
tesliper

Release 0.9.3

Michał M. Więclaw

Apr 04, 2022

INTRODUCTION

1	Key features	3
2	Motivation and context	5
3	References	7
3.1	Installation	7
3.2	Conventions and Terms	8
3.3	Graphical Interface	9
3.4	Scripting with <code>tesliper</code>	31
3.5	Advanced guide	41
3.6	Available data genres	50
3.7	Math and Algorithms	53
3.8	<code>tesliper</code>	59
3.9	Change Log	174
3.10	Index	179
	Python Module Index	181
	Index	183



tesliper is a package for batch processing of Gaussian output files, focusing on extraction and processing of data related to simulation of optical spectra. The software offers a Python API and a graphical user interface (GUI), allowing for your preferred style of interaction with the computer: visual or textual. It's main goal is to minimize time and manual work needed to simulate optical spectrum of investigated compound.

KEY FEATURES

tesliper was designed for working with multiple conformers of a compound, represented by a number of files obtained from Gaussian quantum-chemical computations software. It allows you easily exclude conformers that are not suitable for further analysis: erroneous, not optimized, of higher energy than a user-given threshold, or very similar to some other structure in the set. Data parsed from files and data calculated may be exported to other file formats for storage or further analysis with other tools. Below is a quick overview of features it provides:

- Batch processing of Gaussian output files regarding structure optimization and simulation of spectral properties
- Conditional, property-based filtering of conformers
- Geometry comparison via the RMSD sieve
- Calculation of Boltzmann distribution—based populations of conformers
- Simulation of IR, VCD, UV, ECD, Raman, and ROA spectra from spectral activities
- Export of extracted and calculated data to .txt, .csv, and .xlsx file formats
- Export of .gjf files for further calculations in Gaussian software
- Free & open source (OSI approved BSD 2-Clause license)
- Graphical and programmatic interfaces

MOTIVATION AND CONTEXT

Simulation of optical spectra of organic compounds becomes one of the routine tasks for chemical analysts – it is a necessary step in one of the increasingly popular methods of establishing compound's absolute configuration. However, the process of obtaining a simulated spectrum may be cumbersome, as it usually involves analyzing a large number of potentially stable conformers of the studied molecule. **tesliper** was created to aid in such work.

It should be noted that **tesliper** is not the only software that is capable of providing a simulated spectrum, given output of quantum-chemical computations. The table below summarizes other available GUI tools and compares features they offer. Among listed **tesliper** is the only one that is open source and allows to easily filter parsed data.

Table 1: How does **tesliper** fit into the market?

	Tes- liper	SpecDis ¹	CD- specTech ²	Com- puteVOA ³	GaussView ⁴	Chem- Craft ⁵
Free	✓	✓	✓			
Open Source	✓					
Batch Processing	✓	✓	✓		.gjf export	.gjf modif.
Geometry Compari- son	✓			✓		✓
Averaging	✓	✓	✓			
Conditional Filtering	✓					
Job File Creation	✓			✓	✓	✓
Electronic Spectra	✓	✓	✓		✓	
Scattering Spectra	✓		✓	✓	✓	✓
Multi-platform	✓	✓	✓		✓	needs wine
Conformational Search				✓	optional	
Molecule Visualiza- tion				✓	✓	✓

¹ **SpecDis**: T. Bruhn, A. Schaumlöffel, Y. Hemberger, G. Pescitelli, *SpecDis version 1.71*, Berlin, Germany, **2017**, <http://specdis-software.jimdo.com>

² **CDspecTech**: C. L. Covington, P. L. Polavarapu, *Chirality*, **2017**, 29, 5, p. 178, DOI: 10.1002/chir.22691

³ **ComputeVOA**: E. Debie, P. Bultinck, L. A. Nafie, R. K. Dukor, BioTools Inc., Jupiter, FL, **2010**, <https://biotools.us/software-2>

⁴ **GaussView**: R. Dennington, T. A. Keith, J. M. Millam, Semichem Inc., *GaussView version 6.1*, Shawnee Mission, KS, **2016**

⁵ **ChemCraft**: <https://www.chemcraftprog.com>

REFERENCES

3.1 Installation

3.1.1 GUI for Windows

For Windows users that only intend to use graphical interface, `tesliper` is available as a standalone .exe application, available for download from the [latest release](#) under the **Assets** section at the bottom of the page. No installation is required, just double-click the downloaded **Tesliper.exe** file to run the application.

Unfortunately, a single-file installation is not available for unix-like systems. Please follow a terminal-based installation instructions below.

3.1.2 Install from terminal

`tesliper` is a Python package [distributed via PyPI](#). You can install it to your python distribution simply by running:

```
python -m pip install tesliper
```

in your terminal. This will download and install `tesliper` along with it's essential dependencies. A graphical interface have an additional dependency, but it may be easily included in your installation if you use `python -m pip install tesliper[gui]` instead. Some users of unix-like systems may also need to instal `tkinter` manually, if it is not included in their distribution by default. Please refer to relevant online resources on how to do this in your system, if that is your case.

Note: Reminder for zsh users to quote the braces like this `'tesliper[gui]'` or like this `tesliper\[gui]` when installing extras. This is necessary, because normally zsh uses `some[thing]` syntax for pattern matching.

3.1.3 Requirements

This software needs at least Python 3.6 to run. It also uses some additional packages:

```
numpy  
openpyxl  
matplotlib (optional, for GUI)
```

Note: `tesliper` uses `tkinter` to deliver the graphical interface. It is included in most Python distributions, but please be aware, that some might miss it. You will need to install it manually in such case.

3.2 Conventions and Terms

3.2.1 Reading and writing

tesliper was designed to deal with multiple conformers of a single molecule. It identifies conformers using a stem of an extracted file (i.e. its filename without extension). When files with identical names (save extension) are extracted in course of subsequent `Tesliper.extract()` calls (or in recursive extraction, see method's documentation), they are treated as the same conformer. This enables to join data from subsequent calculations steps, e.g. geometry optimization, vibrational spectra simulation, and electronic spectra simulation.

Note: If specific data genre is available from more than one file, only recently extracted values will be stored.

Also, writing extracted and calculated data to files is done in batch, as usually multiple files are produced. Hence, tesliper will chose names for these files automatically, only allowing to specify output directory (as `Tesliper.output_dir` attribute). If you need more control over this process, you will need to use one of the writer objects directly. These are easily available *via* the `writer_base.writer()` factory function.

3.2.2 Handling data

tesliper stores multiple data entries of various types for each conformer. To prevent confusion with Python's data type and with data itself, tesliper refers to specific kinds of data as *genres*. Genres in code are represented by specific strings, used as identifiers. To learn about data genres known to tesliper, see documentation for `GaussianParser`, which lists them.

Note: Given the above, you may wonder why is it *genres* and not just *kinds* of data then? The reason is that naming things is hard (one of the only two hard things in Computer Science, as [Phil Karlton said](#)). As of time of deciding on this name, I did not come up with the second one. Hopefully, this small oddity will not bother you too much.

tesliper may not work properly when used to process data concerning different molecules (i.e. having different number of atoms, different number of degrees of freedom, etc.). If you want to use it for such purpose anyway, you may set `Tesliper.conformers.allow_data_inconsistency` to `True`. tesliper will then stop complaining and try to do its best.

3.2.3 Glossary

genre A specific kind of data, e.g. SCF energy, dipole strengths, atoms' positions in space, or command used for calculations. Represented in code by a short string. Not to be confused with Python's data type. See [Available data genres](#).

trimming Internally marking certain conformers as *not kept*. tesliper provides an easy way to trim conformers to user's needs, see [Filtering conformers](#).

kept Conformers may be internally marked as *kept* or *not kept*. *Kept* conformers will be normally processed by tesliper, *not kept* conformers will be ignored. See [Conformers.kept](#).

arrayed About data turned into an instance of `DataArray`-like object, usually by `Conformers'` method of the same name. See [Conformers.arrayed\(\)](#).

data array Type of objects used by tesliper to handle data read from multiple conformers. The same data array class may be used to represent more than one genre. Sometimes referred to as `DataArray`-like classes or objects. See [arrays](#).

data inconsistency An event of data having non-uniform properties, e.g. when number of values doesn't match number of conformers, or when some conformers provide a different number of values than other conformers for a particular data genre. See [array_base](#).

3.3 Graphical Interface

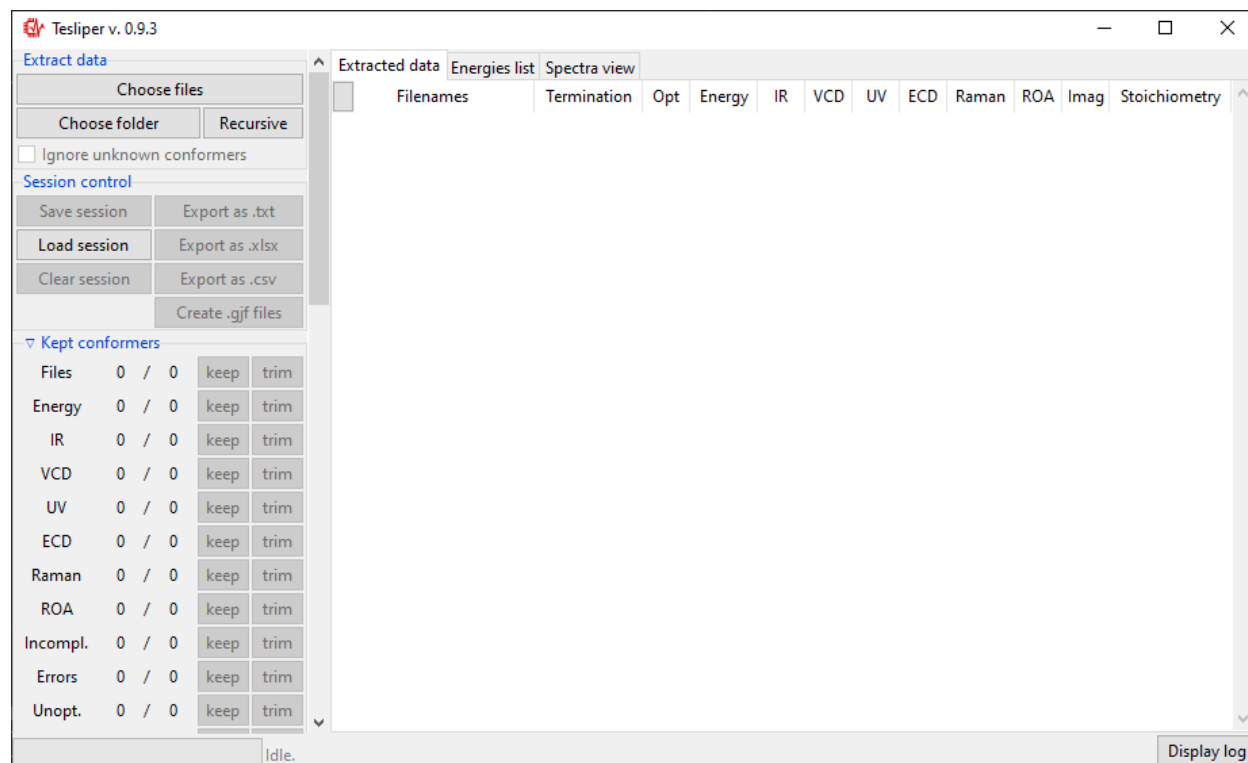
This part discusses the use of the Graphical User Interface (GUI). For tutorial on using `tesliper` in Python scripts, see tutorial.

On Windows system you may start the GUI by downloading and double-clicking `Tesliper.exe` file available in the [latest release](#), as described in the [Installation section](#). Executable files are not available for other systems, unfortunately, but you may start the GUI from the command line as well:

```
$ python -m pip install tesliper[gui] # only once
$ tesliper-gui # starts GUI
```

Note: If you'd like to start the graphical interface from the local copy, you may also run it as a module with `python -m tesliper.gui`.

Please note that the first launch may take additional time. After the application starts, a window like the one bellow will appear. It's actual looks will depend on your operating system.



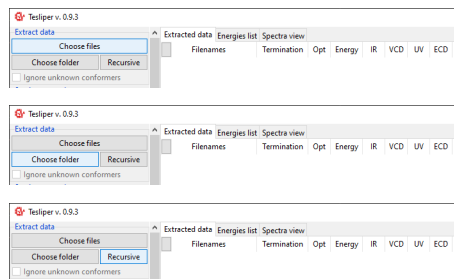
The Interface is divided in two parts: controls on the left and views (initially empty) on the right. Controls panel are further divided into sections, some of which may be collapsed by clicking on the section title (those with a small arrow on the left). Each section will be described further in this tutorial, in the appropriate section.

There are tree views available: `Extracted data` and `Energies list` summarize all conformers read from files. `Extracted data` details what data is available and shows status of calculations for each conformer. `Energies list`

shows values of conformers' energies calculated by quantum chemical software and derived values: Boltzmann factors and conformers' populations.

3.3.1 Reading files

tesliper supports reading data from computations performed using Gaussian software. To load data, use controls in the **Extract data** section. **Choose files** button allows you to select individual files to read using the popup dialog. **Choose folder** button shows a similar dialog, but allowing you to select a single directory - all Gaussian output files in this directory (but not subdirectories) will be read. Finally, using the **Recursive** button will also read files from all subdirectories, recursively.



Note: Make sure you do not have mixed .log and .out files in the directory, when using **Choose folder** button.

Once you select files or directory and confirm your selection, the process of data extraction will start and **Extract data** view will be updated for each read conformer. It will show if calculation job terminated normally (**Termination**), if conformer's structure was optimized and if optimization was successful (**Opt**), if extended set of energies is available (**Energy**), what spectral data it available (**IR**, **VCD**, **UV**, **ECD**, **Raman**, **ROA**), how many imaginary frequencies are reported for conformer (**Imag**), and what is conformer's stoichiometry (**Stoichiometry**).

Extract data

Choose files

Choose folder **Recursive**

☐ Ignore unknown conformers

Session control

Save session **Export as .txt**

Load session **Export as .xlsx**

Clear session **Export as .csv**

Create .gjf files

Kept conformers

Files	Energy	IR	VCD	UV	ECD	Raman	ROA	Incompl.	Errors	Unopt.
0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
keep	trim	keep	trim	keep	trim	keep	trim	keep	trim	keep

FileNames	Termination	Opt	Energy	IR	VCD	UV	ECD	Raman	ROA	Imag	Stoichiometry
<input checked="" type="checkbox"/> Tolbutamid_c1	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c10	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c11	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c12	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c13	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c14	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c15	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c16	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c17	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c18	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c19	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c2	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c20	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c21	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c22	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c23	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S
<input checked="" type="checkbox"/> Tolbutamid_c24	normal	n/a	X	X	X	ok	ok	X	X	X	C12H18N2O3S

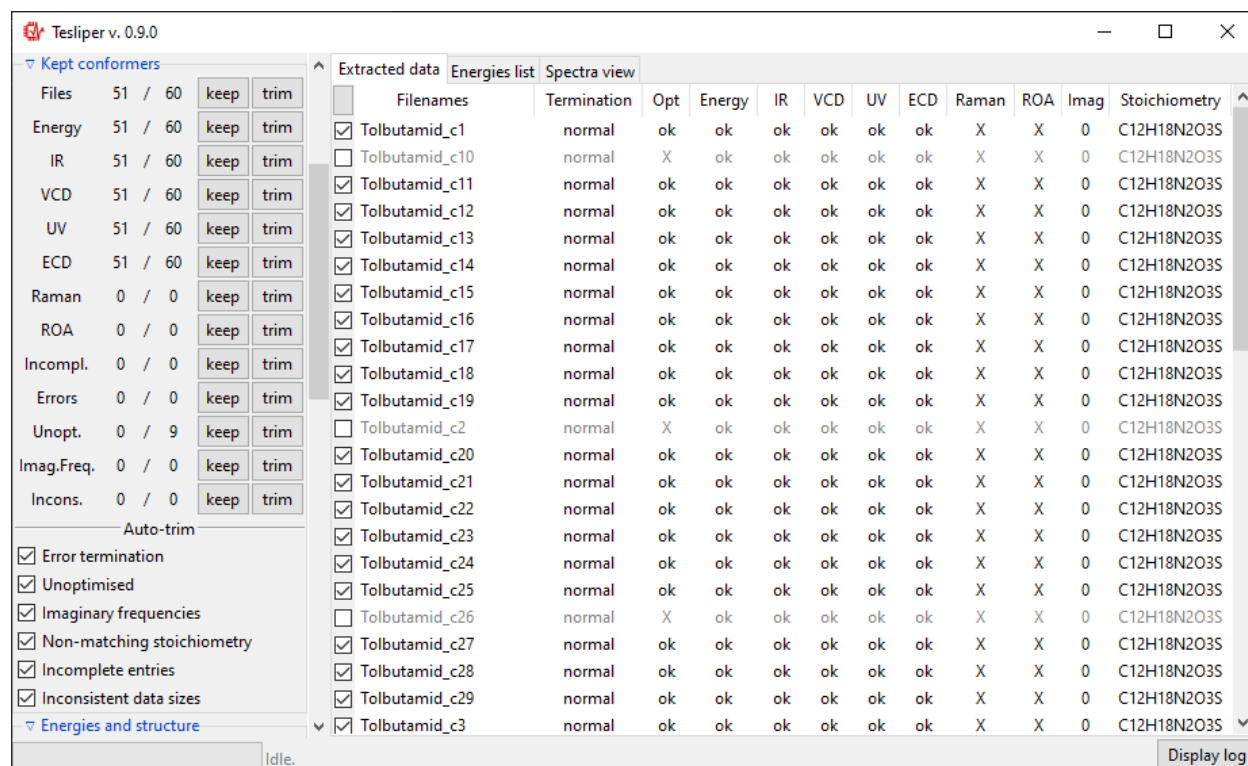
Extracting Tolbutamid_c25...

Display log

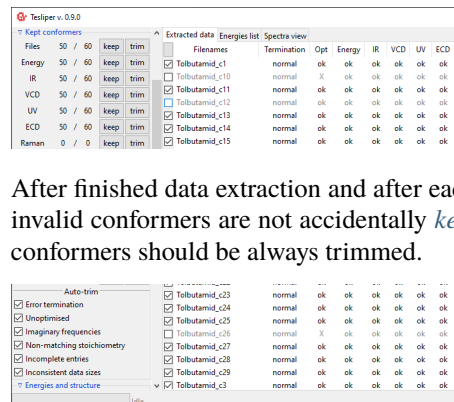
When data extraction is finished, the status barr att the bottom will show Idle again. After reading first portion of files, you may tick the Ignore unknown conformers option. When this option is ticked, tesliper will only read files that correspond to conformers it already knows (judging by the filename).

3.3.2 Trimming conformers

Conformers may be marked as *kept not kept (trimmed)*. Only *kept* conformers are processed by tesliper, *trimmed* ones are ignored. This mechanism allows you to select which conformers should be included in the final averaged spectrum, etc. Trimmed conformers are shown in gray.



Kept conformers section shows how many conformers contain certain data and allows to easily keep/trim whole groups of conformers, using **keep** and **trim** buttons beside the appropriate group. You may also keep/trim individual conformers by ticking/unticking checkboxes beside the conformers name (left of **FileNames** column).



After finished data extraction and after each manual trimming, auto-trimming is performed to make sure corrupted or invalid conformers are not accidentally *kept*. Checkboxes in the **Auto-trim** subsection, shown below, control which conformers should be always trimmed.

Tip: Incomplete entries are conformers that miss some data, which other conformers include, e.g. those that were

left out in one of calculations steps. **Inconsistent data sizes** indicates that some multi-value data has different number of data points than in case of other conformers. This usually suggests that conformer in question is not actually a conformer but a different molecule.

3.3.3 Trimming with sieves

The **Energies** and **structure** section, described in this part, is related with the **Energies list** view. This view shows, as the name suggests, list of energies for each conformer and energies-derived values.

