attribute.

By default, the `FormatReader` interface performs some final transformation to the data parsed from input formats. This finalization code is written in `read_data()` and involves conversion, completion of missing state names, assignment of state types, etc. and overall dummy values for some missing information for certain formats. If you do not wish for these transformations to be applied, please also provide your own implementation of `read_data()`.

3.2 Data pre-processing (`st.clean`)

The `st.clean` module implements filtering based on standard energetic criteria used in SH simulations (*e.g.*, SHARC diagnostics [7]) and on bond-length based geometric thresholds, using multiple filtering strategies. Energetic descriptors are derived from kinetic, potential, and total energies and include total energy drift from $t = 0$, frame-to-frame energy changes, and explicit tracking of state hopping events to detect issues such as timestep inconsistency, energy conservation violations, or hopping artifacts. Default energy and geometry thresholds are listed in **Table S5** and are automatically converted between units. These thresholds can be modified via the `energy_thresholds` and `geometry_thresholds` parameters by passing dictionaries. For geometry based filtering, custom or additional features can be specified, typically involving two atoms, using a bond SMARTS pattern as the dictionary key and the corresponding threshold (in Å) as the value. An example is given below.

```python
import os
import shnitsel as st
from shnitsel.clean import sanity_check
# parse trajectory data from SHARC output files
dt_I01 = st.io.read('test_data/sharc/traj_I01_v4.0', kind='sharc').set_compound_info('I01')
dt_I01_clean = sanity_check(dt_I01, geometry_thresholds={"[*]~[*]": 2.25, "[#6,#7][#1]":1.5})
dt_I01_clean
```

Table S4: Default features and thresholds for sanity-check (filtration by energy and bond-length). Bonds are selected using SMARTS from a connectivity-graph determined by a single reference conformation

Type	Name	Explanation	Default value
Energetic	`epot_active_step`	The change in active-state potential energy from one step to the next	0.7 eV
	`epot_hop_step`	The change in active-state potential energy at a hop	1.0 eV
	`ekin_step`	The change in kinetic energy from one step to the next	0.7 eV
	`etot_step`	The change in total energy from one step to the next	0.1 eV
	`etot_drift`	The change in total energy from the beginning of a trajectory	0.2 eV
Geometric	`[*]~[*]`	The length of any bond	300 pm
	`[#6,#7][H]`	The length of any C-H or N-H bond	200 pm

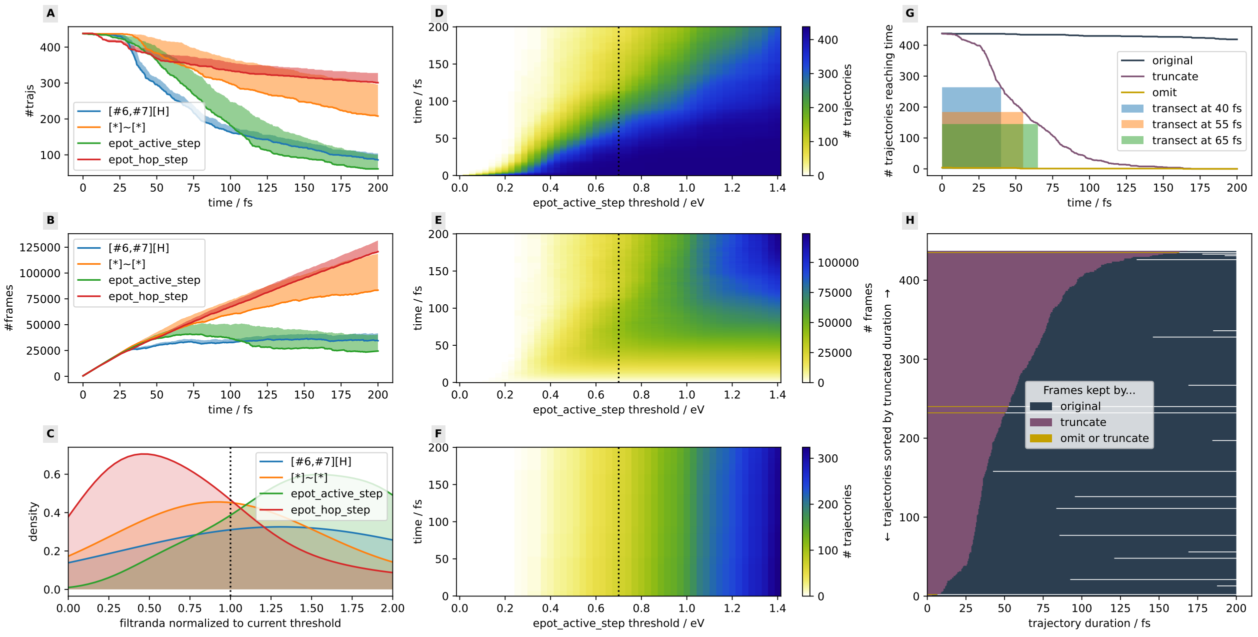


Figure S4: Illustration of the data-cleaning workflow for the unfiltered `shnitsel-data` A01 (ethene) ensemble [3]. Panels **A**–**C** provide guidance for selecting filtering criteria and thresholds for the *truncate*, *transect*, and *omit* strategies. **A**: Time-dependent populations of valid trajectories for each filtering criterion (solid lines), with shaded regions indicating the effect of increasing each threshold by 20 %. C–H and N–H bond lengths are initially most restrictive, whereas the potential energy step dominates at later times. **B**: Number of frames retained by each criterion upon transection at a given time, with corresponding 20 % threshold variations. **C**: Distributions of per-trajectory feature maxima relative to the current thresholds; the *omit* strategy retains only trajectories whose maxima lie below the threshold (central line). For this ensemble, `epot_active_step` is the dominant limiting feature. Panels **D**–**F** illustrate how varying a single threshold, namely the step size of the potential energy of the active state, affects data retention. **D** and **E** show the number of retained trajectories and frames, respectively, as functions of this threshold when using the *transect* strategy, while **F** shows the corresponding trajectory retention for the *omit* strategy. The vertical slices in **D** and **E** correspond to the green curves in **A** and **B**, respectively. **G**: Populations retained by each filtering strategy at the default threshold values, including three example transections. *Transect* and *omit* yield subsets of *truncate*, although *omit* may exceed any individual transection depending on the criteria. **H**: Final outcome of each trajectory for the different strategies.

3.3 Data processing

`shnitsel-tools` provides a broad set of tools for processing and interrogating trajectory ensembles and their associated properties. The available functionality includes:

- **Hopping analysis:** selection of specific surface-hopping events (e.g., state- or direction-resolved), alignment of trajectories to hopping times, and handling of multiple hops either per trajectory or per event.
- **Dimensionality reduction:** principal component analysis (PCA), linear discriminant analysis (LDA), and partial least squares (PLS).
- **Geometric descriptors:** internal coordinates, bond length alternation and pairwise distances, including optional geometry alignment via the Kabsch algorithm.
- **Populations:** classical state populations based on the active adiabatic state.
- **Unit handling:** automatic unit conversion across all analysis routines.
- **Statistical analysis:** ensemble averages, confidence intervals, normalization, baseline subtraction, and combinations of observables (e.g., energy differences).
- **Electronic spectra:** Calculation electronic absorption data from transition energies and transition dipole moments and Gaussian broadening of the resulting data to obtain absorption spectra.