FileNames	Thermal	Enthalpy	Gibbs	SCF	Zero-Point
<input checked="" type="checkbox"/> Tolbutamid_c1	-1201.538576	-1201.537631	-1201.609674	-1201.853442	-1201.557778
<input type="checkbox"/> Tolbutamid_c10	-1201.537206	-1201.536262	-1201.607415	-1201.852221	-1201.556326
<input type="checkbox"/> Tolbutamid_c11	-1201.537670	-1201.536726	-1201.607995	-1201.852705	-1201.556737
<input checked="" type="checkbox"/> Tolbutamid_c12	-1201.537675	-1201.536730	-1201.607953	-1201.852705	-1201.556737
<input checked="" type="checkbox"/> Tolbutamid_c13	-1201.539802	-1201.538857	-1201.610723	-1201.854742	-1201.559069
<input checked="" type="checkbox"/> Tolbutamid_c14	-1201.539789	-1201.538845	-1201.610314	-1201.854750	-1201.559026
<input checked="" type="checkbox"/> Tolbutamid_c15	-1201.538886	-1201.537942	-1201.609522	-1201.853770	-1201.558057
<input checked="" type="checkbox"/> Tolbutamid_c16	-1201.538904	-1201.537960	-1201.609544	-1201.853769	-1201.558079
<input checked="" type="checkbox"/> Tolbutamid_c17	-1201.537827	-1201.536882	-1201.608186	-1201.852887	-1201.556918
<input checked="" type="checkbox"/> Tolbutamid_c18	-1201.537840	-1201.536896	-1201.608028	-1201.852882	-1201.556931
<input checked="" type="checkbox"/> Tolbutamid_c19	-1201.537853	-1201.536909	-1201.608894	-1201.852861	-1201.557091
<input type="checkbox"/> Tolbutamid_c2	-1201.538573	-1201.537628	-1201.610445	-1201.853446	-1201.557816
<input checked="" type="checkbox"/> Tolbutamid_c20	-1201.532185	-1201.531241	-1201.603439	-1201.846597	-1201.551871
<input checked="" type="checkbox"/> Tolbutamid_c21	-1201.537670	-1201.536726	-1201.607992	-1201.852705	-1201.556737
<input checked="" type="checkbox"/> Tolbutamid_c22	-1201.537799	-1201.536854	-1201.608762	-1201.852746	-1201.557047
<input checked="" type="checkbox"/> Tolbutamid_c23	-1201.532183	-1201.531239	-1201.603647	-1201.846592	-1201.551885
<input checked="" type="checkbox"/> Tolbutamid_c24	-1201.537867	-1201.536922	-1201.609509	-1201.852863	-1201.557154
<input checked="" type="checkbox"/> Tolbutamid_c25	-1201.537675	-1201.536731	-1201.607953	-1201.852705	-1201.556738
<input type="checkbox"/> Tolbutamid_c26	-1201.537818	-1201.536873	-1201.609798	-1201.852743	-1201.557143
<input checked="" type="checkbox"/> Tolbutamid_c27	-1201.532600	-1201.531655	-1201.602908	-1201.847108	-1201.552090
<input checked="" type="checkbox"/> Tolbutamid_c28	-1201.532596	-1201.531652	-1201.602799	-1201.847099	-1201.552078
<input checked="" type="checkbox"/> Tolbutamid_c29	-1201.532067	-1201.531123	-1201.603685	-1201.846462	-1201.551662
<input checked="" type="checkbox"/> Tolbutamid_c3	-1201.538598	-1201.537654	-1201.609292	-1201.853531	-1201.557801

Using a **Show:** drop-down menu you may select a different energies-derived data to show in the view. **Delta** is conformer's energy difference to the most stable (lowest-energy) conformer (in kcal/mol units), **Min. Boltzmann factor** is conformer's Boltzmann factor in respect to the most stable conformer (unitless) and **Population** is population of conformers according to the Boltzmann distribution (in percent). Original Energy values are shown in Hartree units.

Tesliper v. 0.9.1

☒ Imaginary frequencies
☒ Non-matching stoichiometry
☒ Incomplete entries
☒ Inconsistent data sizes

▾ Energies and structure
 Show: Population /%
 Use: Energy /Hartree
 Temperature: Delta /(kcal/mol)
 Min. Boltzmann factor: Population /%
 Minimum: 0.0004 %
 Maximum: 12.7235 %
 Trim to...
 RMSD sieve
 Window size: 5.0 kcal/mol
 Threshold: 1.0 angstrom
☒ Ignore hydrogen atoms
 Trim similar

Extracted data	Energies list	Spectra view	Thermal	Enthalpy	Gibbs
<input checked="" type="checkbox"/> Tolbutamid_c1			3.3930	3.3905	4.7347
<input type="checkbox"/> Tolbutamid_c10			--	--	--
<input checked="" type="checkbox"/> Tolbutamid_c11			1.2786	1.2791	0.7760
<input checked="" type="checkbox"/> Tolbutamid_c12			1.2855	1.2846	0.7416
<input checked="" type="checkbox"/> Tolbutamid_c13			12.7092	12.6999	14.6565
<input checked="" type="checkbox"/> Tolbutamid_c14			12.5324	12.5368	9.4340
<input checked="" type="checkbox"/> Tolbutamid_c15			4.7381	4.7397	4.0196
<input checked="" type="checkbox"/> Tolbutamid_c16			4.8308	4.8325	4.1160
<input checked="" type="checkbox"/> Tolbutamid_c17			1.5142	1.5131	0.9532
<input checked="" type="checkbox"/> Tolbutamid_c18			1.5356	1.5361	0.8040
<input checked="" type="checkbox"/> Tolbutamid_c19			1.5572	1.5578	2.0436
<input type="checkbox"/> Tolbutamid_c2			--	--	--
<input checked="" type="checkbox"/> Tolbutamid_c20			0.0035	0.0035	0.0057
<input checked="" type="checkbox"/> Tolbutamid_c21			1.2786	1.2791	0.7735
<input checked="" type="checkbox"/> Tolbutamid_c22			1.4692	1.4682	1.7728
<input checked="" type="checkbox"/> Tolbutamid_c23			0.0035	0.0035	0.0072

Both types of sieves provided depend on the selected value of the Use: drop-down menu. It determines, which energy values are used by the sieves. Only available energies will be shown in the list. In case their names are not intuitive enough, here is the explanation:

Thermal: sum of electronic and thermal Energies;

Enthalpy: sum of electronic and thermal Enthalpies;

Gibbs: sum of electronic and thermal Free Energies;

SCF: energy calculated with the self-consistent field method;

Zero-Point: sum of electronic and zero-point Energies.

Tesliper v. 0.9.1

☒ Imaginary frequencies
☒ Non-matching stoichiometry
☒ Incomplete entries
☒ Inconsistent data sizes

Energies and structure

Show: Population /%
 Use: Gibbs
 Temperature: Thermal
 Minimum: Gibbs
 Maximum: SCF
 Zero-Point
 Trim to...

RMSD sieve

Window size: 5.0 kcal/mol
 Threshold: 1.0 angstrom
☒ Ignore hydrogen atoms
 Trim similar

Calculate Spectra

Extracted data	Energies list	Spectra view	Thermal	Enthalpy	Gibbs
<input checked="" type="checkbox"/> Tolbutamid_c1			3.4154	3.4130	4.7413
<input type="checkbox"/> Tolbutamid_c10			--	--	--
<input checked="" type="checkbox"/> Tolbutamid_c11			1.3083	1.3088	0.8010
<input checked="" type="checkbox"/> Tolbutamid_c12			1.3153	1.3143	0.7661
<input checked="" type="checkbox"/> Tolbutamid_c13			12.5132	12.5042	14.4013
<input checked="" type="checkbox"/> Tolbutamid_c14			12.3421	12.3463	9.3384
<input checked="" type="checkbox"/> Tolbutamid_c15			4.7428	4.7445	4.0363
<input checked="" type="checkbox"/> Tolbutamid_c16			4.8341	4.8358	4.1314
<input checked="" type="checkbox"/> Tolbutamid_c17			1.5450	1.5439	0.9805
<input checked="" type="checkbox"/> Tolbutamid_c18			1.5664	1.5670	0.8294
<input checked="" type="checkbox"/> Tolbutamid_c19			1.5881	1.5887	2.0755
<input type="checkbox"/> Tolbutamid_c2			--	--	--
<input checked="" type="checkbox"/> Tolbutamid_c20			0.0039	0.0039	0.0064
<input checked="" type="checkbox"/> Tolbutamid_c21			1.3083	1.3088	0.7984
<input checked="" type="checkbox"/> Tolbutamid_c22			1.4999	1.4988	1.8047
<input checked="" type="checkbox"/> Tolbutamid_c23			0.0039	0.0039	0.0080

The Range sieve lets you to trim conformers that have a current Show: value outside of the specified range. After you fill the Minimum and Maximum fields to match your needs, click Trim to... button to perform trimming. The example below shows trimming of conformers, which Free Energy-derived population is below 1%. Please note that values in the Energies list are recalculated and Minimum and Maximum fields are updated to show real current max and min values.

Energies and structure

Show: Population /%
 Use: Gibbs
 Temperature: 298.15 K
 Range sieve
 Minimum: 1 %
 Maximum: 14.4013 %
 Trim to...

<input checked="" type="checkbox"/> Tolbutamid_c11	1.3083	1.3088	0.8010
<input checked="" type="checkbox"/> Tolbutamid_c12	1.3153	1.3143	0.7661
<input checked="" type="checkbox"/> Tolbutamid_c13	12.5132	12.5042	14.4013
<input checked="" type="checkbox"/> Tolbutamid_c14	12.3421	12.3463	9.3384
<input checked="" type="checkbox"/> Tolbutamid_c15	4.7428	4.7445	4.0363
<input checked="" type="checkbox"/> Tolbutamid_c16	4.8341	4.8358	4.1314
<input checked="" type="checkbox"/> Tolbutamid_c17	1.5450	1.5439	0.9805
<input checked="" type="checkbox"/> Tolbutamid_c18	1.5664	1.5670	0.8294
<input checked="" type="checkbox"/> Tolbutamid_c19	1.5881	1.5887	2.0755
<input type="checkbox"/> Tolbutamid_c10	--	--	--
<input type="checkbox"/> Tolbutamid_c11	--	--	--
<input type="checkbox"/> Tolbutamid_c12	--	--	--
<input checked="" type="checkbox"/> Tolbutamid_c13	13.8928	13.8829	15.3859
<input checked="" type="checkbox"/> Tolbutamid_c14	13.7028	13.7076	9.9769
<input checked="" type="checkbox"/> Tolbutamid_c15	5.2657	5.2676	4.3122
<input checked="" type="checkbox"/> Tolbutamid_c16	5.3671	5.3690	4.4139
<input type="checkbox"/> Tolbutamid_c17	--	--	--
<input type="checkbox"/> Tolbutamid_c18	--	--	--
<input checked="" type="checkbox"/> Tolbutamid_c19	1.7632	1.7630	2.2174

Energies and structure

Show: Population /%
 Use: Gibbs
 Temperature: 298.15 K
 Range sieve
 Minimum: 1.9280 %
 Maximum: 15.3859 %
 Trim to...

The RMSD Sieve lets you mathematically compare structures of conformers and trim duplicates and almost-duplicates. RMSD stands for root-mean-square deviation of atomic positions and is a conformers similarity measure. The sieve calculates the average distance between atoms of two conformers and trims the less stable (higher-energy) conformer of the two, if the resulting RMSD value is smaller than value of the Threshold field.

Calculating an RMSD value is quite resource-costly. To assure efficient trimming, each conformer is compared only with conformers inside its energy window, defined by the **Window size** field value. Conformers of energy this much higher or lower are automatically considered different.

RMSD sieve		<input type="checkbox"/> Tolbutamid_c18	--	--	--
Window size: 5.0	kcal/mol	<input checked="" type="checkbox"/> Tolbutamid_c19	1.7632	1.7639	2.2174
Threshold: 1.0	angstrom	<input type="checkbox"/> Tolbutamid_c2	--	--	--
<input checked="" type="checkbox"/> ignore hydrogen atoms		<input type="checkbox"/> Tolbutamid_c20	--	--	--
Trim similar		<input type="checkbox"/> Tolbutamid_c21	--	--	--
		<input checked="" type="checkbox"/> Tolbutamid_c22	1.6602	1.6640	1.0381

3.3.4 Temperature of the system

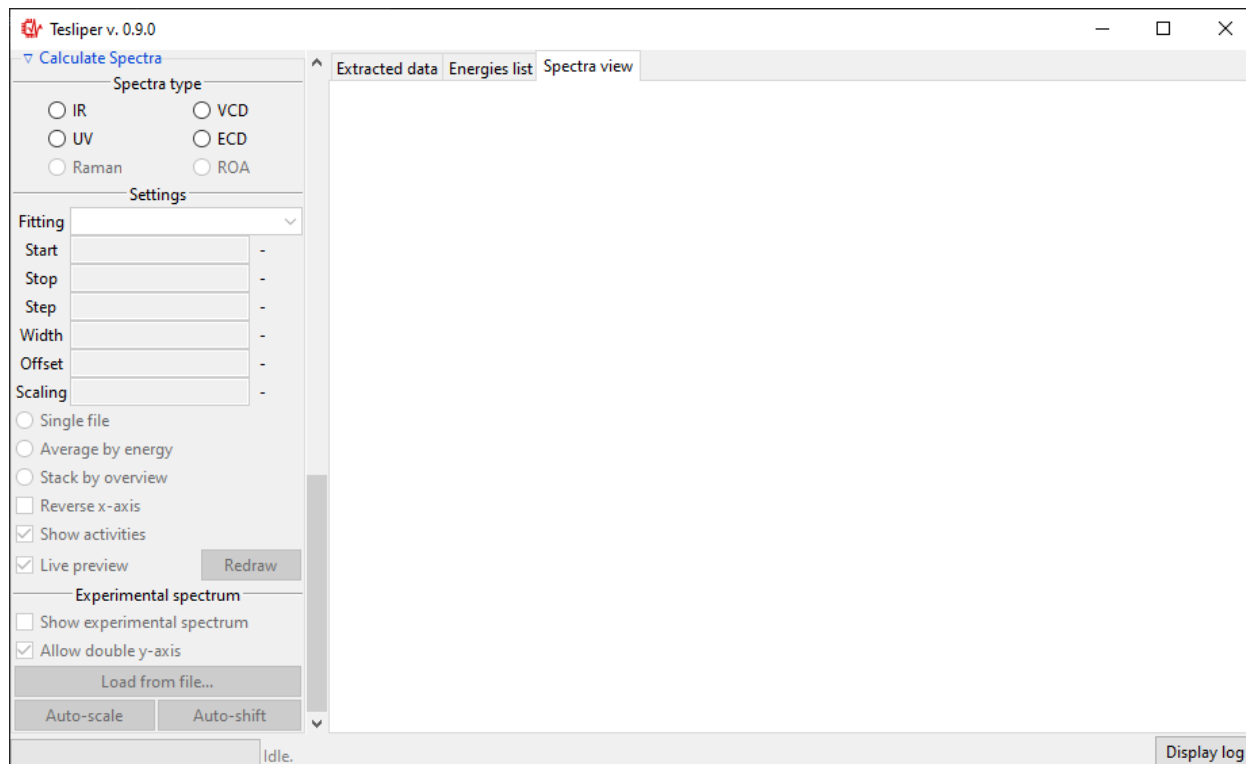
The **Energies** and **structure** section also allows you to specify the temperature of the studied system. This parameter is important for calculation of the Boltzmann distribution of conformers, which is used to estimate conformers' population and average conformers' spectra. The default value is the room temperature, expressed as 298.15 Kelvin (25.0°C). Changing this value will trigger automatic recalculation of **Min. Boltzmann factor** and **Population** values, and average spectra will be redrawn.

RMSD sieve		<input type="checkbox"/> Tolbutamid_c18	--	--	--
Window size: 5.0	kcal/mol	<input checked="" type="checkbox"/> Tolbutamid_c19	1.7632	1.7639	2.2174
Threshold: 1.0	angstrom	<input type="checkbox"/> Tolbutamid_c2	--	--	--
<input checked="" type="checkbox"/> ignore hydrogen atoms		<input type="checkbox"/> Tolbutamid_c20	--	--	--
Trim similar		<input type="checkbox"/> Tolbutamid_c21	--	--	--
		<input checked="" type="checkbox"/> Tolbutamid_c22	1.6602	1.6640	1.0381

New in version 0.9.1: The **Temperature** entry allowing to change the temperature value.

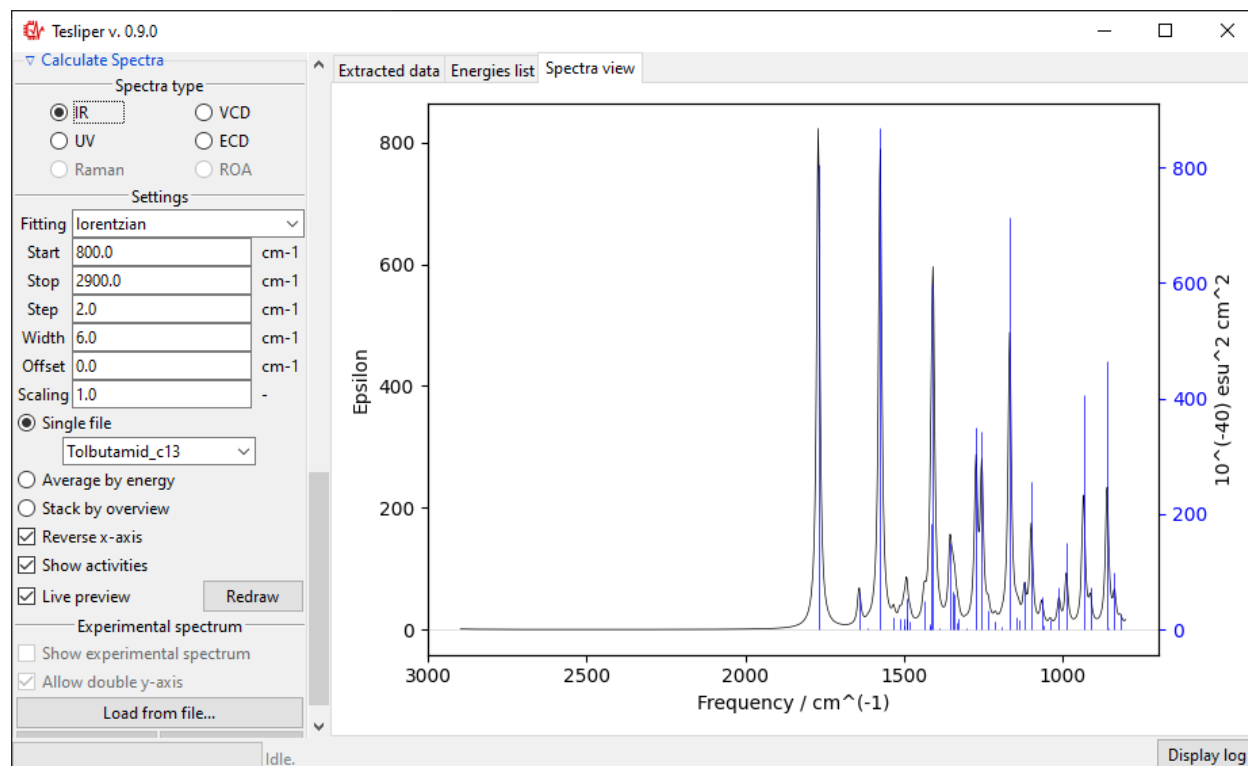
3.3.5 Spectra simulation

Calculate Spectra controls section and **Spectra view** tab allow to preview the simulation of selected spectrum type with given parameters.