The following sections illustrate these capabilities with representative workflows focused on the analysis of geometric and property spaces, including PCA based on different structural descriptors and subsets of trajectories, as well as time-resolved plots for selected electronic states.

3.3.1 Geometric analysis of SH trajectories

`st` provides a set of tools for exploring and comparing the geometric space sampled during surface hopping molecular dynamics. These include the calculation of chemically meaningful internal coordinates (bond lengths, bond angles, pyramidalization angles and torsion angles), pairwise distances as well as bond-length alternation in conjugated chromophores. All descriptors can be evaluated for the full molecule or for selected substructures and analyzed as time series or statistical distributions, for example to characterize dihedral-angle distributions at surface hopping events.

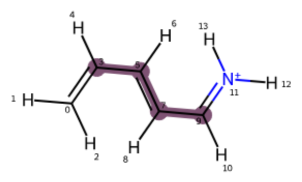
To enable meaningful comparisons across related systems with different atom counts or numbering, `st` supports the selection of atomically compatible substructures based on molecular topology. Substructures can be defined by user-provided SMARTS patterns or determined automatically using `RDKit`'s maximum common substructure (MCS) algorithm, after which atoms are reordered consistently across datasets. This allows multiple ensembles to be combined and analyzed within a common coordinate framework.

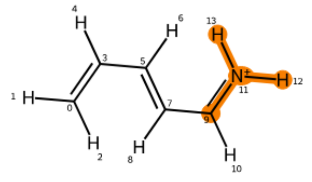
In the following, these capabilities are illustrated with two examples. First, we demonstrate structure selection for the retinal model system **I02**. Second, using butene (**A03**), we show how principal component analysis (PCA) of trajectory ensembles depends on the data-pre-processing, and both, the chosen subset of structures and the geometric descriptors employed, comparing internal-coordinate and pairwise-distance representations.

- **Structure selection (I02)**

In the following we demonstrate the selection of certain substructures of **I02**, namely the central CC=CC dihedral angle, defined by a SMARTS, the pyramidalisation angle of the immonium unit defined by atom indices and the automatic selection of the conjugated chromophore, which can be used for calculating the bond length alternation (BLA). A detailed tutorial on structure selection, illustrated for I02, is available in the tutorials section of the `shnitsel-tools` GitHub repository [1].

```
import shnitsel as st
from shnitsel.filtering import StructureSelection

# load data of I02 and stack trajectories
dt_I02 = st.io.read('../shnitsel-tools/tutorials/test_data/shnitsel/traj_I02.nc').set_charge(1)
ds_I02 = dt_I02.as_stacked

# select all features
base_selection = ds_I02.st.struc_sel(['atoms', 'bonds', 'angles', 'pyramids', 'dihedrals'])

# define substructures by smarts or indices
smarts_dih = '[#6;D3]C=C[#6;D3]'
idx_pyr = (11,(12,13,9))

# select substructures
dih_selection = base_selection.select_dihedrals(smarts_dih)
pyr_selection = base_selection.select_pyramids(idx_pyr)
bla_selection = base_selection.select_BLA_chromophor()

# visualize results
dih_selection.draw(flag_level='dihedrals'), pyr_selection.draw(flag_level='pyramids'), bla_selection.draw(
    flag_level='dihedrals')
```


(a) dihedral


(b) pyramids

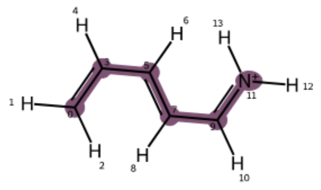

(c) BLA

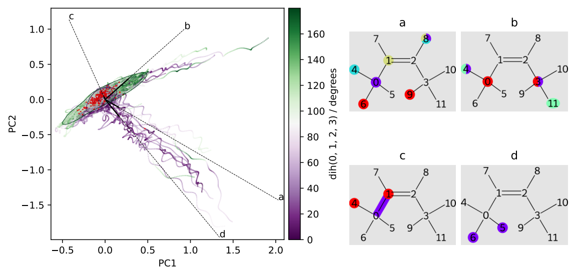
Figure S5: Illustration of structure-selection annotations for the **I02** molecular structure, highlighting the identified torsion angle (a), pyramidalization angle (b), and bond length alternation (BLA) chromophore (c). The color coding reflects the corresponding feature-flag levels.

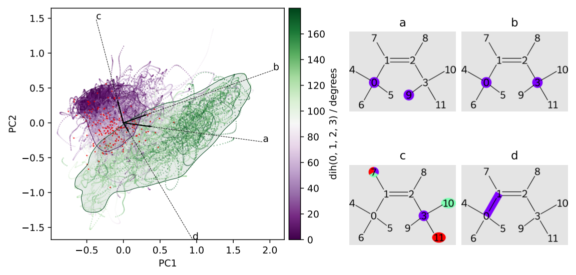
• Exploring the geometry space of butene (A03)

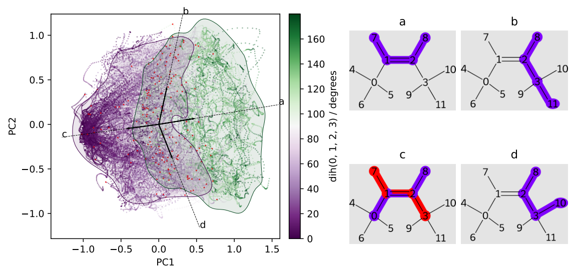
A **biplot** represents PCA-reduced data alongside loadings, which are projections of the original structural variables onto the reduced space. In the case of PCA on atom-atom distances, each loading corresponds to a specific pair-wise distance $|\mathbf{r}_{ij}|$. If a given configuration exhibits an increase in $|\mathbf{r}_{ij}|$, its projection in the reduced space shifts in the direction of the corresponding loading, with the magnitude of movement proportional to the loading's strength. However, because multiple loadings often point in similar directions, identifying the exact structural changes responsible for a given projection can be challenging. To aid interpretation, `st` provides an option to visualize relevant molecular structures using `RDKit`, highlighting the atom pairs associated with clustered loadings (see **Figure S6**).

The following shows an example to perform PCA on pairwise distances as descriptor without any data pre-processing for the ensemle data of butene (**A03**) as reported on Zenodo [3], yielding the plot as shown in **Figure S6a**.

```python
import shnitsel as st
import shnitsel.xarray

# download data from zenodo and load it
dt_A03 = st.read('A03_butene.nc').as_stacked
st.vis.plot.biplot_kde(dt_A03, 0,1,2,3, geo_kde_ranges=[(0,3), (5,20)], scatter_color_property='geo')
```


(a) read $\rightarrow r_{ij} \rightarrow$ PCA


(b) read $\rightarrow$ clean $\rightarrow r_{ij} \rightarrow$ PCA


(c) read $\rightarrow$ clean $\rightarrow$ BATs $\rightarrow$ PCA

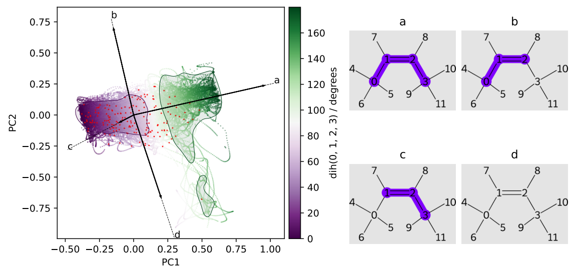

(d) read $\rightarrow$ clean $\rightarrow$ filtering $\rightarrow$ BATs $\rightarrow$ PCA

Figure S6: Comparison of PCA biplots for butene (A03) under different data-processing and descriptor choices. **(a)** PCA of pairwise distances computed from the raw trajectory data. **(b)** PCA of pairwise distances after filtering the data using standard energy and geometry thresholds. **(c)** PCA of bonds, angles, and torsions (BATs) of the full molecule after filtering. **(d)** PCA of BATs computed for a heavy-atom substructure selected *via* structure selection (*e.g.*, SMARTS `[#6][#6][#6][#6]`), again after filtering.

Using the hops module, specific surface-hopping events can be selected and analyzed in terms of their geometric characteristics. In the following example, the **A03** dataset is loaded and cleaned, the dihedral angle of the central CC=CC substructure is computed for all trajectories, and $S_1 \rightarrow S_0$ hopping events are selected to visualize the corresponding dihedral-angle distribution.

```python
import shnitsel as st
import xarray as xr
import shnitsel.xarray
from shnitsel.geo.geocalc import get_dihedrals
from shnitsel.analyze.hops import filter_data_at_hops
import matplotlib.pyplot as plt
from shnitsel.vis.colormaps import st_grey

# load and clean data
dt_A03 = st.io.read('data/tree/stacked/A03_butene_stacked.nc').set_charge(0).set_compound_info('A03')
dt_A03_clean = st.clean.sanity_check(dt_A03, geometry_thresholds={"[*]~[*]": 2.25, "[#6,#7][#1]":1.5})

# get dihedrals for central heavy atoms
dt_dih = get_dihedrals(dt_A03_clean, structure_selection='[#6][#6]=[#6][#6]', deg=True, signed=True)

# extract dihedrals at S1->S0 hopping points and plot their distribution
hops_positions_21 = filter_data_at_hops(dt_dih, "2->1")

plt.hist(hops_positions_21.as_stacked.isel(descriptor=0), bins=20,
         color=st_grey, edgecolor=st_grey, linewidth=1.0, alpha=0.85)
plt.xlim([-180,180])
plt.xlabel(f"${hops_positions_21.as_stacked.attrs['long_name']}$ / degree")
plt.ylabel("count")
```

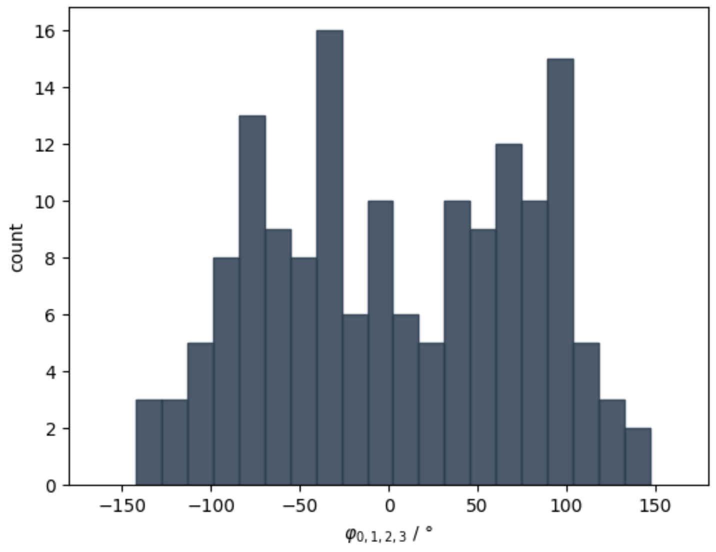


Figure S7: Distributions of the CC=CC dihedral angles of **A03** extracted at $S_1 \rightarrow S_0$ hopping points.

3.3.2 Analysis of electronic properties of SH trajectories

In the following example we show how to inspect the per-state distributions of properties across the ensemble.

```
import shnitsel as st
import shnitsel.xarray
from shnitsel.vis.datasheet import Datasheet

dt_A03 = st.read('A03_butene_stacked.nc')
dt_A03_clean = st.sanity_check(dt_A03)
sheet = Datasheetdt_A03_clean
sheet.plot_per_state_histograms()
```

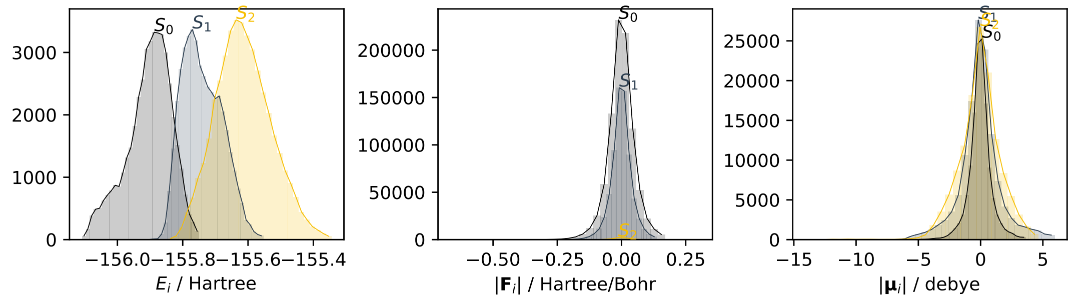


Figure S8: Global distributions of energies, forces, and permanent dipole moments obtained from a trajectory ensemble of **A03** [3], generated using the `plot_per_state_histograms` function from the `Datasheet` class (see code example above).

`st` provides a flexible plotting framework for visualizing the time evolution of trajectory dependent observables across multiple trajectories and electronic states. Trajectories can be displayed as individual traces or aggregated into bootstrap-based confidence intervals. State- or transition-resolved observables can be displayed within a single plot using color coding, or, when color encodes other information, as separate panels for each state or state combination.

High-density trajectory overlays are supported *via* datashader, enabling convolution-like "hairplots" that reveal regions of high trajectory overlap via the `timeplot` function (see **Figure S9**).

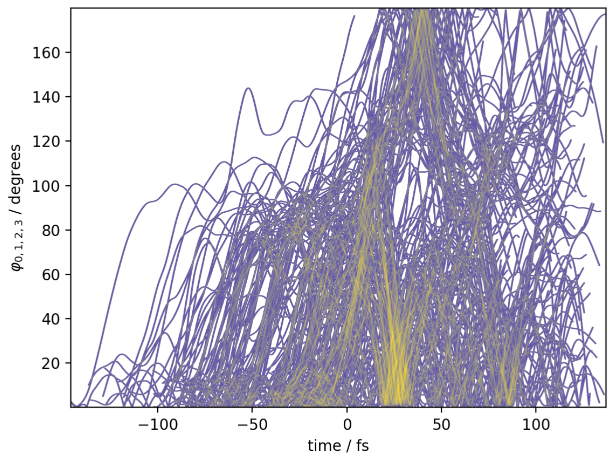


Figure S9: Unsigned dihedral angles of the central CC=CC substructure of **A03**, with trajectories aligned to their last $S_1 \rightarrow S_0$ hopping event. Colors indicate the density of overlapping trajectories, ranging from blue (low) to yellow (high).