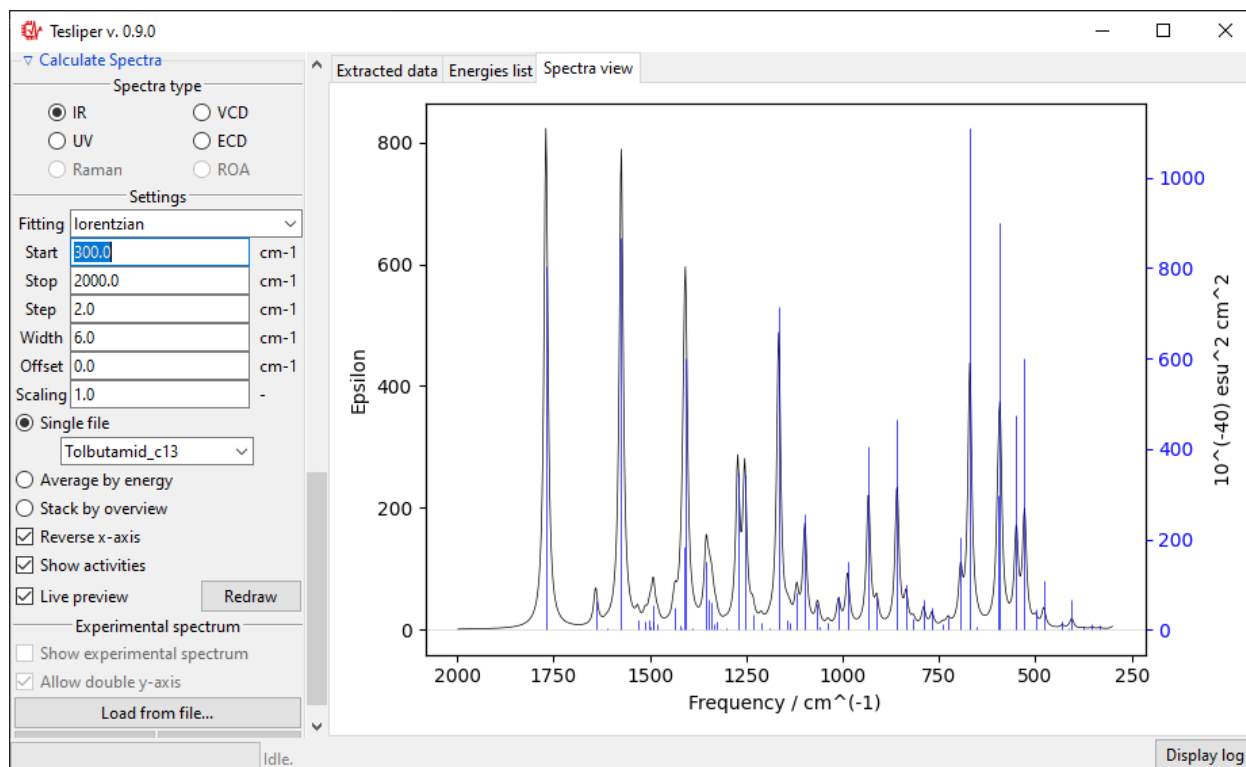


The **Spectra view** tab is initially empty, but when you select one of the available **Spectra types**, **Settings** sub-section will become enabled and the spectrum will be drawn.

Tip: You can turn off automatic recalculation of the spectrum by unchecking the Live preview box.

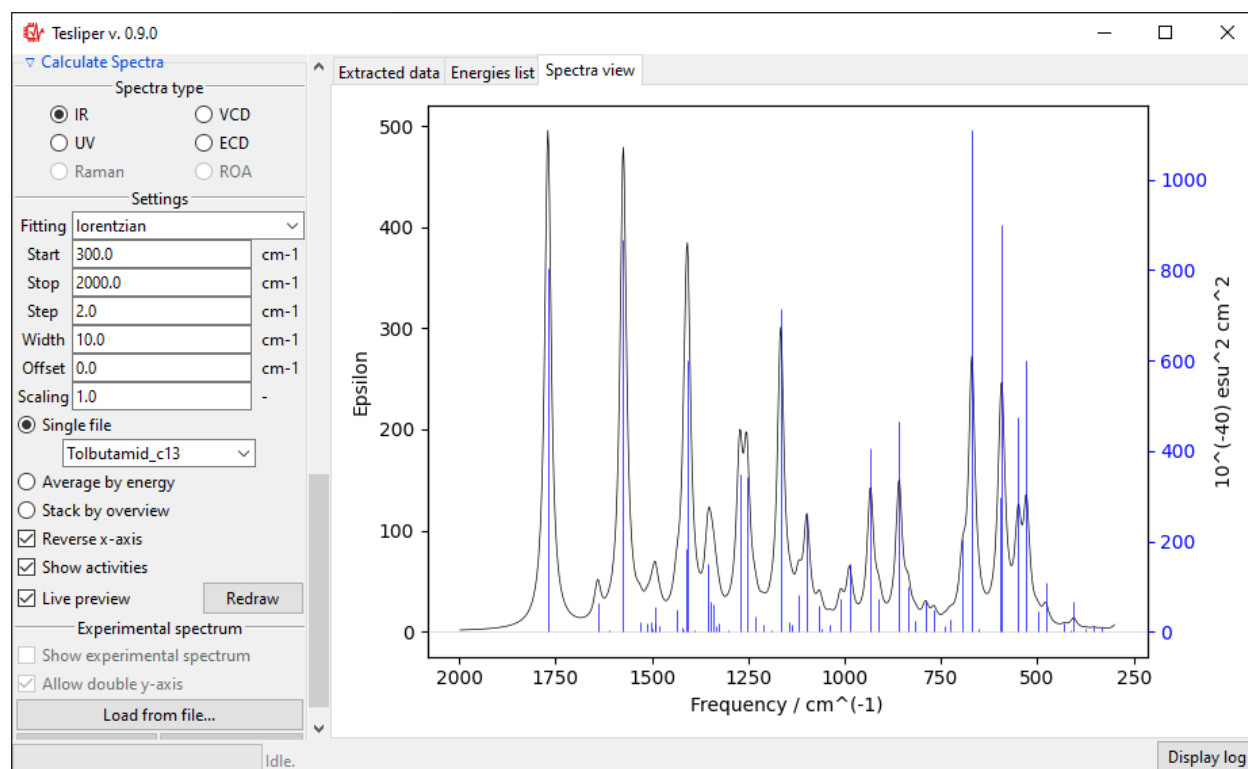


Beginning and end of the simulated spectral range may be set using **Start** and **Stop** fields. The view on the right will match these boundaries. Please note that **Start** must have lower value than **Stop**. There is also a **Step** field that allows you to adjust points density in the simulated spectrum.

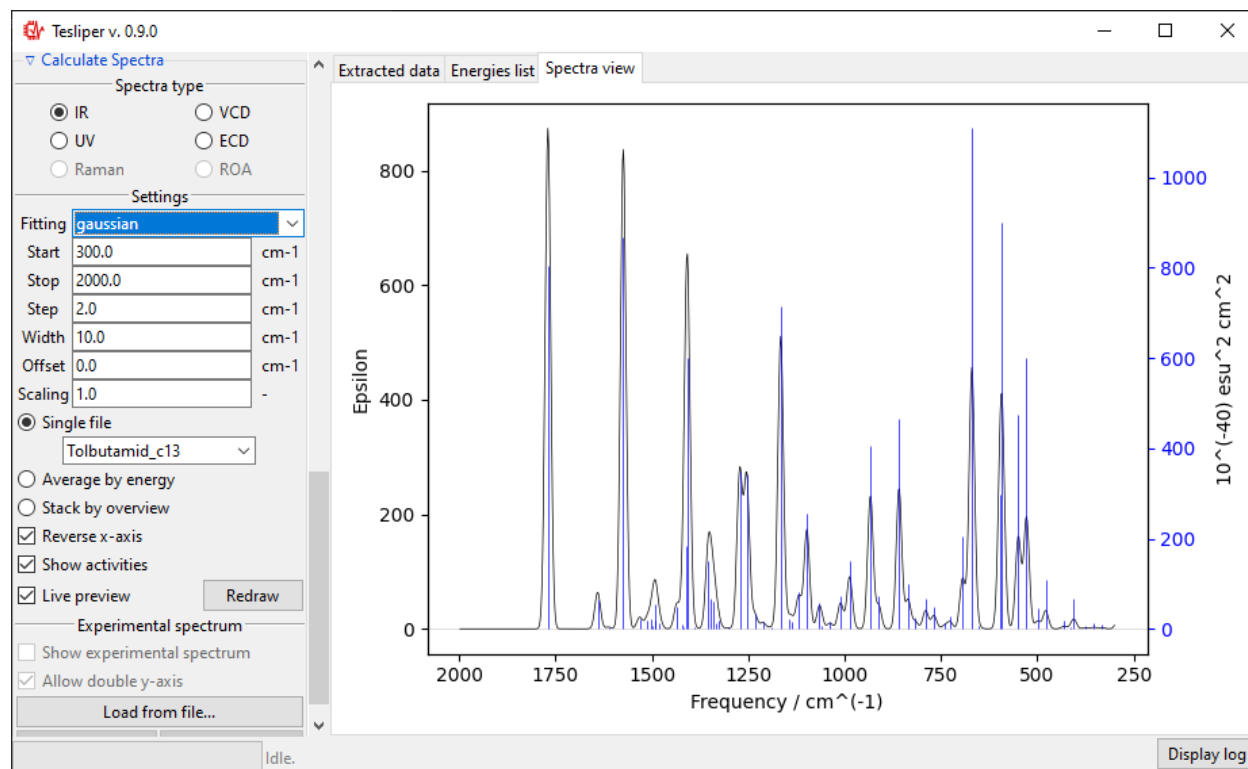


Width field defines a peak width in the simulated spectrum. Its exact meaning depends on the chosen fitting function (see below). For gaussian fitting Width is interpreted as **half width of the peak at $\frac{1}{e}$ of its maximum value (HW10eM)**. For lorentzian function it is interpreted as **half width at half maximum height of the peak (HWHM)**.

Tip: You may change fields' values with the mouse wheel. Point the field with mouse cursor and allow for a small delay before switching from the scroll mode to the value-changing mode. Move the mouse cursor away from the field to switch back.

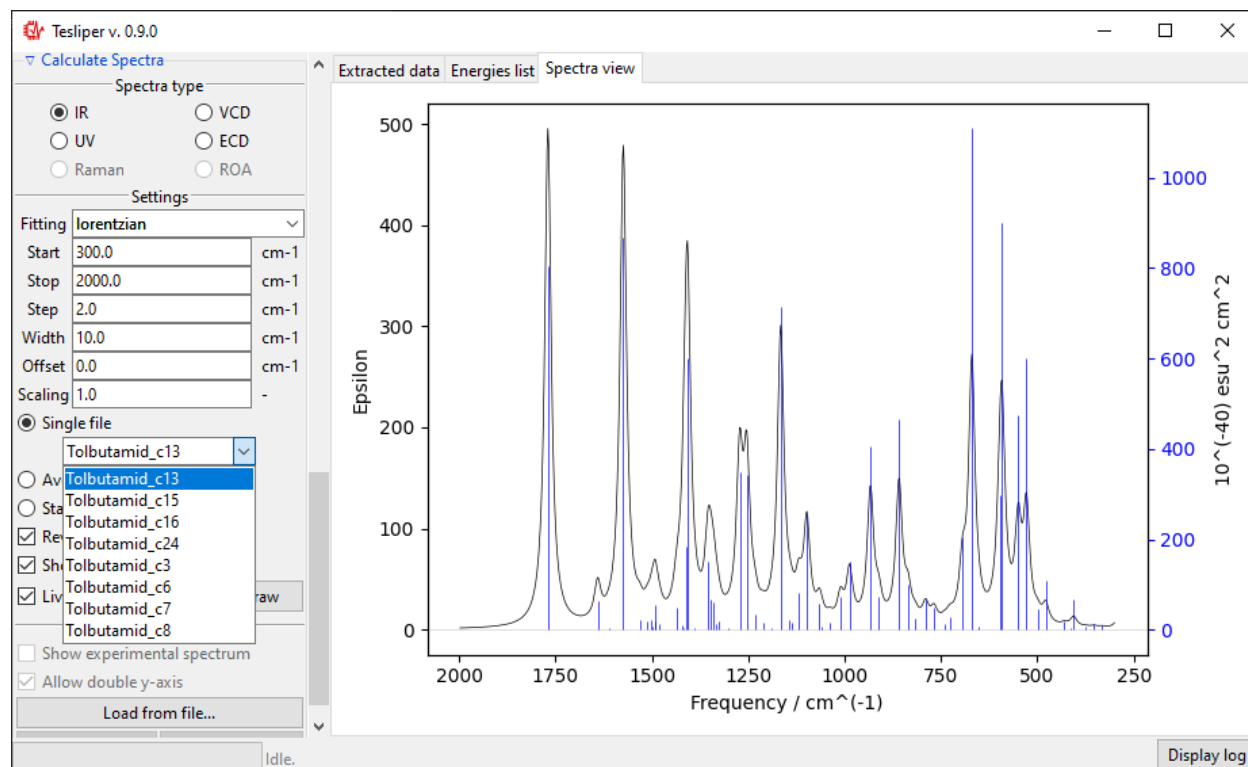


Finally, you may choose the fitting function used to simulate the spectrum from the calculated intensities values - this will have a big impact on simulated peaks' shape. Two such functions are available: gaussian and lorentzian functions. Usually lorentzian function is used to simulate vibrational spectra and gaussian function for electronic spectra.

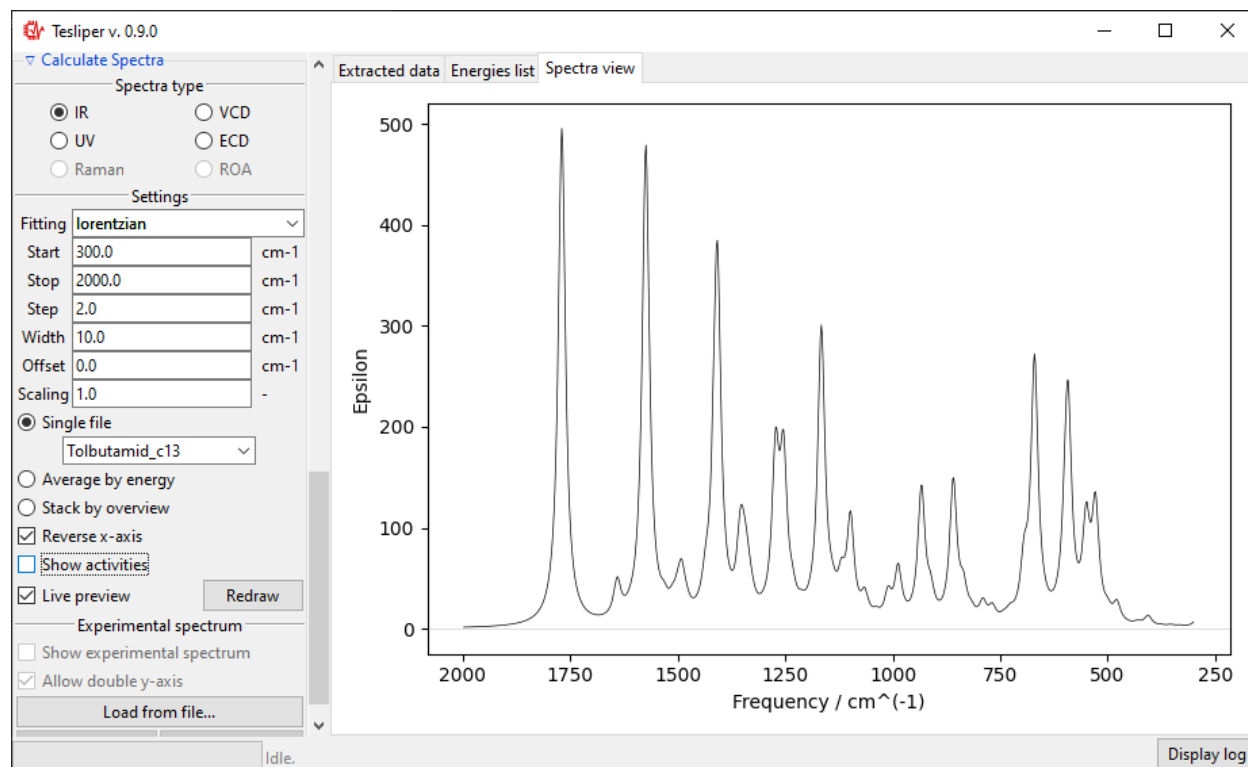


The default spectra preview is a Single file preview that allows you to see the simulated spectrum for the selected

conformer. You may change the conformer to preview using the drop-down menu shown in the screenshot below.

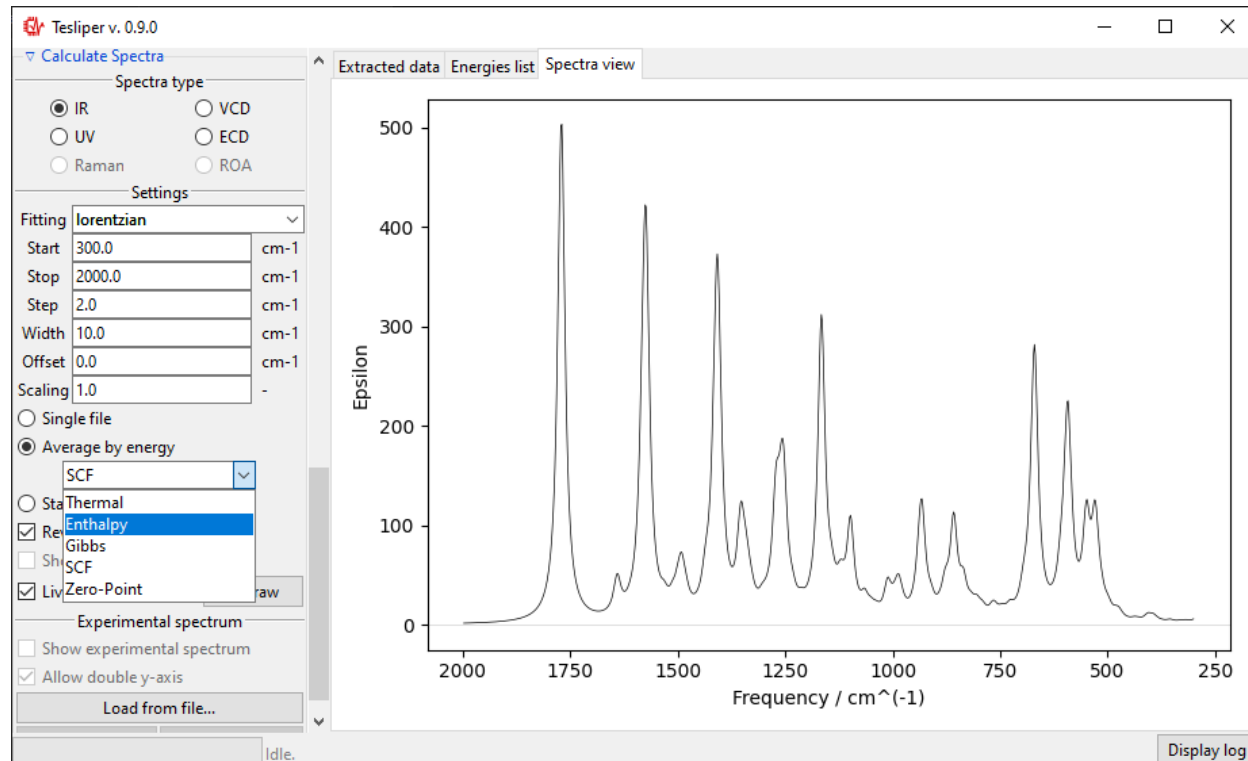


When in a Single file preview, spectral activities used to simulate the spectrum are also shown on the right. You may turn this off by unticking the Show activities box.

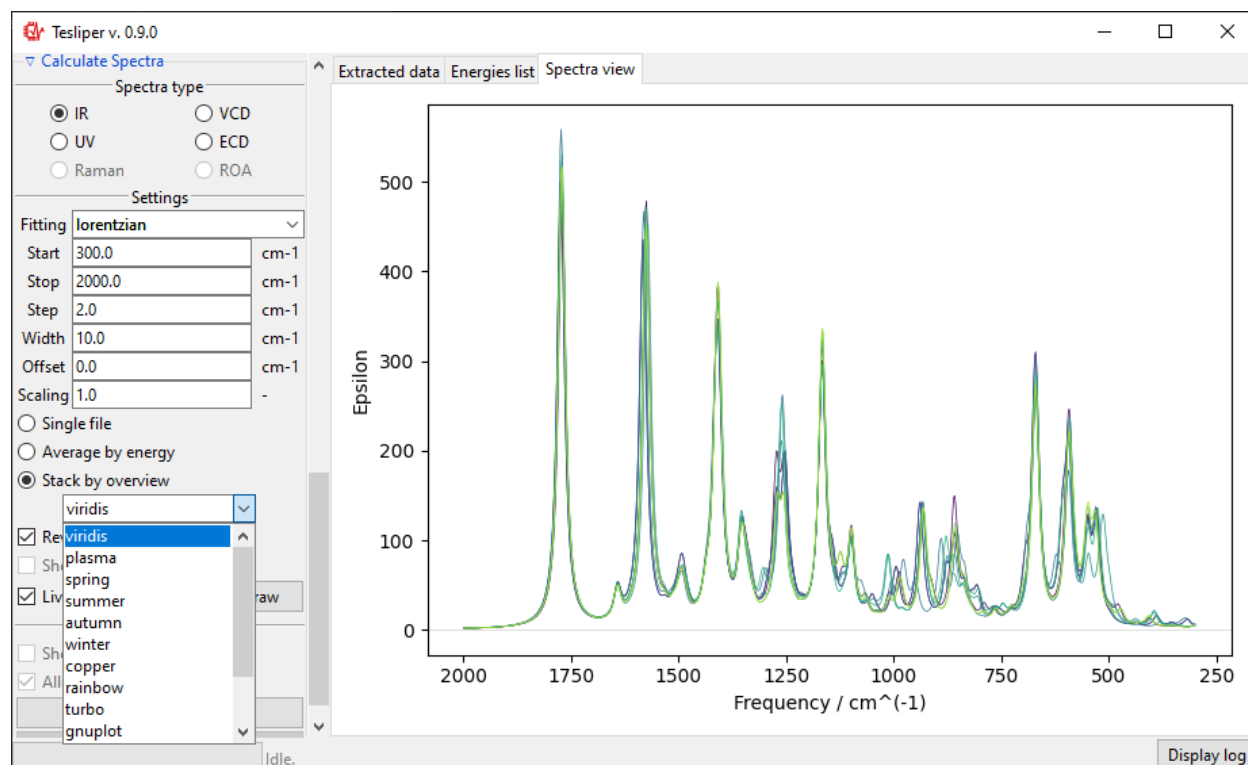


You can also preview an population-weighted average spectrum of all *kept* conformers, by selecting Average by

energy. The drop-down menu lets you select the energies that tesliper should use to calculate conformers populations.

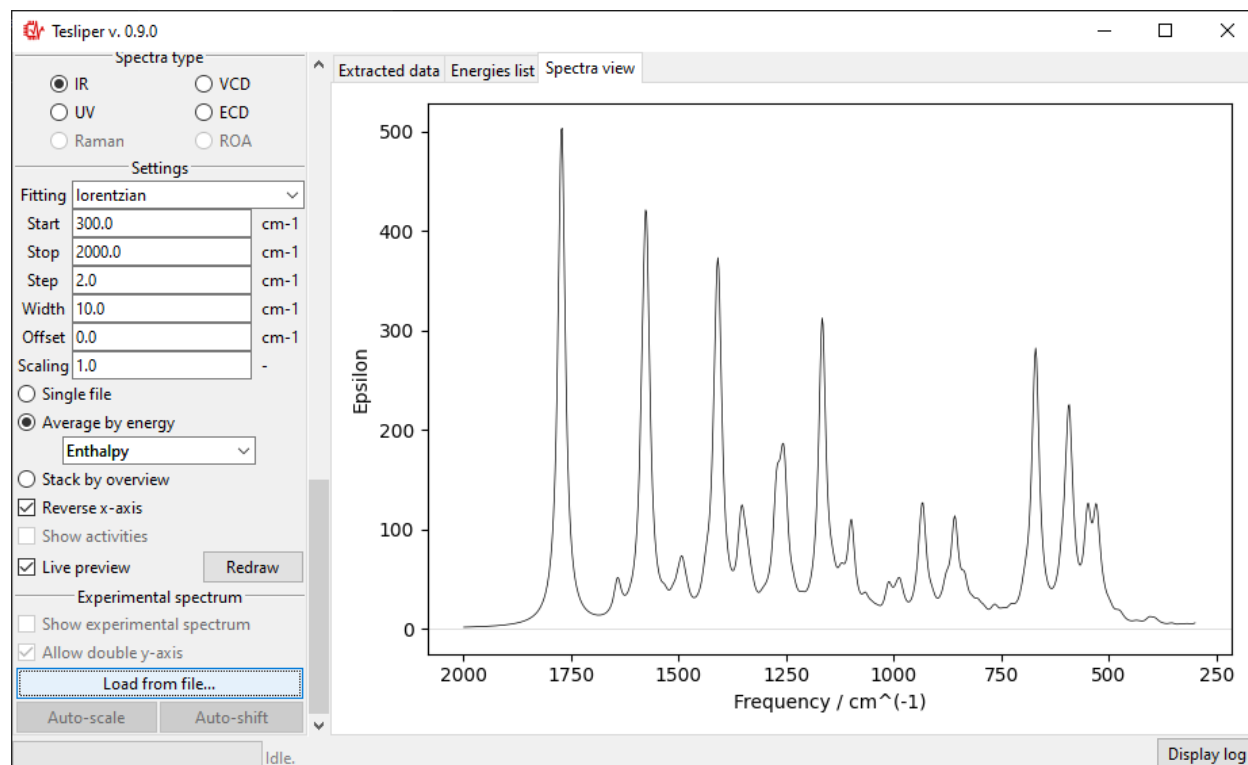


The final option is to show all *kept* conformers at once by selecting Stack by overview option. The drop-down menu allows to choose a color scheme for the stacked spectra lines.

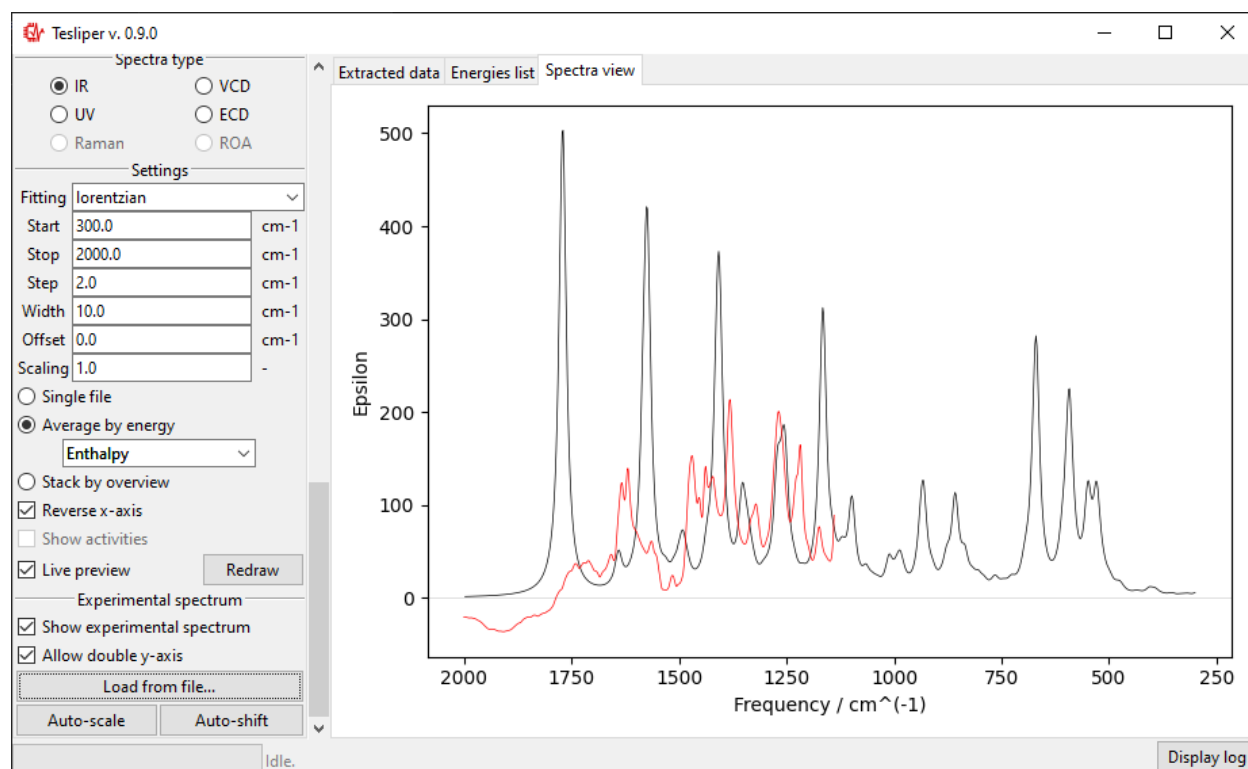


3.3.6 Comparing with experiment

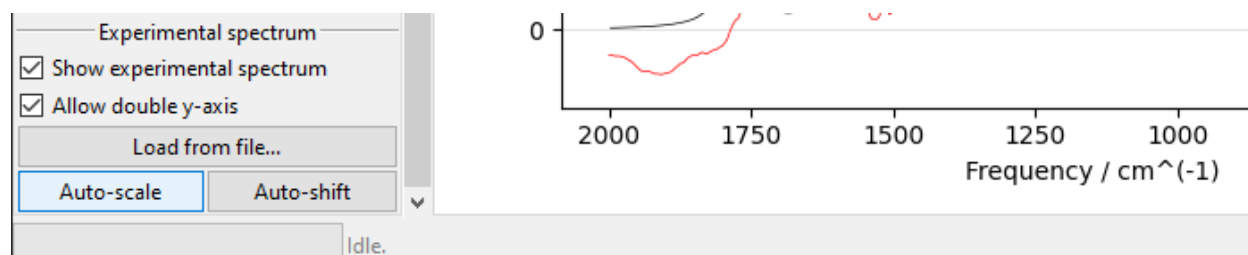
It's possible to and an overlay with the experimental spectrum to Single file and Average by energy previews. To load an experimental spectrum, use Load from file button in the Experimental spectrum subsection. tesliper can read spectrum in the .txt (or .xy) file format. Binary .spc formats are not supported.

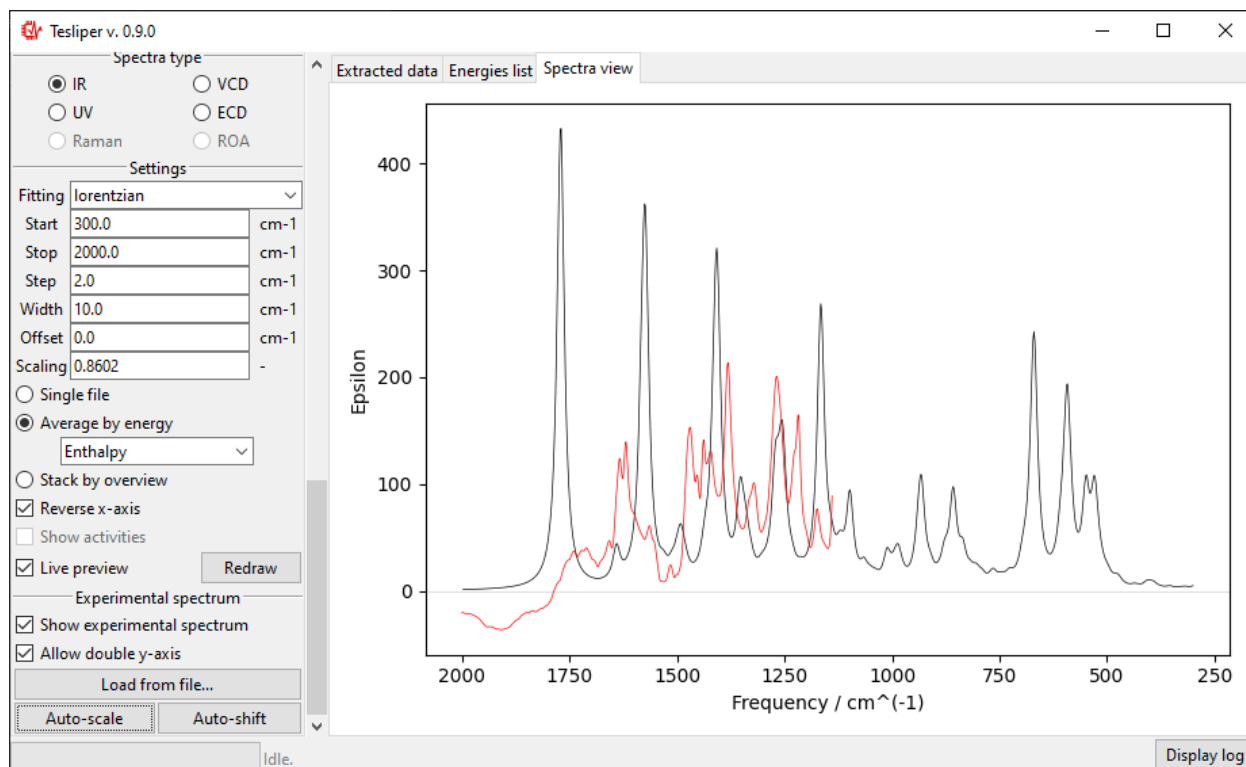


When you choose the experimental spectrum file, it's curve is drawn on the right with respect to the Start and Stop bounds. Red color is used for the experiment. In case of a significant difference in the magnitude of intensity in both spectra, the second scale will be added to the drawing.

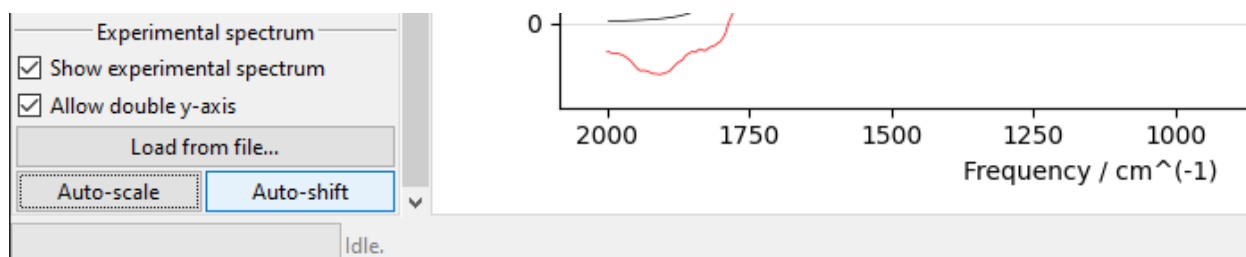


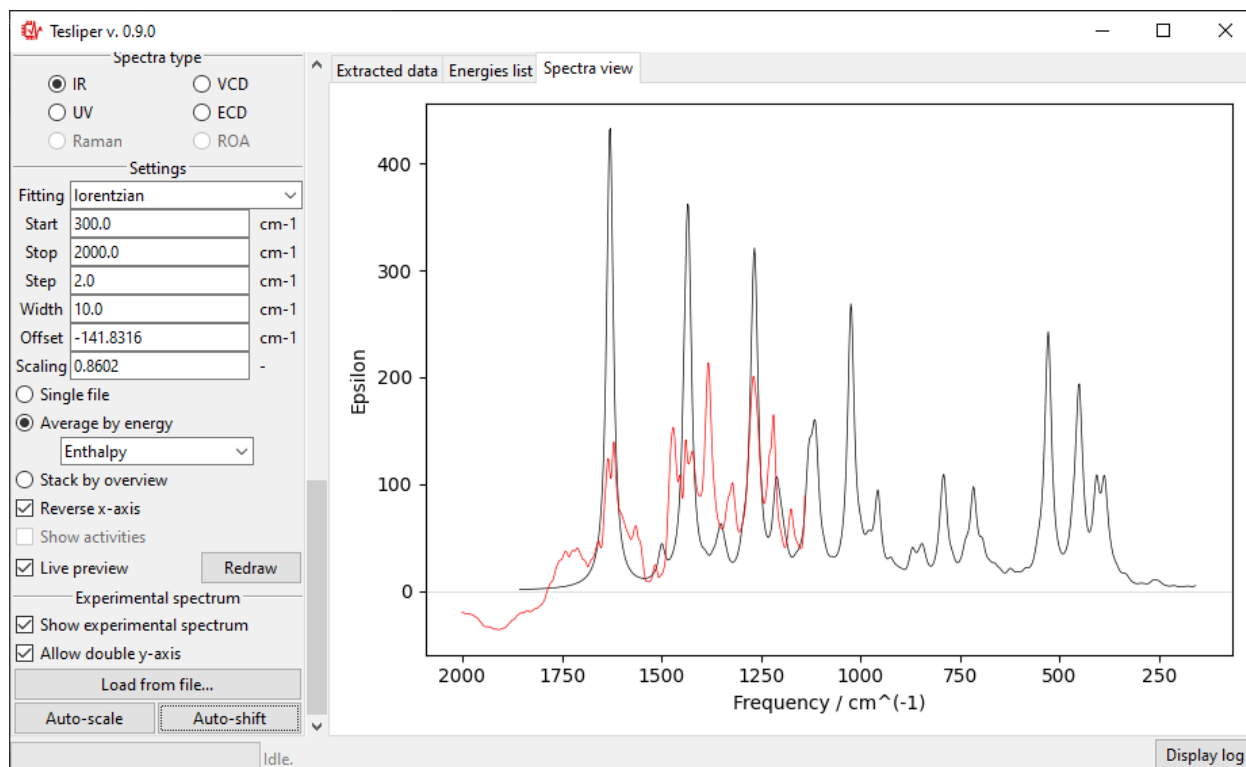
The scale of the simulated values may be automatically adjusted to roughly match the experiment with the Auto-scale button. It may be also adjusted manually by changing the value of the Scaling field.



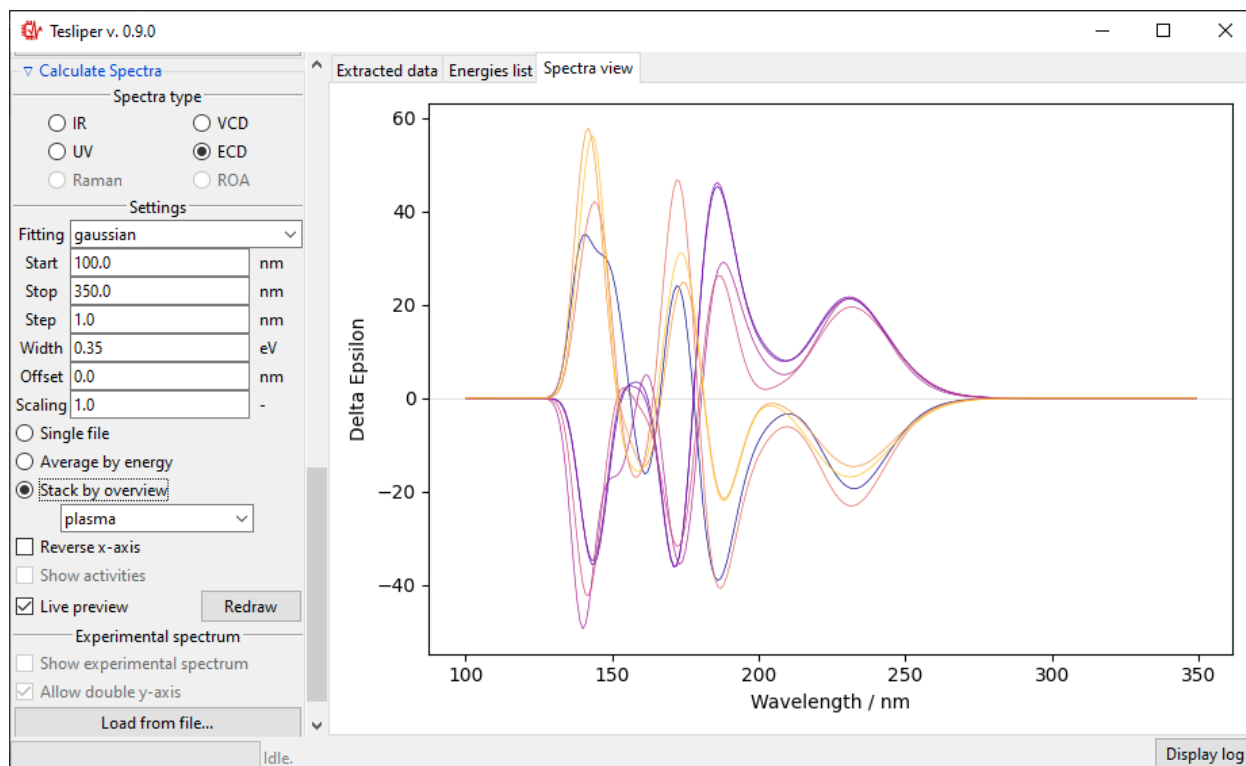


Similarly, Auto-shift button and Offset field let you to adjust simulated spectrum's position on the x-axis. Positive Offset shifts the spectrum bathochromically, a negative one shifts it hypsochromically.