A standard collection of time-resolved plots, including state-resolved energies, populations, and energy gaps, is available through the datasheet module and can be generated using the example below.

```python
import shnitsel as st
import shnitsel.xarray
from shnitsel.vis.datasheet import Datasheet

dt_A03 = st.read('A03_butene_stacked.nc')
dt_A03_clean = st.sanity_check(dt_A03)
sheet = Datasheetdt_A03_clean
sheet.plot_timeplots()
```

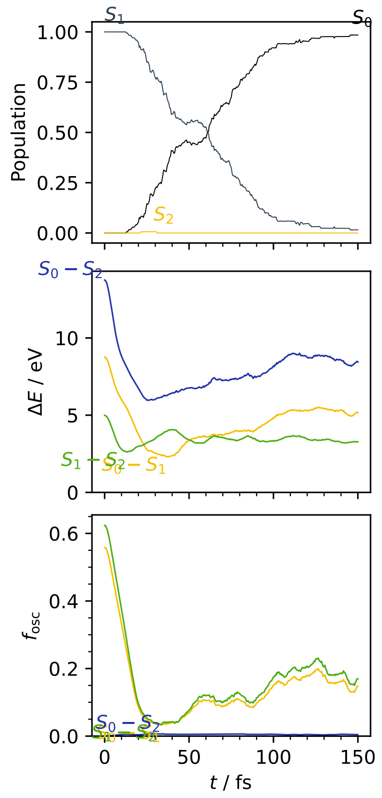


Figure S 10: Visualization of the evolution of populations in the three lowest singlet states, along with the corresponding energy gaps and oscillator strengths for transitions between these states (from top to bottom) for **A03**.

3.4 Data visualization

st also provides a unified plotting interface for visualizing both structural and electronic observables. In the data pre-processing stage filtering results can be visualized as validity and population curves. Other supported key representations include PCA biplots, time-resolved plots of properties along individual trajectories or ensemble averages with statistical summaries (*e.g.*, confidence intervals or convolutions), and *ski plots* of the electronic absorption spectra. Furthermore molecular structures and trajectories can be exported for visualization in VMD. An overview of a dataset can be generated and stored as multi-page PDF file using the Datasheet class.

Visualization in st is tailored to the type of observable: state-resolved quantities are shown as state-colored histograms, inter-state properties as two-dimensional correlation plots, and tensorial quantities (*e.g.*, forces, nonadiabatic couplings, dipole moments) are reduced via the Frobenius norm. An overview of these property-space visualization strategies is provided in Figure S11.

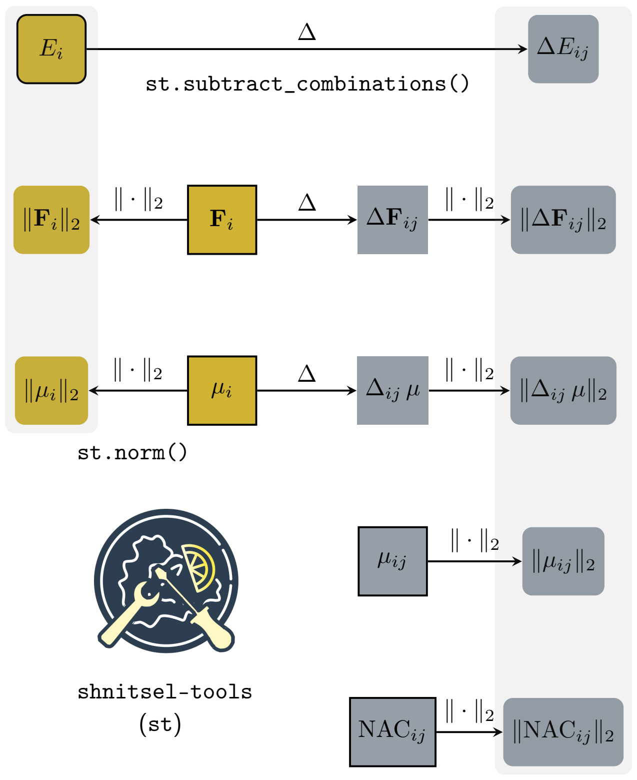
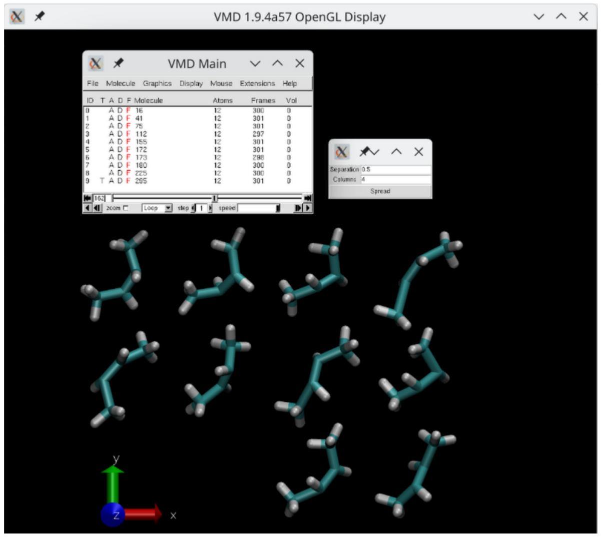


Figure S11: Overview of the properties stored in a dataset generated with shnitsel-tools (st) and their relationships for visualization purposes. Quantities in the leftmost column depend only on frame and state indices and can be directly plotted against each other. Those in the rightmost column depend on frame and state combinations and can likewise be plotted together. The middle columns contain vectorial/tensorial quantities, which are reduced *via* the 2-norm ($||\cdot||_2$) prior to visualization. *Per-state* and *inter-state* quantities are highlighted in yellow and blue, respectively. To compare these two types, pairwise differences (Δ) with respect to electronic states can be computed. Properties directly available in a st-parsed dataset are indicated by black frames.

- **Viewing 3D geometries**

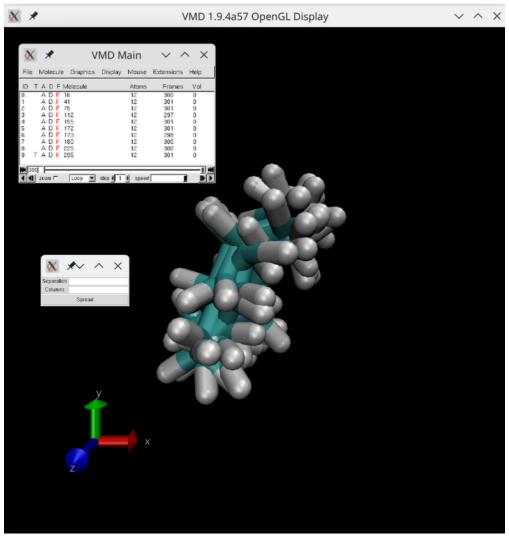
Most NAMD packages generate trajectory files (*e.g.*, `.xyz`) that can be directly opened in standard molecular viewers. Within an interactive Python environment, `st` provides equivalent functionality *via* `py3Dmol` and VMD [13], enabling direct visualization of selected subsets of the data without exporting intermediate files. `py3Dmol` renders individual structures or trajectories directly in the notebook, either singly or in a grid layout. VMD, which is installed separately, offers more advanced visualization and analysis capabilities and may perform better for large datasets. When multiple trajectories are loaded into VMD, they are initially overlaid; a dedicated extension is provided to arrange them in a grid for easier comparison (see **Figure S12**).

```python
import shnitsel as st
import xarray as xr
import shnitsel.xarray
from shnitsel.vis.vmd import traj_vmd

# load and clean data
dt_A03 = st.io.read('data/tree/stacked/A03_butene_stacked.nc').set_charge(0).set_compound_info('A03')
dt_A03_clean = st.clean.sanity_check(dt_A03, geometry_thresholds={"[*]~[*]": 2.25, "[#6,#7][#1]":1.5})
ds_A03 = dt_A03_clean['A03/1/data']

# visualized cleaned dataset in VMD
traj_vmd(ds_A03.atXYZ)
```