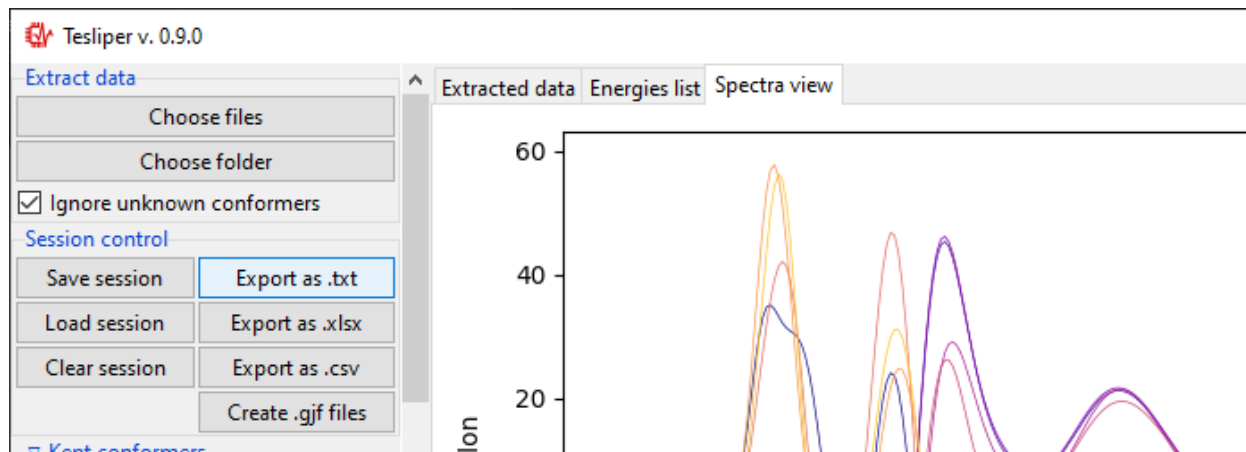


Scaling and Offset values are remembered for the current spectra type, just like the other parameters.

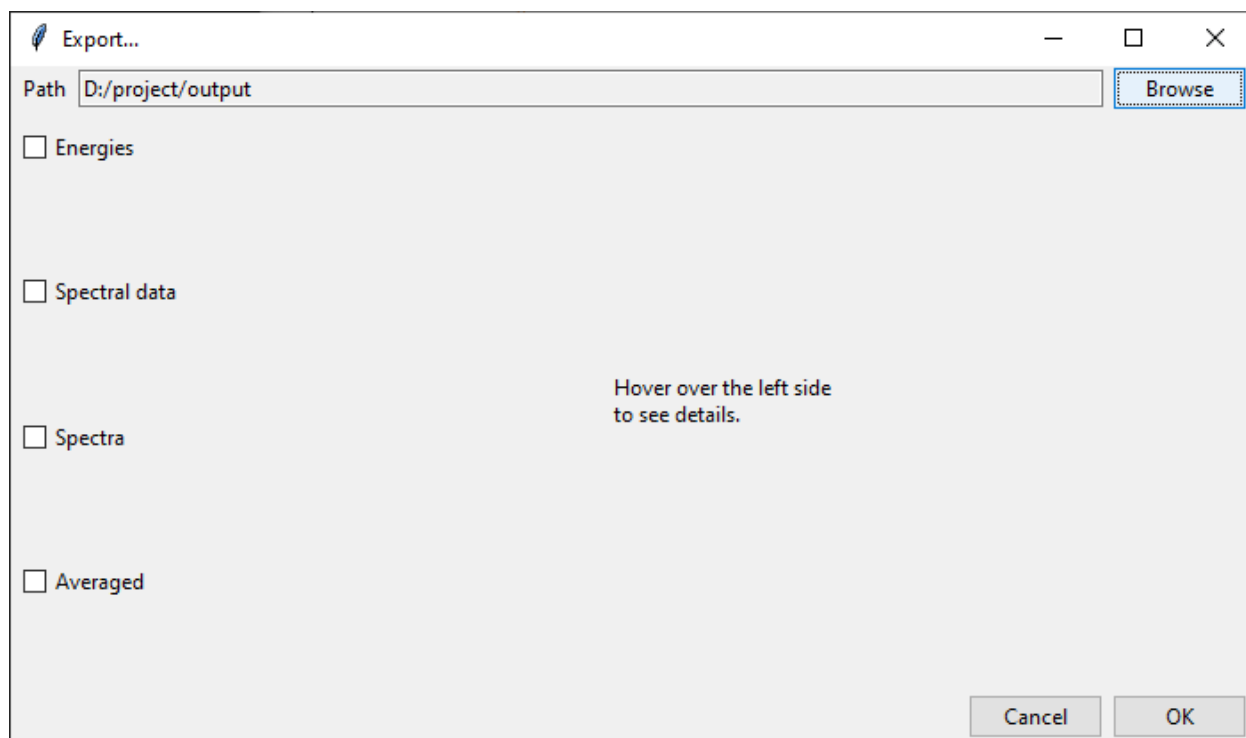


3.3.7 Data export

Calculated and extracted data may be exported to disk in three different formats: text files with **Export to .txt** button, csv files with **Export to .csv** button and Excel files with **Export to .xlsx** button. Clicking on any of those will bring up the **Export . . .** dialog.



At the top of the **Export . . .** dialog is displayed the path to the currently selected output directory. It may be changed by clicking on the **Browse** button and selecting a new destination. Files generated by **tesliper** will be written to this directory.



On the left side of the dialog window you may select what type of data you want to export by ticking appropriate boxes. Once you hover over the certain category, more detailed list of available data will be shown on the right. By ticking/unticking selected boxes you can fine-tune what should be written to disk.

Export...

Path: D:/project/output Browse

<input checked="" type="checkbox"/> Energies	Vibrational Data		
	<input type="checkbox"/> IR Intensity	<input checked="" type="checkbox"/> Dip. Strength	<input checked="" type="checkbox"/> Rot. Strength
	<input type="checkbox"/> Reduced masses	<input type="checkbox"/> Force constants	<input type="checkbox"/> E-M Angle
<input type="checkbox"/> Spectral data	Electronic Data		
	<input checked="" type="checkbox"/> Dip. (velo)	<input type="checkbox"/> Dip. (length)	<input checked="" type="checkbox"/> Rot. (velo)
	<input type="checkbox"/> Rot. (length)	<input type="checkbox"/> Osc. (velo)	<input type="checkbox"/> Osc. (length)
	<input checked="" type="checkbox"/> Highest contrib. transitions	<input type="checkbox"/> All transitions	<input type="checkbox"/> E-M Angle
<input type="checkbox"/> Spectra	Scattering Data		
	<input type="checkbox"/> Raman scatt. activities	<input type="checkbox"/> Raman scatt. activities	<input type="checkbox"/> Raman inten. ICPu/SCPu(180)
	<input type="checkbox"/> ROA inten. ICPu/SCPu(180)	<input type="checkbox"/> Raman inten. ICPd/SCPd(90)	<input type="checkbox"/> ROA inten. ICPd/SCPd(90)
	<input type="checkbox"/> Raman inten. DCPI(180)	<input type="checkbox"/> ROA inten. DCPI(180)	<input type="checkbox"/> Depolar-P Raman
	<input type="checkbox"/> Depolar-U Raman	<input type="checkbox"/> Depolar-P ROA	<input type="checkbox"/> Depolar-U ROA
<input type="checkbox"/> Averaged	<input type="checkbox"/> Raman invariant Alpha2	<input type="checkbox"/> Raman invariant Beta2	<input type="checkbox"/> ROA invariant AlphaG
	<input type="checkbox"/> ROA invariant Gamma2	<input type="checkbox"/> ROA invariant Delta2	<input type="checkbox"/> CID ICPu/SCPu(180)
	<input type="checkbox"/> CID ICPd/SCPd(90)	<input type="checkbox"/> CID DCPI(180)	<input type="checkbox"/> Degree of circularity

Cancel OK

In the Spectra category, beside each available spectra type, there is a note that informs if calculation parameters were altered by the user. Spectra will be recalculated with current parameters upon the export confirmation.

Export...

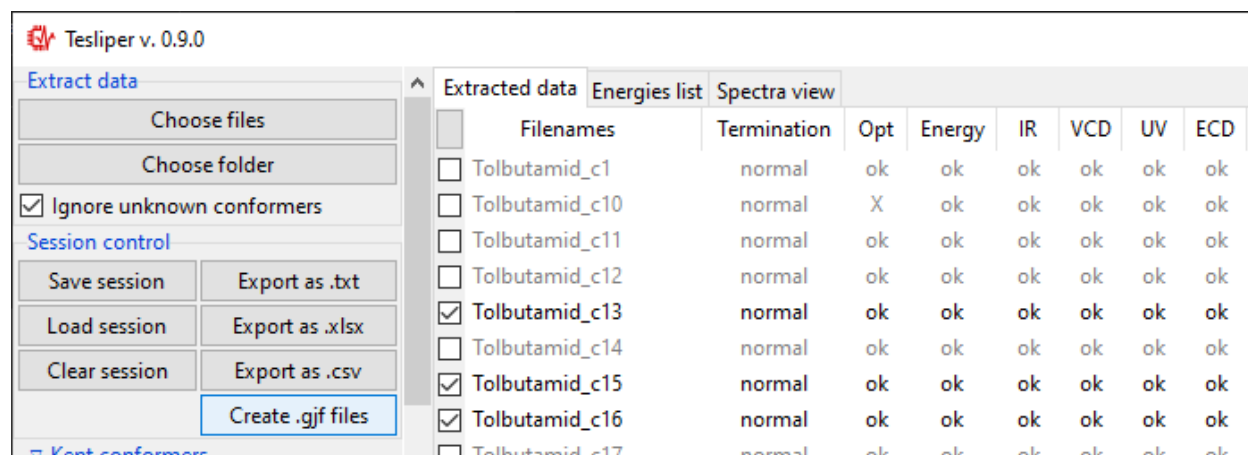
Path: D:/project/output Browse

<input checked="" type="checkbox"/> Energies	<input checked="" type="checkbox"/> IR	[user parameters]
<input type="checkbox"/> Spectral data	<input checked="" type="checkbox"/> VCD	[default parameters]
	<input checked="" type="checkbox"/> UV	[default parameters]
<input type="checkbox"/> Spectra	<input checked="" type="checkbox"/> ECD	[user parameters]
<input type="checkbox"/> Averaged	<input type="checkbox"/> Raman	
	<input type="checkbox"/> ROA	

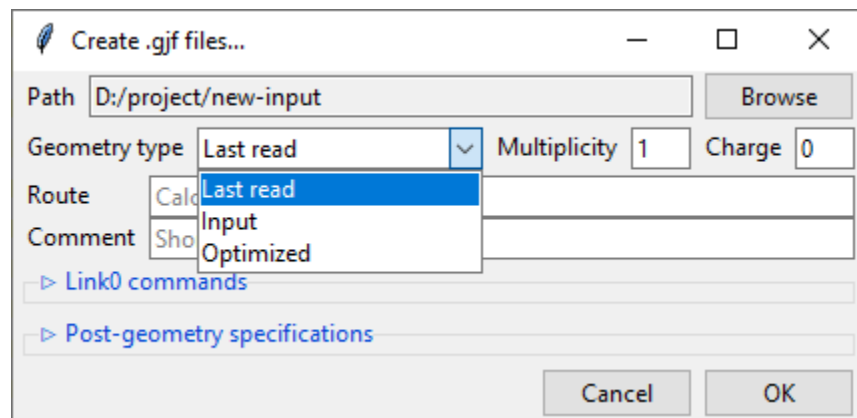
Cancel OK

3.3.8 Creating Gaussian input

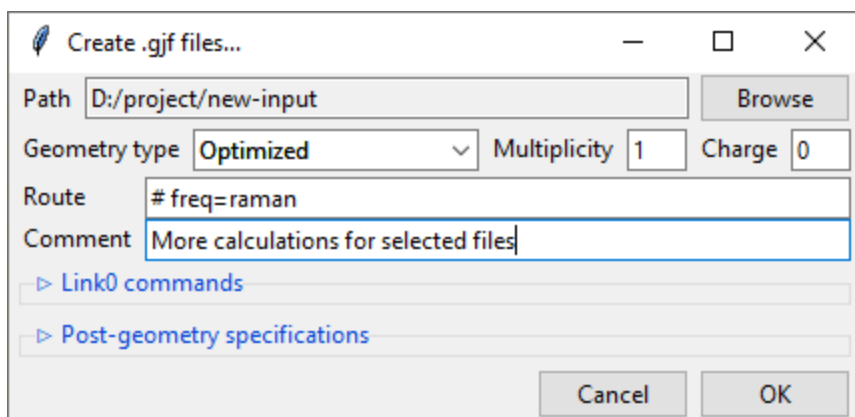
Clicking on the Create .gjf files... will open a dialog window that lets you setup a next step of calculations to conduct with the Gaussian software.



Similarly to the previews one, this dialog also features a Path field that specifies the output directory, which may be changed by clicking on the Browse button. Below it is the Geometry type drop-down menu that allows you to select, which geometry specification should be used in the new input files. Input is the geometry used as an input in the extracted .log/.out files, Last read is the one that was lastly encountered in these files. Optimized is the geometry marked as optimized by Gaussian, but it is only available from the successful optimization calculations. You also need to specify the Charge and the Multiplicity of the molecule.



Below are the Route and Comment fields. The first one specifies the calculation directives for the Gaussian software. The second one is a title section required by GAussian.



Create .gjf files...

Path: D:/project/new-input Browse

Geometry type: Optimized Multiplicity: 1 Charge: 0

Route: # freq=raman

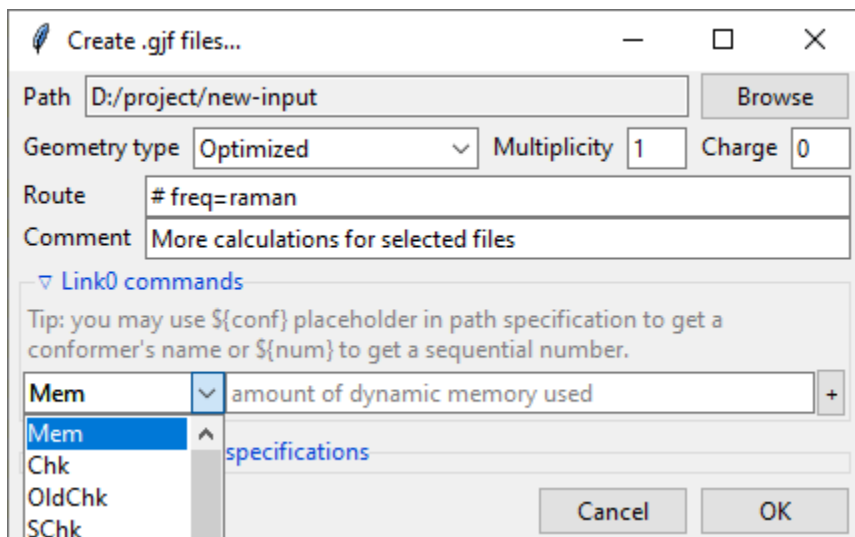
Comment: More calculations for selected files

▶ Link0 commands

▶ Post-geometry specifications

Cancel OK

Further below is the expandable Link0 commands panel that allows to specify Link 0 directives, which define location of scratch files, memory usage, etc. Select a command name from the drop-down menu, filed on right will show a hint about its purpose.



Create .gjf files...

Path: D:/project/new-input Browse

Geometry type: Optimized Multiplicity: 1 Charge: 0

Route: # freq=raman

Comment: More calculations for selected files

▼ Link0 commands

Tip: you may use \${conf} placeholder in path specification to get a conformer's name or \${num} to get a sequential number.

Mem	amount of dynamic memory used	+
Mem		
Chk	specifications	
OldChk		
SChk		

Cancel OK

Provide a value in the input field and click a + button to add a command. It will be added to the list below. You can update the selected command by providing a new value and clicking the + button again or remove it by clicking the - button.

Create .gjf files...

Path: D:/project/new-input Browse

Geometry type: Optimized Multiplicity: 1 Charge: 0

Route: # freq=raman

Comment: More calculations for selected files

▼ Link0 commands

Tip: you may use \${conf} placeholder in path specification to get a conformer's name or \${num} to get a sequential number.

Mem: 100MW + -

▶ Post-geometry specifications

Cancel OK

Path-like commands may be parametrized: `${conf}` will be substituted with the name of conformer and `${num}` will be substituted with the sequential number.

Create .gjf files...

Path: D:/project/new-input Browse

Geometry type: Optimized Multiplicity: 1 Charge: 0

Route: # freq=raman

Comment: More calculations for selected files

▼ Link0 commands

Tip: you may use \${conf} placeholder in path specification to get a conformer's name or \${num} to get a sequential number.

Chk: /checkpoints/\${conf}.chk + -

Mem: 100MW -

▶ Post-geometry specifications

Cancel OK

Finally, you can add a post-geometry specification. It will be written to the end of each .gjf file.

Create .gif files...

Path

Geometry type Multiplicity Charge

Route

Comment

▼ Link0 commands

Tip: you may use \${conf} placeholder in path specification to get a conformer's name or \${num} to get a sequential number.

Chk

Mem

Chk

▼ Post-geometry specifications

```
Zr 0
LANL2DZ
****
C1 N O C H 0
6-31G(d,p)
****

Zr 0
LANL2
```

3.3.9 Saving session

You can save a session (all data, along with current trimmed and parameters) with a **Save session** button. A popup dialog will be opened, where you can specify a target session file location.

Tesliper v. 0.9.0

Extract data

☒ Ignore unknown conformers

Session control

▼ Kent conformers

Extracted data	Energies list	Spectra view	Termination	Opt	Energy	IR	VCD	UV	ECD
<input type="checkbox"/> Tolbutamid_c1			normal	ok	ok	ok	ok	ok	ok
<input type="checkbox"/> Tolbutamid_c10			normal	X	ok	ok	ok	ok	ok
<input type="checkbox"/> Tolbutamid_c11			normal	ok	ok	ok	ok	ok	ok
<input type="checkbox"/> Tolbutamid_c12			normal	ok	ok	ok	ok	ok	ok
<input checked="" type="checkbox"/> Tolbutamid_c13			normal	ok	ok	ok	ok	ok	ok
<input type="checkbox"/> Tolbutamid_c14			normal	ok	ok	ok	ok	ok	ok
<input checked="" type="checkbox"/> Tolbutamid_c15			normal	ok	ok	ok	ok	ok	ok
<input checked="" type="checkbox"/> Tolbutamid_c16			normal	ok	ok	ok	ok	ok	ok
<input type="checkbox"/> Tolbutamid_c17			normal	ok	ok	ok	ok	ok	ok

To load previously saved session use the **Load session** button. You can also discard all currently held data by clicking the **Clear session** button.

Warning: Loading and clearing session cannot be undone! A confirmation dialog will be displayed for those actions.

3.4 Scripting with tesliper

This part discusses basics of using Python API. For tutorial on using a Graphical Interface, see [gui](#).

tesliper provides a *Tesliper* class as a main entry point to its functionality. This class allows you to easily perform any typical task: read and write files, filter data, calculate and average spectra. It is recommended to read [Conventions and Terms](#) to get a general idea of what to expect. The next paragraphs will introduce you to basic use cases with examples and explanations. The examples do not use real data, but simplified mockups, to not obscure the logic presented.

```
from tesliper import Tesliper

# extract data from Gaussian output files
tslr = Tesliper(input_dir="./opt_and_freq")
tslr.extract()

# conditional filtering of conformers
tslr.conformers.trim_non_normal_termination()
tslr.conformers.trim_not_optimized()
tslr.conformers.trim_imaginary_frequencies()
tslr.conformers.trim_to_range("gib", maximum=10, attribute="deltas")
tslr.conformers.trim_rmsd(threshold=1.0, window_size=0.5, energy_genre="gib")

# calculate and average spectra, export data
tslr.calculate_spectra()
tslr.average_spectra()
tslr.export_energies(fmt="txt")
tslr.export_averaged(fmt="csv")
```

3.4.1 Reading files

After importing *Tesliper* class, we instantiate it with an *input_dir* parameter, which is a path to the directory containing output files from quantum chemical calculations software. You may also provide an *output_dir* parameter, defining where tesliper should write the files it generates. Both of those parameters are optional and default to the current working directory, if omitted. You may also provide a *wanted_files* parameter, which should be a list of filenames that *Tesliper* should parse, ignoring any other files present in *input_dir*. Omitting *wanted_files* means that no file should be ignored.

Note: *Tesliper* accepts also *quantum_software* parameter, which is a hint for tesliper on how it should parse output files it reads. However, only Gaussian software is supported out-of-the-box, and *quantum_software="gaussian"* is a default value. If you wish to use tesliper to work with another qc package, you will need to define a custom parser that subclasses the *ParserBase* class. Refer to its documentation for more information.

You can extract data from the files in *output_dir* using *Tesliper.extract()* method. *Tesliper.extract()* respects *input_dir* and *wanted_files* given to *Tesliper*, but *path* and *wanted_files* parameters provided to the method

call will take precedence. If you would like to read files in the whole directory tree, you may perform a recursive extraction, using `extract(recursive=True)`. So assuming a following directory structure:

```
project
├── optimization
│   ├── conf_one.out
│   ├── conf_two.out
│   └── conf_three.out
└── vibrational
    ├── conf_one.out
    ├── conf_two.out
    └── conf_three.out
```

you could use any of the following to get the same effect.

```
# option 1: change *input_dir*
tslr = Tesliper(input_dir="./project/optimization")
tslr.extract()
tslr.input_dir = "./project/vibrational"
tslr.extract()

# option 2: override *input_dir* only for one call
tslr = Tesliper(input_dir="./project/optimization")
tslr.extract()
tslr.extract(path="./project/vibrational")

# option 3: read the whole tree
tslr = Tesliper(input_dir="./project")
tslr.extract(recursive=True)
```

tesliper will try to guess the extension of files it should parse: e.g. Gaussian output files may have “.out” or “.log” extension. If those are mixed in the source directory, an exception will be raised. You can prevent this by providing the *extension* parameter, only files with given extension will be parsed.

```
project
├── conf_one.out
└── conf_two.log
```

```
tslr = Tesliper(input_dir="./project")
tslr.extract() # raises ValueError
tslr.extract(extension="out") # ok
```

3.4.2 Filtering conformers

`Tesliper.extract()` will read and parse files it thinks are output files of the quantum chemical software and update a `Tesliper.conformers` internal data storage. It is a dict-like `Conformers` instance, that stores data for each conformer in a form of an ordinary dict. This inner dict uses *genre* names as keys and data as values (the form of which depends on the genre itself). `Conformers` provide a number of methods for filtering conformers it knows, allowing to easily hide data that should be excluded from further analysis. tesliper calls this process a *trimming*. The middle part of the first code snippet are example of trimming conformers:

```
tslr.conformers.trim_non_normal_termination()
tslr.conformers.trim_not_optimized()
tslr.conformers.trim_imaginary_frequencies()
tslr.conformers.trim_to_range("gib", maximum=10, attribute="deltas")
tslr.conformers.trim_rmsd(threshold=1.0, window_size=0.5, energy_genre="gib")
```

As you may suspect, `trim_non_normal_termination()` hides data from calculations that did not terminate normally, `trim_not_optimized()` hides data from conformers that are not optimized, and `trim_imaginary_frequencies()` hides data from conformers that have at least one imaginary frequency. More trimming methods is described *below*.

Conformers hidden are *not kept*. Information about which conformers are *kept* and *not kept* is stored in `Conformers.kept` attribute, which may also be manipulated more directly. More on this topic will be *explained later*.

As mentioned earlier, `Tesliper.conformers` is a dict-like structure, and as such offers a typical functionality of Python's dicts. However, checking for presence with `conf in tslr.conformers` or requesting a view with standard `keys()`, `values()`, or `items()` will operate on the whole data set, ignoring any trimming applied earlier. `Conformers` class offers additional `kept_keys()`, `kept_values()`, and `kept_items()` methods, that return views that acknowledge trimming.

Trimming methods

There is a number of those methods available for you, beside those mentioned above. Below you will find them listed with a short summary and a link to a more comprehensive explanation in the method's documentation.

`trim_incomplete()` Filters out conformers that doesn't contain data for as many expected genres as other conformers.

`trim_imaginary_frequencies()` Filters out conformers that contain imaginary frequencies (any number of negative frequency values).

`trim_non_matching_stoichiometry()` Filters out conformers that have different stoichiometry than expected.

`trim_not_optimized()` Filters out conformers that failed structure optimization.

`trim_non_normal_termination()` Filters out conformers, which calculation job did not terminate normally (was erroneous or interrupted).

`trim_inconsistent_sizes()` Filters out conformers that have iterable data genres in different size than most conformers. Helpful when `InconsistentDataError` occurs.

`trim_to_range()` Filters out conformers that have a value of some specific data or property outside of the given range, e.g. their calculated population is less than 0.01.

`trim_rmsd()` Filters out conformers that are identical to another conformer, judging by a given threshold of the root-mean-square deviation of atomic positions (RMSD).

`select_all()` Marks all conformers as *kept*.

`reject_all()` Marks all conformers as *not kept*.

Manipulating `Conformers.kept`

Information, which conformer is *kept* and which is not, is stored in the `Conformers.kept` attribute. It is a list of booleans, one for each conformer stored, defining which conformers should be processed by `tesliper`.

```
# assuming "conf_two" has imaginary frequencies
tslr.conformers.trim_imaginary_frequencies()
tslr.conformers.kept == [True, False, True] # True
tslr.export_data(["genres", "to", "export"])
# only files for "conf_one" and "conf_three" are generated
```

`Conformers.kept` may be modified using trimming methods described *earlier*, but also more directly: by setting it to a new value. Firstly, it is the most straightforward to just assign a new list of boolean values to it. This list should have the same number of elements as the number of conformers contained. A `ValueError` is raised if it doesn't.

```
>>> tslr.conformers.kept
[True, True, True]
>>> tslr.conformers.kept = [False, True, False]
>>> tslr.conformers.kept
[False, True, False]
>>> tslr.conformers.kept = [False, True, False, True]
Traceback (most recent call last):
...
ValueError: Must provide boolean value for each known conformer.
4 values provided, 3 excepted.
```

Secondly, list of filenames of conformers intended to be *kept* may be given. Only these conformers will be *kept*. If given filename is not in the underlying `tslr.conformers`' dictionary, `KeyError` is raised.

```
>>> tslr.conformers.kept = ['conf_one']
>>> tslr.conformers.kept
[True, False, False]
>>> tslr.conformers.kept = ['conf_two', 'other']
Traceback (most recent call last):
...
KeyError: Unknown conformers: other.
```

Thirdly, list of integers representing conformers' indices may be given. Only conformers with specified indices will be *kept*. If one of given integers can't be translated to conformer's index, `IndexError` is raised. Indexing with negative values is not supported currently.

```
>>> tslr.conformers.kept = [1, 2]
>>> tslr.conformers.kept
[False, True, True]
>>> tslr.conformers.kept = [2, 3]
Traceback (most recent call last):
...
IndexError: Indexes out of bounds: 3.
```

Fourthly, assigning `True` or `False` to this attribute will mark all conformers as *kept* or *not kept* respectively.

```
>>> tslr.conformers.kept = False
>>> tslr.conformers.kept
[False, False, False]
```

(continues on next page)

(continued from previous page)

```
>>> tslr.conformers.kept = True
>>> tslr.conformers.kept
[True, True, True]
```

Warning: List of *kept* values may be also modified by setting its elements to True or False. It is advised against, however, as a mistake such as `tslr.conformers.kept[:2] = [True, False, False]` will break some functionality by forcibly changing size of `tslr.conformers.kept` list.

Trimming temporarily

Conformers provide two convenience context managers for temporarily trimming its data: `untrimmed` and `trimmed_to()`. The first one will simply undo any trimming previously done, allowing you to operate on the full data set or apply new, complex trimming logic. When Python exits `untrimmed` context, previous trimming is restored.

```
>>> tslr.conformers.kept = [False, True, False]
>>> with tslr.conformers.untrimmed:
>>>     tslr.conformers.kept
[True, True, True]
>>> tslr.conformers.kept
[False, True, False]
```

The second one temporarily applies an arbitrary trimming, provided as a parameter to the `trimmed_to()` call. Any value normally accepted by `Conformers.kept` may be used here.

```
>>> tslr.conformers.kept = [True, True, False]
>>> with tslr.conformers.trimmed_to([1, 2]):
>>>     tslr.conformers.kept
[False, True, True]
>>> tslr.conformers.kept
[True, True, False]
```

Tip: To trim conformers temporarily without discarding a currently applied trimming, you may use:

```
with tslr.conformers.trimmed_to(tslr.conformers.kept):
    ... # temporary trimming upon the current one
```

3.4.3 Simulating spectra

To calculate a simulated spectra you will need to have spectral activities extracted. These will most probably come from a *freq* or *td* Gaussian calculation job, depending on a genre of spectra you would like to simulate. `tesliper` can simulate IR, VCD, UV, ECD, Raman, and ROA spectra, given the calculated values of conformers' optical activity. When you call `Tesliper.calculate_spectra()` without any parameters, it will calculate spectra of all available genres, using default activities genres and default parameters, and store them in the `Tesliper.spectra` dictionary. Aside from this, the spectra calculated are returned by the method.

You can calculate a specific spectra genres only, by providing a list of their names as a parameter to the `Tesliper.calculate_spectra()` call. Also in this case a default activities genres and default parameters will be used to calculate desired spectra, see *Activities genres* and *Calculation parameters* below to learn how this can be customized.

```
ir_and_uv = tslr.calculate_spectra(["ir", "uv"])
assert ir_and_uv["ir"] is tslr.spectra["ir"]
```

Calculation parameters

tesliper uses [Lorentzian](#) or [Gaussian](#) fitting function to simulate spectra from corresponding optical activities values. Both of these require to specify a desired width of peak, as well as the beginning, end, and step of the abscissa (x-axis values). If not told otherwise, tesliper will use a default values for these parameters and a default fitting function for a given spectra genre. These default values are available *via* [Tesliper.standard_parameters](#) and are as follows.

Table 1: Default calculation parameters

Parameter	IR, VCD, Raman, ROA	UV, ECD
width	6 [cm ⁻¹]	0.35 [eV]
start	800 [cm ⁻¹]	150 [nm]
stop	2900 [cm ⁻¹]	800 [nm]
step	2 [cm ⁻¹]	1 [nm]
fitting	lorentzian()	gaussian()

You can change the parameters used for spectra simulation by altering values in the [Tesliper.parameters](#) dictionary. It stores a dict of parameters' values for each of spectra genres ("ir", "vcd", "uv", "ecd", "raman", and "roa"). *start*, *stop*, and *step* expect its values to be in cm⁻¹ units for vibrational and scattering spectra, and nm units for electronic spectra. *width* expects its value to be in cm⁻¹ units for vibrational and scattering spectra, and eV units for electronic spectra. *fitting* should be a callable that may be used to simulate peaks as curves, preferably one of: [gaussian\(\)](#) or [lorentzian\(\)](#).

```
# change parameters' values one by one
tslr.parameters["uv"]["step"] = 0.5
tslr.parameters["uv"]["width"] = 0.5

tslr.parameters["vcd"].update( # or with an update
    {"start": 500, "stop": 2500, "width": 2}
)

# "fitting" should be a callable
from tesliper import lorentzian
tslr.parameters["uv"]["fitting"] = lorentzian
```

Table 2: Descriptions of parameters

Parameter	type	Description
width	float or int	the beginning of the spectral range
start	float or int	the end of the spectral range
stop	float or int	step of the abscissa
step	float or int	width of the peak
fitting	Callable	function used to simulate peaks as curves

Warning: When modifying [Tesliper.parameters](#) be careful to not delete any of the key-value pairs. If you need to revert to standard parameters' values, you can just reassign them to [Tesliper.standard_parameters](#).


```
tslr.parameters["ir"] = {
...     "start": 500, "stop": 2500, "width": 2
... } # this will cause problems!
# revert to default values
tslr.parameters["ir"] = tslr.standard_parameters["ir"]
```

Activities genres

Instead of specifying a spectra genre you'd like to get, you may specify an activities genre you prefer to use to calculate a corresponding spectrum. The table below summarizes which spectra genres may be calculated from which activities genres.

Table 3: Spectra and corresponding activities genres

Spectra	Default activity	Other activities
IR	dip	iri
VCD	rot	
UV	vosc	losc, vdip, ldip
ECD	vrot	lrot
Raman	raman1	ramact, ramanactiv, raman2, raman3
ROA	roa1	roa2, roa3

Warning: If you provide two different genres that map to the same spectra genre, only one of them will be accessible, the other will be thrown away. If you'd like to compare results of simulations using different genres, you need to store the return value of `Tesliper.calculate_spectra()` call.

```
>>> out = tslr.calculate_spectra(["vrot", "lrot"])
>>> list(out.keys()) # only one representation returned
["ecd"]
>>> velo = tslr.calculate_spectra(["vrot"])
>>> length = tslr.calculate_spectra(["lrot"])
>>> assert not velo["ecd"] == length["ecd"] # different
```

Averaging spectra

Each possible conformer contributes to the compound's spectrum proportionally to its population in the mixture. `tesliper` can calculate conformers' population from their relative energies, using a technique called [Boltzmann distribution](#). Assuming that any energies genre is available (usually at least *scf* energies are), after calculating spectra you want to simulate, you should call `Tesliper.average_spectra()` to get the final simulated spectra.

```
>>> averaged = tslr.average_spectra() # averages available spectra
>>> assert tslr.averaged is averaged # just a reference
```

`Tesliper.average_spectra()` averages each spectra stored in the `Tesliper.spectra` dictionary, using each available energies genre. Generated average spectra are stored in `Tesliper.averaged` dictionary, using a tuples of ("spectra_genre", "energies_genre") as keys. `average_spectra()` returns a reference to this attribute.

Note: There is also a `Tesliper.get_averaged_spectrum()` method for calculating a single averaged spectrum

using a given spectra genre and energies genre. Value returned by this method is not automatically stored.

Temperature of the system

Boltzmann distribution depends on the temperature of the system, which is assumed to be the room temperature, expressed as 298.15 Kelvin (25.0°C). You can change it by setting `Tesliper.temperature()` attribute to the desired value. This must be done before calculation of the average spectrum to have an effect.

```
>>> tslr.temperature
298.15 # default value in Kelvin
>>> averaged = tslr.average_spectra() # averages available spectra
>>> tslr.temperature = 300.0 # in Kelvin
>>> high_avg = tslr.average_spectra()
>>> assert not averaged == high_avg # resulting average is different
```

Note: `Tesliper.temperature()` value must be a positive number, absolute zero or lower is not allowed, as it would cause problems in calculation of Boltzmann distribution. An attempt to set temperature to equal or below 0 K will raise a `ValueError`.

3.4.4 Comparing with experiment

The experimental spectrum may be loaded with `tesliper` from a text or CSV file. The software helps you adjust the shift and scale of your simulated spectra to match the experiment. Unfortunately, `tesliper` does not offer broad possibilities when it comes to mathematical comparison of the simulated spectra and the experimental one. You will need to use an external library or write your own logic to do that.

Loading experimental spectra

To load an experimental spectrum use `Tesliper.load_experimental()` method. You will need to provide a path to the file (absolute or relative to the current `Tesliper.input_dir`) and a genre name of the loaded experimental spectrum. When the file is read, its content is stored in `Tesliper.experimental` dictionary.

```
>>> spectrum = tslr.load_experimental("path/to/spectrum.xy", "ir")
>>> tslr.experimental["ir"] is spectrum
True
```

Adjusting calculated spectra

Spectra calculated and loaded from disk with `tesliper` are stored as instances of `Spectra` or `SingleSpectrum` classes. Both of them provide a `scale_to()` and `shift_to()` methods that adjust a scale and offset (respectively) to match another spectrum, provided as a parameter. Parameters found automatically may not be perfect, so you may provide them yourself, by manually setting `scaling` and `offset` to desired values.

```
>>> spectra = tslr.spectrum["ir"]
>>> spectra.scaling # affects spectra.y
1.0
>>> spectra.scale_to(tslr.experimental["ir"])
```

(continues on next page)

(continued from previous page)

```
>>> spectra.scaling
1.32
>>> spectra.offset = 50 # bathochromic shift, affects spectra.x
```

Corrected values may be accessed *via* `spectra.x` and `spectra.y`, original values may be accessed *via* `spectra.abscissa` and `spectra.values`.

3.4.5 Writing to disk

Once you have data you care about, either extracted or calculated, you most probably would like to store it, process it with another software, or visualize it. `tesliper` provides a way to save this data in one of the supported formats: CSV, human-readable text files, or .xlsx spreadsheet files. Moreover, `tesliper` may produce Gaussian input files, allowing you to easily setup a next step of calculations.

Exporting data

`tesliper` provides a convenient shorthands for exporting certain kinds of data:

- `Tesliper.export_energies()` will export information about conformers' energies and their calculated populations;
- `Tesliper.export_spectral_data()` will export data that is related to spectral activity, but cannot be used to simulate spectra;
- `Tesliper.export_activities()` will export unprocessed spectral activities, normally used to simulate spectra;
- `Tesliper.export_spectra()` will export spectra calculated so far that are stored in `Tesliper.spectra` dictionary;
- `Tesliper.export_averaged()` will export each averaged spectrum calculated so far that is stored in `Tesliper.averaged` dictionary.

Each of these methods take two parameters: *fnt* and *mode*. *fnt* is a file format, to which data should be exported. It should be one of the following values: "txt" (the default), "csv", "xlsx". *mode* denotes how files should be opened and should be one of: "a" (append to existing file), "x" (the default, only write if file doesn't exist yet), "w" (overwrite file if it already exists).

These export methods will usually produce a number of files in the `Tesliper.output_dir`, which names will be picked automatically, according to the genre of the exported data and/or the conformer that data relates to. `export_energies()` will produce a file for each available energies genre and an additional overview file, `export_spectra()` will create a file for spectra genre and each conformer, and so on.

Note: "xlsx" format is an exception from the above - it will produce only one file, named "tesliper-output.xlsx", and create multiple spreadsheets inside this file. The appending mode is useful when exporting data to "xlsx" format, as it allows to write multiple kinds of data (with calls to multiple of these methods) to this single destination .xlsx file.

There is also a `Tesliper.export_data()` available, which will export only genres you specifically request (plus "freq" or "waven", if any genre given genre is a spectral data-related). The same applies here in the context of output format, write mode, and names of produced files.

Tip: If you would like to customize names of the files produced, you will need to directly use one of the writer objects provided by `tesliper`. Refer to the [writing](#) module documentation for more information.

Creating input files

You can use `Tesliper.export_job_file()` to prepare input files for the quantum chemical calculations software. Apart from the typical *fmt* (only "gjf" is supported by default) and *mode* parameters, this method also accepts the *geometry_genre* and any number of additional keyword parameters, specifying calculations details. *geometry_genre* should be a name of the data genre, representing conformers' geometry, that should be used as input geometry. Additional keyword parameters are passed to the writer object, relevant to the *fmt* requested. Keywords supported by the default "gjf"-format writer are as follows:

- route** A calculations route: keywords specifying calculations directives for quantum chemical calculations software.
- link0** Dictionary with "link zero" commands, where each key is command's name and each value is this command's parameter.
- comment** Contents of title section, i.e. a comment about the calculations.
- post_spec** Anything that should be placed after conformer's geometry specification. Will be written to the file as given.