VMD 1.9.4a57 OpenGL Display
VMD Main
File Molecule Graphics Display Mouse Extensions Help
zoom
Loop
step
speed
Separation 0.5
Columns 4
Spread
y
x
z

ID	T	A	D	F	Molecule	Atoms	Frames	Vol
0		A	D	F	16	12	300	0
1		A	D	F	41	12	301	0
2		A	D	F	75	12	301	0
3		A	D	F	112	12	297	0
4		A	D	F	155	12	301	0
5		A	D	F	172	12	301	0
6		A	D	F	173	12	298	0
7		A	D	F	180	12	300	0
8		A	D	F	225	12	300	0
9	T	A	D	F	295	12	301	0

(a) grid


VMD 1.9.4a57 OpenGL Display
VMD Main
File Molecule Graphics Display Mouse Extensions Help
zoom
Loop
step
speed
Separation
Columns
Spread
y
x
z

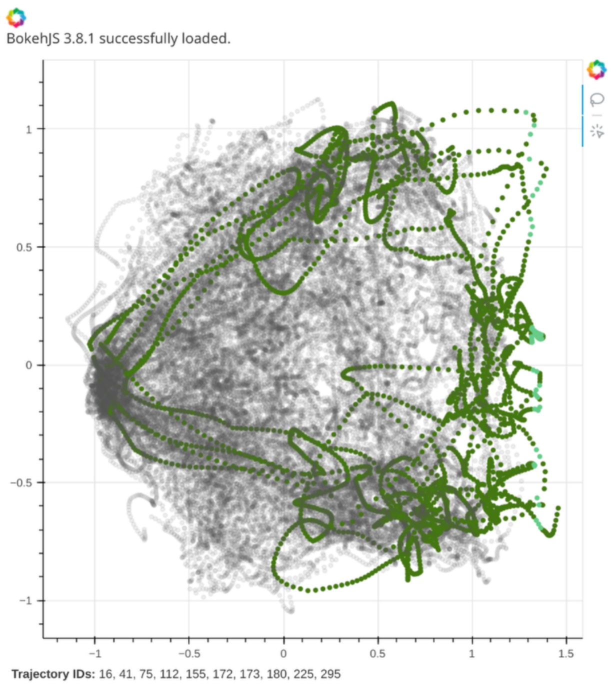
ID	T	A	D	F	Molecule	Atoms	Frames	Vol
0		A	D	F	16	12	300	0
1		A	D	F	41	12	301	0
2		A	D	F	75	12	301	0
3		A	D	F	112	12	297	0
4		A	D	F	155	12	301	0
5		A	D	F	172	12	301	0
6		A	D	F	173	12	298	0
7		A	D	F	180	12	300	0
8		A	D	F	225	12	300	0
9	T	A	D	F	295	12	301	0

(b) overlay

Figure S12: Visualization of surface-hopping trajectories using the `traj_vmd` module. Screenshots show example outputs rendered with VMD in grid-view (a) or as overlay (b), illustrating the rendering of selected trajectory subsets.

• Interactive Visualizations

When frames have been represented as points in 2D space by dimensional reduction or otherwise, it may be of interest to select outliers or otherwise notable subsets for inspection. When working in a Jupyter Notebook or similar, the points can be shown in an interactive widget which allows them to be selected using the mouse; the selection is immediately communicated back to the Jupyter kernel and is then accessible from Python. This workflow is supported by functions in the `st.plot.vis.select` module: `FrameSel` and `TrajSel`. These differ only in their interactive behaviour: in `TrajSel`, when one frame is selected, all others in the same trajectory are highlighted as well, and trajectory IDs of selected frames are displayed below the plot. Both are implemented using the Bokeh library.


BokehJS 3.8.1 successfully loaded.

Trajectory IDs: 16, 41, 75, 112, 155, 172, 173, 180, 225, 295

Figure S13: Screenshot of interactive trajectory selection widget for butene (**A03**) as generated by means of `ds.atXYZ.st.pca().results.st.TrajSelector()`, where `ds` refers to a stacked dataset. Frames selected by the user using the lasso tool are highlighted in light green, while the trajectories to which those frames belong are highlighted in dark green. A live-updating list of selected trajectories is displayed at the bottom of the widget.

- **Electronic absorption spectra (Ski plot)**

```python
import shnitsel as st
import xarray as xr
import shnitsel.xarray
import matplotlib.pyplot as plt

# load and clean data
dt_A03 = st.io.read('data/tree/stacked/A03_butene_stacked.nc').set_charge(0).set_compound_info('A03')
dt_A03_clean = st.clean.sanity_check(dt_A03, geometry_thresholds={"[*]~[*]": 2.25, "[#6,#7][#1]":1.5})
ds_A03 = dt_A03_clean['A03/1/data']

# get inter state properties
inter_state = ds_A03.st.get_inter_state()
inter_state = inter_state.unstack('frame')

# calculate spectra
spectra = st.analyze.spectra.apply_gauss_broadening(inter_state.energy_interstate, inter_state.fosc, agg_dim='
    atrajectory')

# plot spectra
st.vis.plot.spectra3d.ski_plots(spectra, threshold=0.6)
```

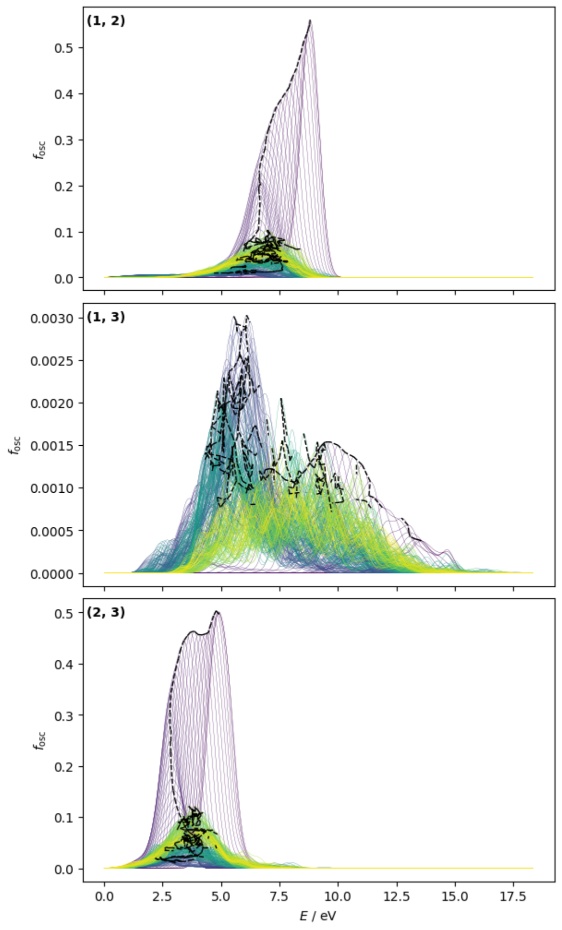


Figure S14: Visualization of the evolution of ground-state (top: $S_0 \rightarrow S_1$, middle: $S_0 \rightarrow S_2$) and excited-state absorption features (bottom: $S_1 \rightarrow S_2$) during the non-adiabatic molecular dynamics simulation of **A03** as obtained with the `ski_plots` function. At each time step, the spectrum represents the average over all trajectories. The black dashed line traces the peak of the absorption band over time. The data is taken from the `shnitsel-tools` tree-format dataset [3].

- **Custom plotting**

Beyond the convenient presets offered by `shnitsel-tools`, there are many ways to visualize data. `DataArrays` and `Datasets` can be consumed by Matplotlib directly (the latter *via* the `plt.scatter(data=ds, x='var1', y='var2')` syntax) or converted to Pandas `DataFrames` for consumption by `Seaborn`. `Xarray` has a built-in `.plot` accessor which can produce line-plots, scatter-plots, 1D and 2D histograms, contour-plots, quiver-plots and 3D surfaces. Its methods make use of metadata in the objects, use appropriate defaults and produce `Matplotlib` objects that can be combined and modified.

Each function is accessed through the plot accessor, *e.g.* `ds.plot.scatter(...)`, and wraps the Matplotlib function with the same name, *e.g.* `matplotlib.pyplot.scatter(...)`. `DataArrays` can be plotted by indicating the role different array dimensions should play in the plot. `Datasets` can be plotted by specifying `data_vars` to be plotted against each other. If there are more dimensions to visualize than can be represented in a single plot, a `FacetGrid` similar to that used in `Seaborn` is generated. An overview of this built-in plotting functionality is given in **Table S5**.

Table S5: Overview of plotting built-in functions provided by `DataArrays` and `Datasets` (adapted from `https://docs.xarray.dev/en/latest/api/plotting.html`).

Plot type	DataArray support	Dataset support
contourf	✓	
contour	✓	
hist	✓	
imshow	✓	
line	✓	
pcolormesh	✓	
quiver		✓
scatter	✓	✓
step	✓	
streamplot		✓
surface	✓	

Xarray is one of many data sources supported by the hvPlot [14] package, which offers a grammar of graphics to combine and connect visualizations. Using its sister-library HoloViews [15], it can output visualizations using Matplotlib, Bokeh and Plotly. The latter two are Javascript-based and offer rich interactivity for in-notebook use or HTML export. The UltraPlot [16] project (descended from ProPlot), which offers a thin layer of usability improvements over the Matplotlib interface, also professes explicit support for Xarray objects.

4 Showcase Example

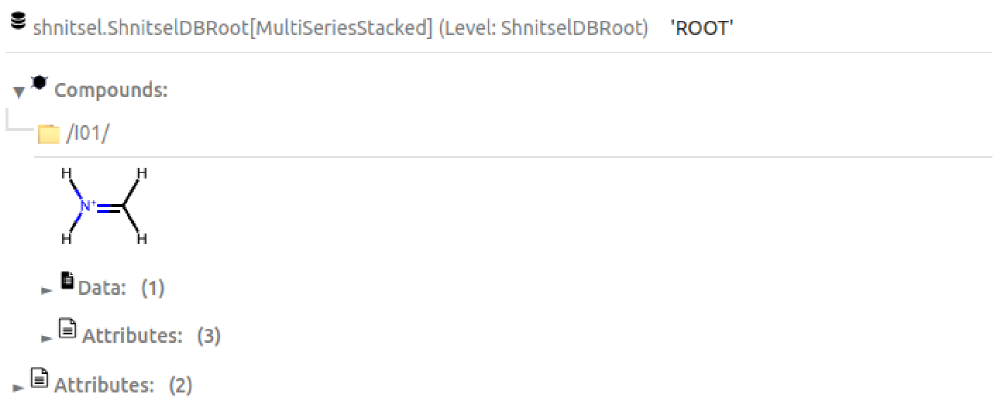
This section documents the computational workflow used to generate the cross-compound analysis for ethene (**A01**), propene (**A02**), *Z*-but-2-ene (**A03**), and the methylenimmonium cation (**I01**) presented in Section 3 of the main manuscript. The corresponding code blocks reproduce each step of the workflow, including data loading, structure selection, geometric descriptor calculation, and principal component analysis (PCA). The descriptions below summarize the purpose of each step and its role in the analysis; code snippets are provided to enable full reproducibility.

Note, the data of the tutorial is described in [5] and available on Zenodo [3] and not provided in the `shnitsel-tools` Github repository.

Data loading and organization

Trajectory data of **A01**, **A02**, **A03** and **I01** were imported into a single `DataTree` structure. Each compound was assigned to its own branch, allowing subsequent operations to be applied either per compound or across the full ensemble in a consistent manner. This hierarchical representation enables cross-compound comparisons while preserving access to the original trajectory and metadata for each system.

```python
from pathlib import Path
import shnitsel as st
import xarray as xr
import shnitsel.xarray
from shnitsel.data.tree.support_functions import tree_merge

alkenes_sources = {'A01': ('data/tree/A01_ethene_stacked.nc', 0),
                   'A02': ('data/tree/A02_propene_stacked.nc', 0),
                   'A03': ('data/tree/A03_butene_stacked.nc', 0),
                   'I01': ('data/tree/I01_ch2nh2_stacked.nc', 1),
                   }

all_dt_in = []
for name , (path, charge) in alkenes_sources.items():
    dt_in = st.io.read(path=path).set_charge(charge).set_compound_info(name)
    all_dt_in.append(dt_in)

all_dt_in[3]
```



```
shnitsel.ShnitselDBRoot[MultiSeriesStacked] (Level: ShnitselDBRoot)    'ROOT'

▼ Compounds:
  └─ /I01/

     ► Data:  (1)
     ► Attributes:  (3)
► Attributes:  (2)
```

Sanity Check

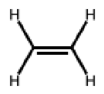
Prior to analysis, all trajectories were subjected to standard energetic and geometric sanity checks using the `clean` module. Energetic thresholds were applied to remove frames affected by numerical instabilities, and geometries exceeding a bond-length threshold of 200 pm were flagged to exclude non-physical dissociation events. This step ensures that only physically meaningful trajectory segments contribute to the statistical analysis.

```python
from shnitsel.clean import sanity_check

all_dt_filtered = []
for dt in all_dt_in:
    all_dt_filtered.append(sanity_check(dt, geometry_thresholds={"[*]~[*]": 2.25, "[#6,#7][#1]":1.5}))

mc_dt_filtered = tree_merge(*all_dt_filtered)
mc_dt_filtered
```

shnitsel.ShnitselDBRoot[Frames] (Level: ShnitselDBRoot) 'ROOT'