`link0` parameter should be explained in more details. It supports standard link zero commands used with Gaussian software, like `Mem`, `Chk`, or `NoSave`. Full list of these commands may be found in the documentation for `GjfWriter.link0`. Any non-parametric `link0` command (i.e. `Save`, `NoSave`, and `ErrorSave`), should be given a `True` value if it should be included in the `link0` section.

Path-like commands, e.g. `Chk` or `RWF`, may be parametrized for each conformer. You can put a placeholder inside a given string path, which will be substituted when writing to file. The most useful placeholders are probably `${conf}` and `${num}` that evaluate to conformer's name and ordinal number respectively. More information about placeholders may be found in `GjfWriter.make_name()` documentation.

```
>>> list(tslr.conformers.kept_keys())
["conf_one", "conf_three"]
>>> tslr.export_job_file(
...     geometry_genre="optimized_geom",
...     route="# td=(nstates=80)",
...     comment="Example of parametrization in .gjf files",
...     link0={
...         "Mem": "10MW",
...         "Chk": "path/to/${conf}.chk",
...         "Save": True,
...     },
... )
>>> [file.name for file in tslr.output_dir.iterdir()]
["conf_one.gjf", "conf_three.gjf"]
```

Then contents of "conf_one.gjf" is:

```
%Mem=10MW
%Chk=path/to/conf_one.gjf
%Save
```

(continues on next page)

(continued from previous page)

```
# td=(nstates=80)
```

Example of parametrization in .gjf files

```
[geometry specification...]
```

3.4.6 Saving session for later

If you'd like to come back to the data currently contained within a `tesliper` instance, you may serialize it using `Tesliper.serialize()` method. Provide the method call with a `filename` parameter, under which filename the session should be stored inside the current `output_dir`. You may also omit it to use the default `".tslr"` name. All data, extracted and calculated, including current *kept* status of each conformer, is saved and may be loaded later using `Tesliper.load()` class method.

```
curr_dir = tslr.output_dir
tslr.serialize()
loaded = Teslaiper.load(curr_dir / ".tslr")
assert loaded.conformers == tslr.conformers
assert loaded.conformers.kept == tslr.conformers.kept
assert loaded.spectra.keys() == tslr.spectra.keys()
```

3.5 Advanced guide

`tesliper` handles data extracted from the source files in a form of specialized objects, called *data arrays*. These objects are instances of one of the `DataArray` subclasses (hence sometimes referenced as `DataArray`-like objects), described here in a greater detail. `DataArray` base class defines a basic interface and implements data validation, while its subclasses provided by `tesliper` define how certain data *genres* should be treated and processed.

Note: Under the hood, *data arrays*, and `tesliper` in general, use `numpy` to provide fast numeric operations on data.

This part of documentation also shows how to take more control over the data export. `Tesliper` automizes this process quite a bit and exposes only a limited set of possibilities provided by the underlying writer classes. It will be shown here how to use these writer classes directly in your code.

3.5.1 Data array classes

Each `DataArray`-like object has the following four attributes:

genre name of the data genre that *values* represent;

filenames sequence of conformers' identifiers as a `numpy.ndarray(dtype=str)`;

values sequence of values of *genre* data genre for each conformer in *filenames*. It is also a `numpy.ndarray`, but its `dtype` depends on the particular data array class;

allow_data_inconsistency a flag that controls the process of data validation. More about data inconsistency will be said later.

Some data arrays may provide more data. For example, any spectral data values wouldn't be complete without the information about the band that they corresponds to, so data arrays that handle this kind of data also provide a *frequencies* or *wavelengths* attribute.

Note: Attributes that hold a band information are actually *freq* and *wavelen* respectively, *frequencies* and *wavelengths* are convenience aliases.

Creating data arrays

The easiest way to instantiate the data array of desired data genre is to use `Conformers.arrayed()` factory method. It transforms it's stored data into the `DataArray`-like object associated with a particular data genre, ignoring any conformer that is *not kept* or doesn't provide data for the requested genre. You may force it to ignore any trimming applied by adding `full=True` to call parameters (conformers without data for requested genre still will be ignored). Moreover, any other keyword parameters provided will be forwarded to the class constructor, allowing you to override any default values.

```
>>> from tesliper import Conformers
>>> c = Conformers(
...     one={"gib":-123.5},
...     two={},
...     three={"gib": -123.6},
...     four={"gib":-123.7}
... )
>>> c.kept = ["one", "three"]
>>> c.arrayed("gib")
Energies(genre='gib', filenames=['one' 'three'], values=[-123.5 -123.6], t=298.15)
>>> c.arrayed("gib", full=True)
Energies(genre='gib', filenames=['one' 'three' 'four'], values=[-123.5 -123.6 -123.7], t=298.15)
>>> c.arrayed("gib", t=1111)
Energies(genre='gib', filenames=['one' 'three'], values=[-123.5 -123.6], t=1111)
```

You can also instantiate any data array directly, providing data by yourself.

```
>>> from tesliper import Energies
>>> Energies(
...     genre='gib',
...     filenames=['one' 'three'],
...     values=[-123.5 -123.6]
... )
Energies(genre='gib', filenames=['one' 'three'], values=[-123.5 -123.6], t=298.15)
```

Data validation

On instantiation of a data array class, *values* provided to its constructor are transformed to the `numpy.ndarray` of the appropriate type. If this cannot be done due to the incompatibility of type of *values* elements and data array's `dtype`, an exception is raised. However, `tesliper` will try to convert given values to the target type, if possible.

```
>>> from tesliper import IntegerArray
>>> arr = IntegerArray(genre="example", filenames=["one"], values=["1"])
>>> arr
IntegerArray(genre="example", filenames=["one"], values=[1])
>>> type(arr.values)
<class 'numpy.ndarray'>

>>> IntegerArray(genre="example", filenames=["one"], values=["1.0"])
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: '1.0'

>>> IntegerArray(genre="example", filenames=["one"], values=[None])
Traceback (most recent call last):
...
TypeError: int() argument must be a string, a bytes-like object or a number, not
↳ 'NoneType'
```

Also *values* size is checked: its first dimension must be of the same size, as the number of entries in the *filenames*, otherwise `ValueError` is raised.

```
>>> IntegerArray(genre="example", filenames=["one"], values=[1, 2])
Traceback (most recent call last):
...
ValueError: values and filenames must have the same shape up to 1 dimensions. Arrays of_
↳ shape (2,) and (1,) were given.
```

`InconsistentDataError` exception is raised when *values* are multidimensional, but provide uneven number of entries for each conformer (*values* are a jagged array).

```
>>> IntegerArray(genre="example", filenames=["one", "two"], values=[[1, 2], [3]])
Traceback (most recent call last):
...
InconsistentDataError: IntegerArray of example genre with unequal number of values for_
↳ conformer requested.
```

This behavior may be suppressed, if the instance is initiated with `allow_data_inconsistency=True` keyword parameter. In such case no exception is raised if numbers of entries doesn't match, and jagged arrays will be turned into `numpy.ma.masked_array` instead of `numpy.ndarray`, if it is possible.

```
>>> IntegerArray(
...     genre="example",
...     filenames=["one"],
...     values=[1, 2],
...     allow_data_inconsistency=True
... )
IntegerArray(genre="genre", filenames=["one"], values=[1,2], allow_data_
↳ inconsistency=True)
```

(continues on next page)

(continued from previous page)

```
>>> IntegerArray(
...     genre="example",
...     filenames=["one", "two"],
...     values=[[1, 2], [3]],
...     allow_data_inconsistency=True
... )
IntegerArray(genre='genre', filenames=['one' 'two'], values=[[1 2]
[3 --]], allow_data_inconsistency=True)
```

Some data array classes validate also other data provided to its constructor, e.g. [Geometry](#) checks if *atoms* provides an atom specification for each atom in the conformer.

Note: Each validated field is actually a [ArrayProperty](#) or its subclass under the hood, which provides the validation mechanism.

Available data arrays

Data arrays provided by `tesliper` are listed below in categories, along with a short description and with a list of data genres that are associated with a particular data array class. More information about a `DataArray`-like class of interest may be learn in the [API reference](#).

Generic types

Simple data arrays, that hold a data of particular type. They do not provide any functionality beside initial data validation. They are used by `tesliper` for segregation of simple data an as a base classes for other data arrays (concerns mostly [FloatArray](#)).

[IntegerArray](#) For handling data of `int` type.

Table 4: Genres associated with this class:

charge	multiplicity
--------	--------------

[FloatArray](#) For handling data of `float` type.

Table 5: Genres associated with this class:

zpecorr	tencorr	entcorr	gibcorr
---------	---------	---------	---------

[BooleanArray](#) For handling data of `bool` type.

Table 6: Genres associated with this class:

normal_termination	optimization_completed
--------------------	------------------------

[InfoArray](#) For handling data of `str` type.

Table 7: Genres associated with this class:

command	stoichiometry
---------	---------------

Spectral data

Each data array in this category provides a *freq* or *wavelen* attribute, also accessible by their convenience aliases *frequencies* and *wavelengths*. These attributes store an information about frequency or wavelength that the particular spectral value is associated with (x-axis value of the center of the band).

Activities genres, that are the genres that may be used to simulate the spectrum, also provide a *calculate_spectra()* method for this purpose (see *VibrationalActivities.calculate_spectra()*, *ScatteringActivities.calculate_spectra()*, and *ElectronicActivities.calculate_spectra()*), as well as a *intensities* property that calculates a theoretical intensity for each activity value. A convince *spectra_name* property may be used to get the name of spectra pseudo-genre calculated with particular activities genre.

```
>>> act = c["dip"]
>>> act.spectra_name
"ir"
>>> from tesliper import lorentzan
>>> spc = act.calculate_spectra(
...     start=200, # cm-1
...     stop=1800, # cm-1
...     step=1, # cm-1
...     width=5, # cm-1
...     fitting=lorentzan
... )
>>> type(spc), spc.genre
(<class 'tesliper.glassware.spectra.Spectra'>, 'ir')
```

VibrationalData For handling vibrational (IR and VCD related) data that is not a spectral activity.

Table 8: Genres associated with this class:

mass	frc	emang
------	-----	-------

ScatteringData For handling scattering (Raman and ROA related) data that is not a spectral activity.

Table 9: Genres associated with this class:

depolarp	depolaru	depp	depu	alpha2
beta2	alphag	gamma2	delta2	cid1
cid2	cid3	rc180		

ElectronicData For handling electronic (UV and ECD related) data that is not a spectral activity.

Table 10: Genres associated with this class:

eemang

VibrationalActivities For handling vibrational (IR and VCD related) spectral activity data.

Table 11: Genres associated with this class:

iri	dip	rot
-----	-----	-----

ScatteringActivities For handling scattering (Raman and ROA related) spectral activity data.

Table 12: Genres associated with this class:

ramanactiv	ramact	raman1	roa1
raman2	roa2	raman3	roa3

ElectronicActivities For handling electronic (UV and ECD related) spectral activity data.

Table 13: Genres associated with this class:

vdip	ldip	vrot	lrot	vosc	losc
------	------	------	------	------	------

Other data arrays

FileNamesArray Special case of *DataArray*, holds only filenames. *values* property returns same as *filenames* and ignores any value given to its setter. The only genre associated with this class is *filenames* pseudo-genre.

Bands Special kind of data array for band values, to which spectral data or activities correspond. Provides an easy way to convert values between their different representations: frequency, wavelength, and excitation energy. Also allows to easily locate conformers with imaginary frequencies.

```
>>> arr = Bands(
...     genre="freq",
...     filenames=["one", "two", "three"],
...     values=[[-15, -10, 105], [30, 123, 202], [-100, 12, 165]]
... )
>>> arr.imaginary
array([2, 0, 1])
>>> arr.find_imaginary()
{'one': 2, 'three': 1}
```

Table 14: Genres associated with this class:

freq	wavelen	ex_en
------	---------	-------

Energies For handling data about the energy of conformers. Provides an easy way of calculating Boltzmann distribution-based population of conformers *via* a *populations* property.

```
>>> arr = Energies(
...     genre="gib",
...     filenames=["one", "two", "three"],
...     values=[-123.505977, -123.505424, -123.506271]
... )
>>> arr.deltas # difference from lowest energy in kcal/mol
array([0.18448779, 0.53150055, 0.          ])
>>> arr.populations
array([0.34222796, 0.19052561, 0.46724643])
```

Table 15: Genres associated with this class:

scf	zpe	ten	ent	gib
-----	-----	-----	-----	-----

Transitions For handling information about electronic transitions from ground to excited state contributing to each band.

Data is stored in three attributes: *ground*, *excited*, and *values*, which are respectively: list of ground state electronic subshells, list of excited state electronic subshells, and list of coefficients of transitions from corresponding ground to excited subshell. Each of these arrays is of shape (conformers, bands, max_transitions), where 'max_transitions' is a highest number of transitions contributing to single band across all bands of all conformers.

Allows to easily calculate contribution of each transition using *contribution* and to find which transition contributes the most to the particular transition with *highest_contribution*.

Table 16: Genres associated with this class:

transitions

Geometry For handling information about geometry of conformers.

Table 17: Genres associated with this class:

last_read_geom	input_geom	optimized_geom
----------------	------------	----------------

3.5.2 Writing to disk

Teslipper object provides an easy, but not necessarily a flexible way of writing calculated and extracted data to disk. If your process requires more flexibility in this matter, you may use *tesliers* writer objects directly. This will allow you to adjust how generated files are named and will give you more control over what is exported.

Writer classes

A writer object may be created using a *writer()* factory function. It expects a string parameter, that specifies a desired format for data export. *teslipper* provides writers for "txt", "csv", "xlsx", and "gjf" file formats. The second mandatory parameter is a *destination*: the (existing) directory to which files should be written. Just like writing methods of *Teslipper* object, the function also takes a *mode* parameter that defines what should happen if any file already exists. Any additional keyword parameters are forwarded to the writer object constructor.

```
>>> from teslipper import writer
>>> wrt = writer("txt", "/path/to/dir")
>>> type(wrt)
<class 'teslipper.writing.txt_writer.TxtWriter'>

>>> wrt = writer("txt", "/doesnt/exists")
Traceback (most recent call last):
...
FileNotFoundError: Given destination doesn't exist or is not a directory.
```

Note: *writer()* factory function is used by *teslipper* mostly to provide a dynamic access to the writer class most recently registered (on class definition) to handle a particular format. This is useful when you modify an existing writer class or provide a new one.

You can also create any of the writer objects directly, by importing and instantiating its class. The four available writer classes are listed below with a short comment. For more information on which methods they implement and how to use them, refer to the relevant API documentation.

```
from tesliper import TxtWriter
wrt = TxtWriter(destination="/path/to/dir")
```

TxtWriter Generates human-readable text files.

CsvWriter Generates files in CSV format with optional headers. Allows for the same level of output format customization as Python's `csv.writer` (supports specification of *dialect* and other formatting parameters).

XlsxWriter Instead of generating multiple files, creates a single .xlsx file and a variable number of spreadsheets inside it.

Gjfwriter Allows to create input files for new calculation job in Gaussian software.

write() and other methods

Writer objects expect data they receive to be a *DataArray*-like instances. Each writer object provides a *write()* method for writing arbitrary data arrays to disk. This method dispatches received data arrays to appropriate writing methods, based on their type. You are free to use either *write()* for easily writing a number of data genres in batch, or other methods for more control. The table below lists these methods, along with a brief description and *DataArray*-like object, for which the method will be called by writer's *write()* method.

Table 18: Methods used to write certain data

Writer's Method	Description	Supported arrays	Created files
<i>generic()</i>	Generic data: any genre that provides one value for each conformer.	<i>DataArray</i> , <i>IntegerArray</i> , <i>FloatArray</i> , <i>BooleanArray</i> , <i>InfoArray</i> .	one
<i>overview()</i>	General information about conformers: energies, imaginary frequencies, stoichiometry.	<i>Energies</i>	one
<i>energies()</i>	Detailed information about conformers' relative energy, including calculated populations	<i>Energies</i>	for each genre
<i>single_spectrum()</i>	A spectrum - calculated for single conformer or averaged.	<i>SingleSpectrum</i>	one
<i>spectral_data()</i>	Data related to spectral activity, but not convertible to spectra.	<i>VibrationalData</i> , <i>ScatteringData</i> , <i>ElectronicData</i>	for each conformer
<i>spectral_activities()</i>	Data that may be used to simulate conformers' spectra.	<i>VibrationalActivities</i> , <i>ScatteringActivities</i> , <i>ElectronicActivities</i>	for each conformer
<i>spectra()</i>	Spectra for multiple conformers.	<i>Spectra</i>	for each conformer
<i>transitions()</i>	Electronic transitions from ground to excited state, contributing to each band.	<i>Transitions</i>	for each conformer
<i>geometry()</i>	Geometry (positions of atoms in space) of conformers.	<i>Geometry</i>	for each conformer

spectral_data() and *spectral_activities()* methods need some clarification. They will create one file for each conformer in given data arrays, with data from each provided data array joined in the conformer's file. It's im-

portant to remember, that only *values* from each data array are displayed, contrary to band values, which are displayed only once, as provided with the *band* parameter. Consequently, mixing vibrational and scattering data with a custom *name_template* is fine, but mixing either of those with electronic data in a single call is not possible.

Warning: You need to make sure that data contained in `DataArray`-like objects cover the same set of conformers, when passing multiple data array objects to the `write()` method or any other writing method. Passing two data arrays with data for different sets of conformers may produce files with corrupted data or fail silently. `Conformers.trim_incomplete()` trimming method may be helpful in preventing such fails.

Not all writer objects implement each of these writing methods, e.g. `GjfWriter`, that allows to create Gaussian input files, only implements `geometry()` method (because export of, e.g. a calculated spectrum as a Gaussian input would be pointless). Trying to `write()` a data array that should be written by a method that is not implemented, or calling such method directly, will raise a `NotImplementedError`.

Naming files

Usually, calling any of writing methods will produce multiple files in the *destination* directory: one for each given genre, each conformer, etc. `tesliper` provides a reasonable naming scheme for these files, but you can modify it, by providing your own *name_templates* in place of the default ones. To do this you will need to call desired writing methods directly, instead of using `write()`.

Each writing method uses a value of *name_template* parameter given to the method call to create a filename for each file it generates. *name_template* should be a string that contains (zero, one, or more) label identifiers in form of `${identifier}`. These identifiers will be substituted to produce a final filename. Available identifiers and their meaning are as follows:

- `${ext}` - appropriate file extension;
- `${conf}` - name of the conformer;
- `${num}` - number of the file according to internal counter;
- `${genre}` - genre of exported data;
- `${cat}` - category of produced output;
- `${det}` - category-specific detail.