▼ Compounds:

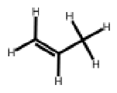
/A01/



► Data: (1)

► Attributes: (3)

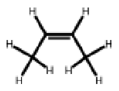
/A02/



► Data: (1)

► Attributes: (3)

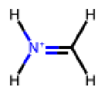
/A03/



► Data: (1)

► Attributes: (3)

/I01/



► Data: (1)

► Attributes: (3)

Common substructure selection

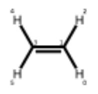
To enable direct geometric comparison between molecules with different atom counts, a common chromophoric substructure was defined using the SMARTS pattern `[#6,#1][#6,#7]([#1])=[#6]([#1])[#6,#1]`. This pattern identifies the central double-bond unit shared by all four molecules, and the two atoms attached to it (either C or H atom). The `StructureSelection` class was used to map this pattern onto each compound and to restrict all coordinate-dependent properties to the resulting four-atom fragment. This produces a canonicalized, atom-aligned representation across all systems.

```python
from shnitsel.bridges import to_mol
from shnitsel.geo.analogs import extract_analogs

smarts_share = '[#6,#1][#6,#7]([#1])=[#6]([#1])[#6,#1]'
analogs_tree = extract_analogs(mc_dt_filtered, smarts=smarts_share)
analogs_tree
```

shnitsel.ShnitselDBRoot[MultiSeriesStacked] (Level: ShnitselDBRoot) 'ROOT'

▼ Compounds:

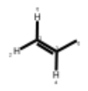
/A01/



► Data: (1)

► Attributes: (3)

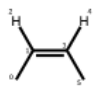
/A02/



► Data: (1)

► Attributes: (3)

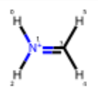
/A03/



► Data: (1)

► Attributes: (3)

/I01/



► Data: (1)

► Attributes: (3)

Geometric descriptors

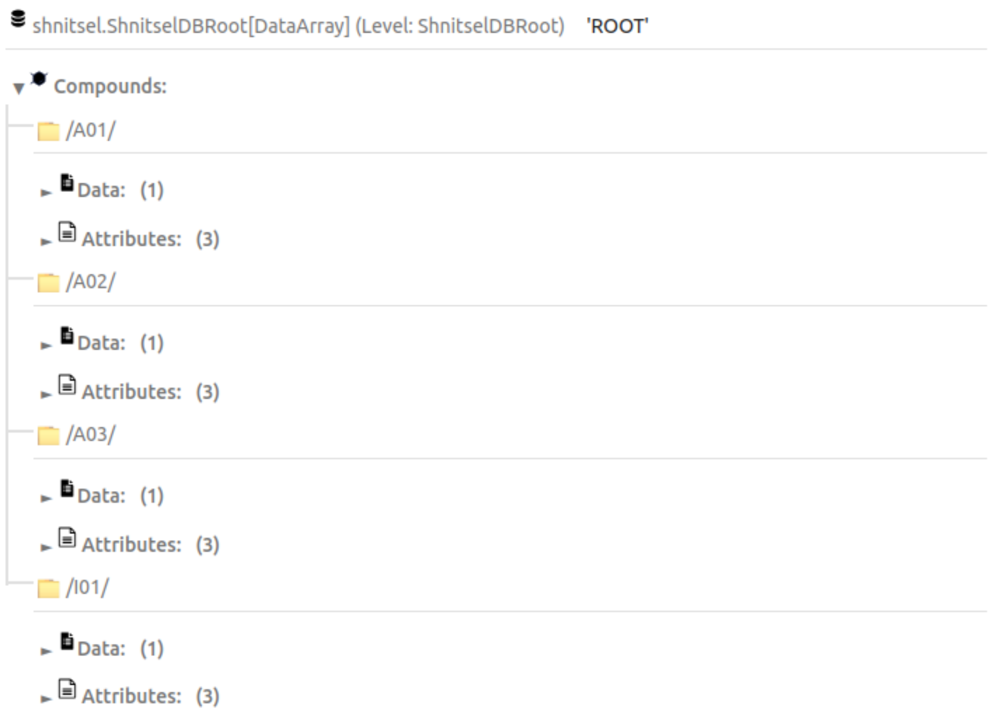
For the selected substructures, internal coordinates and pairwise atomic distances were computed using the `geocalc` module. These descriptors capture the key nuclear motions associated with torsion and pyramidalization around the central double bond and serve as input for the subsequent dimensionality-reduction analysis.

```python
from shnitsel.analyze.generic import pwdists
from shnitsel.geo.geocalc import get_bats

# Calculate pairwise distances
dt_analog_pwdists = analogs_tree.map_data(pwdists)

# Calculate all BATs for all analogs:
selection_keys = ['bonds', 'angles', 'dihedrals']
dt_analog_bats_all = get_bats(analogs_tree, selection_keys)
```

Principal component analysis

Principal component analysis (PCA) was performed using the pairwise distances of the **A03** substructure as reference descriptors. The PCA basis obtained from **A03** was then used to project the trajectories of all four compounds into a common low-dimensional space. This enables direct comparison of the sampled configurational regions and the localization of electronic state-hopping events within the same geometric framework.

```python
from shnitsel.analyze.pca import PCA
from shnitsel.vis.plot import biplot_kde

# Calculate the PCA for all analogs
selection_keys = None
pca_res = PCA(analogs_tree, structure_selection=selection_keys)

# Select reference PCA and project all others on this space
reference_pca = pca_res["A03/pca"].data
mc_projected_features = dt_analog_pwdists.map_data(reference_pca.project_array)
mc_projected_features
```

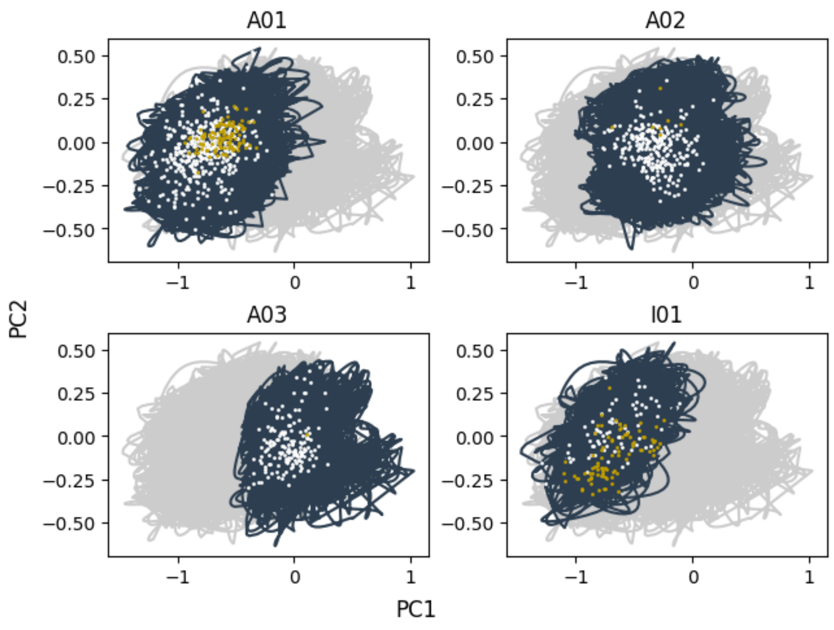


```
shnitsel.ShnitselDBRoot[DataArray] (Level: ShnitselDBRoot)   'ROOT'

▼ Compounds:
  /A01/
    ► Data:  (1)
    ► Attributes:  (3)
  /A02/
    ► Data:  (1)
    ► Attributes:  (3)
  /A03/
    ► Data:  (1)
    ► Attributes:  (3)
  /I01/
    ► Data:  (1)
    ► Attributes:  (3)
```

Comparison of the geometry space

Upon projecting all pairwise distances of the analog molecules onto the PCs of **A03**, we plot the space covered by all molecules in gray and the space visited in the photoinduced dynamics of each single compound in blue. Additionally, we show the directed last hopping points in yellow ($S_2 \rightarrow S_1$) and white ($S_1 \rightarrow S_0$), which can be obtained by the hops module.

```python
import matplotlib.pyplot as plt
from shnitsel.vis.colormaps import st_blue, st_grey, st_yellow, st_violet, st_orange
from shnitsel.analyze.hops import filter_data_at_hops

compound_colors = {"A01": st_grey, "A02": st_grey, "A03": st_grey, "I01": st_grey}

fig, axs = plt.subplot_mosaic([['A01', 'A02'],['A03', 'I01'],], layout='constrained')
for ax_name, ax in axs.items():
    ax.set_title(ax_name)
    for compound_name, data in mc_projected_features.compounds.items():
        stacked_data =  data.as_stacked
        ax.plot(stacked_data.isel(PC=0), stacked_data.isel(PC=1), c='#ccc', rasterized=True)

    compound_data = mc_projected_features[ax_name].as_stacked
    compound_color = compound_colors[ax_name]
    ax.plot(compound_data.isel(PC=0), compound_data.isel(PC=1), c=compound_color, rasterized=True)

    hops_positions_21 = filter_data_at_hops(compound_data, "2->1")
    hops_positions_32 = filter_data_at_hops(compound_data, "3->2")
    ax.scatter(hops_positions_21.isel(PC=0), hops_positions_21.isel(PC=1), c='white', s=1, zorder=10,
     rasterized=True)
    ax.scatter(hops_positions_32.isel(PC=0), hops_positions_32.isel(PC=1), c=st_yellow, s=1, zorder=10,
     rasterized=True)

fig.supxlabel('PC1')
fig.supylabel('PC2')
```



References

1. SHNITSEL-TOOLS 2026. doi:https://github.com/CompPhotoChem/shnitsel-tools. https://github.com/CompPhotoChem/shnitsel-tools.
2. Hoellring, K., Roehrkasten, T., Müller, C. & Group, C. *CompPhotoChem/shnitsel-tools: v2026.01.2* version v2026.01.2. Mar. 2026. doi:10.5281/zenodo.19135766. https://doi.org/10.5281/zenodo.19135766.
3. Höllring, K., Röhrkasten, T. & Müller, C. *SHNITSEL - Surface Hopping Nested Instances Training Set for Excited-state Learning* 2026. doi:10.5281/zenodo.18436276. https://doi.org/10.5281/zenodo.18436276.
4. Hoyer, S. & Hamman, J. xarray: N-D labeled arrays and datasets in Python. *J. Open Res. Softw.* **5,** 1–10. doi:10.5334/jors.148. https://doi.org/10.5334/jors.148 (2017).
5. Curth, R., Röhrkasten, T. E., Müller, C. & Westermayr, J. Surface Hopping Nested Instances Training Set for Excited-state Learning. *Scientific Data* **12,** 1300. doi:10.1038/s41597-025-05443-5. https://doi.org/10.1038/s41597-025-05443-5 (2025).
6. Mai, S., Bachmair, B., Gagliardi, L., Gallmetzer, H. G., Grünewald, L., Hennefarth, M. R., Høyer, N. M., Korsaye, F. A., Mausenberger, S., Oppel, M., Piteša, T., Polonius, S., Gil, E. S., Shu, Y., Singer, N. K., Tiefenbacher, M. X., Truhlar, D. G., Vóŕos, D., Zhang, L. & González, L. *SHARC4.0: Surface Hopping Including Arbitrary Couplings — Program Package for Non-Adiabatic Dynamics* https://sharc-md.org/. 2025. doi:10.5281/zenodo.15496427.
7. Mai, S., Marquetand, P. & González, L. Nonadiabatic Dynamics: The SHARC Approach. *WIREs Comput. Mol. Sci.* **8,** e1370. doi:10.1002/wcms.1370. http://dx.doi.org/10.1002/wcms.1370 (2018).
8. Barbatti, M., Granucci, G., Ruckenbauer, M., Plasser, F., Pittner, J., Persico, M. & Lischka, H. *NEWTON-X: a package for Newtonian dynamics close to the crossing seam, version 1.2* www.netwonx.org. 2011.
9. Barbatti, M., Ruckenbauer, M., Plasser, F., Pittner, J., Granucci, G., Persico, M. & Lischka, H. Newton-X: a surface-hopping program for nonadiabatic molecular dynamics. *WIREs Comput. Mol. Sci.* **4,** 26–33. doi:10.1002/wcms.1158. https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.1158 (2014).
10. Li, J., Reiser, P., Boswell, B. R., Eberhard, A., Burns, N. Z., Friederich, P. & Lopez, S. A. Automatic discovery of photoisomerization mechanisms with nanosecond machine learning photodynamics simulations. *Chem. Sci.* **12,** 5302–5314. ISSN: 2041-6520. doi:10.1039/D0SC05610C. http://dx.doi.org/10.1039/D0SC05610C (2021).
11. Li, J., Stein, R., Adrion, D. M. & Lopez, S. A. Machine-Learning Photodynamics Simulations Uncover the Role of Substituent Effects on the Photochemical Formation of Cubanes. *J. Am. Chem. Soc.* **143,** 20166–20175. ISSN: 0002-7863. doi:10.1021/jacs.1c07725. https://doi.org/10.1021/jacs.1c07725 (2021).
12. Curth, R., Röhrkasten, T., Müller, C. & Westermayr, J. *SHNITSEL - Surface Hopping Nested Instances Training Set for Excited-state Learning* May 2025. doi:10.5281/zenodo.15482819. https://doi.org/10.5281/zenodo.15482819.
13. Humphrey, W., Dalke, A. & Schulten, K. VMD: Visual molecular dynamics. *Journal of Molecular Graphics* **14,** 33–38. ISSN: 0263-7855. doi:https://doi.org/10.1016/0263-7855(96)00018-5. https://www.sciencedirect.com/science/article/pii/0263785596000185 (1996).
14. *hvPlot — hvPlot 0.12.2 documentation* https://hvplot.holoviz.org/en/docs/latest/index.html (2025).
15. *holoviz/holoviews: Version 1.22.1* https://zenodo.org/records/17831576 (2025).
16. *Ultraplot/UltraPlot: Zenodo release* https://zenodo.org/records/15733580 (2025).