The `${ext}` identifier is filled with the value of `Writers.extension` attribute, which value is also used to identify a writer class: "txt", "csv", etc. Other values are provided by the particular writing method.

```
from tesliper import Tesliper, writer
tslr = Tesliper(input_dir="/project/input")
... # data extracted and trimmed
# tslr.conformers.kept_keys() == {"conf_one", "conf_four"}
freq, dip, rot = tslr["freq"], tslr["dip"], tslr["rot"]
wrt = writer("txt", "/project/default")
wrt.spectral_activities(band=freq, data=[dip, rot])
wrt = writer("txt", "/project/custom")
wrt.spectral_activities(
    band=freq, data=[dip, rot],
    name_template="name_${num}_${genre}.xy"
)
```

Listing 1: contents of /project

```
.
├── input
│   └── ...
├── default
│   ├── conf_one.activities-vibrational.txt
│   └── conf_four.activities-vibrational.txt
└── custom
    ├── name_1_freq.xy
    └── name_2_freq.xy
```

3.6 Available data genres

freq *list of floats*; available from freq job

harmonic vibrational frequencies (cm^{-1})

mass *list of floats*; available from freq job

reduced masses (AMU)

fre *list of floats*; available from freq job

force constants (mDyne/\AA)

iri *list of floats*; available from freq job

IR intensities (KM/mole)

dip *list of floats*; available from freq=VCD job

dipole strengths ($10^{-40} \text{esu}^2\text{-cm}^2$)

rot *list of floats*; available from freq=VCD job

rotational strengths ($10^{-44} \text{esu}^2\text{-cm}^2$)

emang *list of floats*; available from freq=VCD job

E-M angle = Angle between electric and magnetic dipole transition moments (deg)

depolarp *list of floats*; available from freq=Raman job

depolarization ratios for plane incident light

depolaru *list of floats*; available from freq=Raman job

depolarization ratios for unpolarized incident light

ramanactiv *list of floats*; available from freq=Raman job

Raman scattering activities ($\text{\AA}^4/\text{AMU}$)

ramact *list of floats*; available from freq=ROA job

Raman scattering activities ($\text{\AA}^4/\text{AMU}$)

depp *list of floats*; available from freq=ROA job

depolarization ratios for plane incident light

depu *list of floats*; available from freq=ROA job
 depolarization ratios for unpolarized incident light

alpha2 *list of floats*; available from freq=ROA job
 Raman invariants $\text{Alpha2} = \alpha^2 (A^4/\text{AMU})$

beta2 *list of floats*; available from freq=ROA job
 Raman invariants $\text{Beta2} = \beta(\alpha)^2 (A^4/\text{AMU})$

alphag *list of floats*; available from freq=ROA job
 ROA invariants $\text{AlphaG} = \alpha G' (10^4 A^5/\text{AMU})$

gamma2 *list of floats*; available from freq=ROA job
 ROA invariants $\text{Gamma2} = \beta(G')^2 (10^4 A^5/\text{AMU})$

delta2 *list of floats*; available from freq=ROA job
 ROA invariants $\text{Delta2} = \beta(A)^2, (10^4 A^5/\text{AMU})$

raman1 *list of floats*; available from freq=ROA job
 Far-From-Resonance Raman intensities =ICPu/SCPu(180) (K)

roa1 *list of floats*; available from freq=ROA job
 ROA intensities =ICPu/SCPu(180) (10^4 K)

cid1 *list of floats*; available from freq=ROA job
 $\text{CID} = (\text{ROA}/\text{Raman}) * 10^4 = \text{ICPu}/\text{SCPu}(180)$

raman2 *list of floats*; available from freq=ROA job
 Far-From-Resonance Raman intensities =ICPd/SCPd(90) (K)

roa2 *list of floats*; available from freq=ROA job
 ROA intensities =ICPd/SCPd(90) (10^4 K)

cid2 *list of floats*; available from freq=ROA job
 $\text{CID} = (\text{ROA}/\text{Raman}) * 10^4 = \text{ICPd}/\text{SCPd}(90)$

raman3 *list of floats*; available from freq=ROA job
 Far-From-Resonance Raman intensities =DCPI(180) (K)

roa3 *list of floats*; available from freq=ROA job
 ROA intensities =DCPI(180) (10^4 K)

cid3 *list of floats*; available from freq=ROA job
 $\text{CID} = (\text{ROA}/\text{Raman}) * 10^4 = \text{DCPI}(180)$

rc180 *list of floats*; available from freq=ROA job
 RC180 = degree of circularity

wavelen *list of floats*; available from td job
 excitation energies (nm)

ex_en *list of floats*; available from td job
 excitation energies (eV)

eemang *list of floats*; available from td job
E-M angle = Angle between electric and magnetic dipole transition moments (deg)

vdip *list of floats*; available from td job
dipole strengths (velocity)

ldip *list of floats*; available from td job
dipole strengths (length)

vrot *list of floats*; available from td job
rotatory strengths (velocity) in cgs (10^{-40} erg-esu-cm/Gauss)

lrot *list of floats*; available from td job
rotatory strengths (length) in cgs (10^{-40} erg-esu-cm/Gauss)

vosc *list of floats*; available from td job
oscillator strengths

losc *list of floats*; available from td job
oscillator strengths

transitions *list of lists of lists of (int, int, float)*; available from td job
transitions (first to second) and their coefficients (third)

scf *float*; always available
SCF energy

zpe *float*; available from freq job
Sum of electronic and zero-point Energies (Hartree/Particle)

ten *float*; available from freq job
Sum of electronic and thermal Energies (Hartree/Particle)

ent *float*; available from freq job
Sum of electronic and thermal Enthalpies (Hartree/Particle)

gib *float*; available from freq job
Sum of electronic and thermal Free Energies (Hartree/Particle)

zpecorr *float*; available from freq job
Zero-point correction (Hartree/Particle)

tencorr *float*; available from freq job
Thermal correction to Energy (Hartree/Particle)

entcorr *float*; available from freq job
Thermal correction to Enthalpy (Hartree/Particle)

gibcorr *float*; available from freq job
Thermal correction to Gibbs Free Energy (Hartree/Particle)

command *str*; always available
command used for calculations

normal_termination *bool*; always available
 true if Gaussian job seem to exit normally, false otherwise

optimization_completed *bool*; available from opt job
 true if structure optimization was performed successfully

version *str*; always available
 version of Gaussian software used

charge *int*; always available
 molecule's charge

multiplicity *int*; always available
 molecule's spin multiplicity

input_atoms *list of str*; always available
 input atoms as a list of atoms' symbols

input_geom *list of lists of floats*; always available
 input geometry as X, Y, Z coordinates of atoms

stoichiometry *str*; always available
 molecule's stoichiometry

last_read_atoms *list of ints*; always available
 molecule's atoms as atomic numbers

last_read_geom *list of lists of floats*; always available
 molecule's geometry (last one found in file) as X, Y, Z coordinates of atoms

optimized_atoms *list of ints*; available from successful opt job
 molecule's atoms read from optimized geometry as atomic numbers

optimized_geom *list of lists of floats*; available from successful opt job
 optimized geometry as X, Y, Z coordinates of atoms

3.7 Math and Algorithms

3.7.1 Simulation of spectra

To simulate a spectrum, each band's theoretical signal intensity, derived from quantum chemical calculations of corresponding optical activity, must be expressed as a broadened peak, instead of the single scalar value. To simulate peak's shape one of the curve fitting functions is used. **tesliper** implements two, most commonly used, such functions: gaussian function¹ and lorentzian function².

For each point on the simulated spectrum's abscissa, the corresponding signal intensity is calculated by applying the fitting function to all bands of the conformer and summing resulting values.

¹ <https://mathworld.wolfram.com/GaussianFunction.html>

² <https://mathworld.wolfram.com/LorentzianFunction.html>

Gaussian fitting function

$$f(\nu) = \frac{1}{\sigma\sqrt{2\pi}} \sum_i I_i e^{-(\nu_i - \nu)^2 / (2\sigma^2)}$$

ν Arbitrary point on the x -axis, for which the signal intensity is calculated.

ν_i Point on the x -axis, at which the i^{th} band occur.

I_i Intensity of the i^{th} band.

$\sigma = \sqrt{2}\omega$ Standard derivation, in this context interpreted as equal to $\sqrt{2}$ times ω .

ω Half width of the peak at $\frac{1}{e}$ of its maximum value (HW1OeM), expressed in the x -axis units.

Lorentzian fitting function

$$f(\nu) = \frac{\gamma}{\pi} \sum_i \frac{I_i}{(\nu_i - \nu)^2 + \gamma^2}$$

ν Arbitrary point on the x -axis, for which the signal intensity is calculated.

ν_i Point on the x -axis, at which the i^{th} band occur.

I_i Intensity of the i^{th} band.

γ Half width of the peak at half of its maximum value (HWHM), expressed in the x -axis units.

3.7.2 Calculation of intensities

Dipole strength and rotator strength is converted to the theoretical intensity as described by Polavarapu³. Constants used in below equations are as follows.

$c = 2.99792458 \times 10^{10} \text{ cm} \cdot \text{s}^{-1}$ Speed of light.

$h = 6.62606896 \times 10^{-30} \text{ kg} \cdot \text{cm}^2 \cdot \text{s}^{-1}$ Planck's constant.

$N_A = 6.02214199 \times 10^{23} \text{ mol}^{-1}$ Avogadro's constant.

$m_e = 9.10938 \times 10^{-28} \text{ g}$ Mass of the electron.

$e = 4.803204 \times 10^{-10} \text{ esu}$ The charge on the electron.

Dipole strength to IR intensities

$$I_k = \frac{100 \cdot 8\pi^3 N_A}{3 \cdot \ln(10) \cdot hc} D_k \nu_k = 0.010886 \cdot D_k \nu_k$$

$D_k [\times 10^{-40} \text{ esu}^2 \text{cm}^2]$ Dipole strength of k^{th} transition.

$\nu_k [\text{cm}^{-1}]$ Frequency of the k^{th} transition.

$I_k [\text{L mol}^{-1} \text{cm}^{-1}]$ Theoretical (“zero-width”) peak intensity in terms of the decadic molar absorption coefficient ϵ .

³ Prasad L. Polavarapu (2017), *Chiroptical Spectroscopy Fundamentals and Applications*, CRC Press

Rotator strength to VCD intensities

$$I_k = 4 \cdot \frac{8\pi^3 N_A}{3 \cdot \ln(10) \cdot hc} R_k \nu_k = 0.0435441 \cdot R_k \nu_k$$

R_k [$\times 10^{-44}$ esu²cm²] Rotator strength of k^{th} transition.

ν_k [cm⁻¹] Frequency of the k^{th} transition.

I_k [L mol⁻¹cm⁻¹] Theoretical (“zero-width”) peak intensity in terms of the decadic molar absorption coefficient ϵ .

Dipole strength to UV intensities

$$I_k = \frac{8\pi^3 N_A}{3 \cdot \ln(10) \cdot hc} D_k \lambda_k = 0.010886 \cdot D_k \lambda_k$$

D_k [$\times 10^{-40}$ esu²cm²] Dipole strength of k^{th} transition.

λ_k [cm⁻¹] Wavelength of the k^{th} transition.

I_k [L mol⁻¹cm⁻¹] Theoretical (“zero-width”) peak intensity in terms of the decadic molar absorption coefficient ϵ .

Rotator strength to ECD intensities

$$I_k = 4 \cdot \frac{8\pi^3 N_A}{3 \cdot \ln(10) \cdot hc} R_k \lambda_k = 0.0435441 \cdot R_k \lambda_k$$

R_k [$\times 10^{-44}$ esu²cm²] Rotator strength of k^{th} transition.

λ_k [cm⁻¹] Wavelength of the k^{th} transition.

I_k [L mol⁻¹cm⁻¹] Theoretical (“zero-width”) peak intensity in terms of the decadic molar absorption coefficient ϵ .

Oscillator strength to UV intensities

Conversion from oscillator strength to signal intensity of UV spectrum is calculated as described by Gaussian⁴.

$$I_k = \frac{e^2 N_A}{10^3 \ln(10) m_e c^2} f_k = 2.315351857 \times 10^8 f_k$$

f_k Oscillator strength of k^{th} transition.

I_k [L mol⁻¹cm⁻¹] Theoretical (“zero-width”) peak intensity in terms of the decadic molar absorption coefficient ϵ .

Raman/ROA intensities

Gaussian-provided Raman and ROA activities are used without any conversion.

⁴ <https://gaussian.com/uvvisplot/>

3.7.3 Population of conformers

Population of conformers is calculated according to the Boltzmann probability distribution that “gives the probability that a system will be in a certain state as a function of that state’s energy and the temperature of the system.”⁵ In this context each conformer is considered one of the possible states of the system (a studied molecule).

Firstly, we calculate a Boltzmann factors for each conformer in respect to the most stable conformer (the one of the lowest energy). Boltzmann factor of two states is defined as:

$$B_b^a = \frac{F(state_a)}{F(state_b)} = e^{(E_b - E_a)/kt}$$

where:

E_a and E_b energies of states a and b ;

$k = 0.0019872041$ kcal/(mol * K) Boltzmann constant;

t temperature of the system.

Boltzmann factor represents a ratio of probabilities of the two states being occupied. In other words, it shows how much more likely it is for the molecule to take the form of one conformer over another conformer. Having a ratio of these probabilities for each possible conformer in respect to the most stable conformer, we are able to find the distribution of conformers (probability of taking the form of each conformer):

$$p_i = \frac{B_0^i}{\sum_j B_0^j}$$

assuming that $state_0$ is the state of the lowest energy (the most stable conformer).

3.7.4 RMSD of conformers

RMSD, or root-mean-square deviation of atomic positions, is used as a measure of similarity between two conformers. As its name hints, it is an average distance between atoms in the two studied conformers: the lower the RMSD value, the more similar are conformers in question.

Finding minimized value of RMSD

In a typical output of the quantum chemical calculations software, molecule is represented by a number of points (mapping to particular atoms) in a 3-dimensional space. Usually, orientation and position of the molecule in the coordinate system is arbitrary and simple overlay of the two conformers may not be the same as their optimal overlap. To neglect the effect of conformers’ rotation and shift on the similarity measure, we will look for the common reference frame and optimal alignment of atoms.

Zero-centring atomic coordinates

To find the common reference frame for two conformers we move both to the origin of the coordinate system. This is done by calculating a centroid of a conformer and subtracting it from each point representing an atom. The centroid is given as an arithmetic mean of all the atoms in the conformer:

$$a_i^0 = a_i - \frac{1}{n} \sum_{j=a}^n a_j$$

where:

⁵ https://en.wikipedia.org/wiki/Boltzmann_distribution

a_i, a_j atom's original position in the coordinate system;

a_i^0 atom's centered position in the coordinate system;

n number of atoms in the molecule.

Rotating with Kabsch algorithm

Optimal rotation of one conformer onto another is achieved using a Kabsch algorithm⁶ (also known as Wahba's problem⁷). Interpreting positions of each conformers' atoms as a matrix, we find the covariance matrix H of these matrices (P and Q):

$$H = P^T Q$$

and then we use the singular value decomposition (SVD)⁸ routine to get U and V unitary matrices.

$$H = U \Sigma V^T$$

Having these, we can calculate the optimal rotation matrix as:

$$R = V \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & d \end{pmatrix} U^T$$

where $d = \text{sign}(\det(VU^T))$ that allows to ensure a right-handed coordinate system.

Note: To allow for calculation of the best rotation between sets of molecules and to compromise between efficiency and simplicity of implementation, tesliper uses Einstein summation convention⁹ via `numpy.einsum()` function. The implementation is as follows:

```
def kabsch_rotate(a: MoleculeOrList, b: MoleculeOrList) -> np.ndarray:
    """Minimize RMSD of conformers *a* and *b* by rotating molecule *a* onto *b*.
    Expects given representation of conformers to be zero-centered.
    Both *a* and *b* may be a single molecule or a set of conformers.

    Parameters
    -----
    a : [Sequence of ]Sequence of Sequence of float
        Set of points representing atoms, that will be rotated to best match reference.
    b : [Sequence of ]Sequence of Sequence of float
        Set of points representing atoms of the reference molecule.
    """
    # this approach is probably not the most efficient
    # but lets us easily perform a matrix multiplication on stacks of matrices
    a, b = np.asanyarray(a), np.asanyarray(b)
    # calculate covariance matrix for each stacked set of points
    # for each of stacked sets of points, equivalent of:
    # >>> cov = a.T @ b
    cov = np.einsum("...ji,...jk", a, b)
    u, s, vh = np.linalg.svd(cov) # singular value decomposition
```

(continues on next page)

⁶ https://en.wikipedia.org/wiki/Kabsch_algorithm

⁷ https://en.wikipedia.org/wiki/Wahba%27s_problem

⁸ https://en.wikipedia.org/wiki/Singular_value_decomposition

⁹ https://en.wikipedia.org/wiki/Einstein_notation

(continued from previous page)

```

# if determinant is negative, swap to ensure right-handed coordinate system
det = np.linalg.det(vh @ u) # works with stacked matrices
# don't introduce new dimension if not necessary
shape = (det.size, 3, 3) if det.size > 1 else (3, 3)
swap = np.zeros(shape)
swap[..., np.arange(2), np.arange(2)] = 1
swap[..., -1, -1] = np.sign(det)
# calculate optimally rotated set/s of points `a`
# for each of stacked sets of points, equivalent of
# >>> rotated = a @ u @ swap @ vh
# where u @ swap @ vh is rotation matrix
return np.einsum("...ij,...jk,...kl,...lm", a, u, swap, vh)

```

Calculating RMSD of atomic positions

Once conformers are aligned, the value of RMSD¹⁰ is calculated simply by finding a distance between each equivalent atoms and averaging their squares and finding the root of this average:

$$\text{RMSD} = \sqrt{\frac{1}{n} \sum_i^n (p_i - q_i)^2}$$

where:

p_i and q_i positions of i^{th} equivalent atoms in conformers P and Q ;

n number of atoms in each conformer.

Comparing conformers

To compare conformers as efficiently as possible, the RMSD values are calculated not in the each-to-each scheme, but inside a rather small moving window. The size of this window determines how many calculations will be done for the whole collection.

Moving window mechanism

tesliper provides three types of moving windows: a *fixed*, *stretching*, and *pyramid* windows. The strategy you choose will affect both the performance and the accuracy of the RMSD sieve, as described below.

fixed The most basic sliding window of a fixed size. Provides the most control over the performance of the sieve, but is the least accurate.

stretching The default, allows to specify the size of the window in the context of some numeric property, usually the energy of conformers. The size may differ in the sense of the number of conformers in each window, but the difference between maximum and minimum values of said property inside a window will not be bigger than the given *size*. Provides a best compromise between the performance and the accuracy.

pyramid The first window will contain the whole collection and each consecutive window will be smaller by one conformer. Allows to perform a each-to-each comparison, but in logarithmic time rather than quadratic time. Best accuracy but worst performance.

¹⁰ https://en.wikipedia.org/wiki/Root-mean-square_deviation_of_atomic_positions

Note: The actual windows produced by sliding window functions are iterables of `numpy.ndarrays` of indices (that point to the value in the original array of conformers).

The sieve

The *RMSD sieve function* takes care of zero-centring and finding the best overlap of the conformers, as described previously. Aside from this, it works as follows: for each window, provided by one of the moving window functions described above, it takes the first conformer in the window (reference) and calculates its minimum RMSD value with respect to each of the other conformers in this window. Any conformers that have lower RMSD value than a given threshold, will be considered identical to the reference conformer and internally marked as *not kept*. The sieve returns an array of boolean value for each conformer: `True` if conformer's structure is "original" and should be kept, `False` if it is a duplicate of other, "original" structure (at least according to threshold given), and should be discarded.

3.7.5 Spectra transformation

Finding best shift

Optimal offset of two spectra is determined by calculating their cross-correlation¹¹ (understood as in the signal processing context) and finding its maximum value. Index of this max value of the discrete cross-correlation array indicates the position of one spectrum in respect to the other spectrum, in which the overlap of the two is the greatest.

Finding optimal scaling

Optimal scaling factor of spectra is determined by comparing a mean y values of target spectrum and a reference spectrum. Values lower than 1% of maximum absolute y value of each spectrum are ignored.

3.8 tesliper

Modules

<code>tesliper.datawork</code>	All core algorithms for data handling live here, along with some helpers.
<code>tesliper.exceptions</code>	Project-specific errors.
<code>tesliper.extraction</code>	Classes for reading and parsing files.
<code>tesliper.glassware</code>	Data containers.
<code>tesliper.tesliper</code>	Provides a facade-like interface for easy access to tesliper's functionality.
<code>tesliper.writing</code>	Objects for data serialization.

¹¹ <https://en.wikipedia.org/wiki/Cross-correlation>

3.8.1 tesliper.datawork

All core algorithms for data handling live here, along with some helpers.

This package implements functions used by `tesliper` to provide its core functionality: comparing conformers, calculating spectra, averaging them, etc. It is divided into few modules based on a field that given function is related to.

Modules

<code>tesliper.datawork.atoms</code>	Converters between string and integer representations of atoms.
<code>tesliper.datawork.energies</code>	Functions for calculating populations of conformers based on their relative energies using Boltzmann distribution.
<code>tesliper.datawork.geometry</code>	Conformers geometry-related functions, primarily an RMSD sieve implementation.
<code>tesliper.datawork.intensities</code>	Optical activity to signal intensity converters.
<code>tesliper.datawork.spectra</code>	Functions that deal with spectra and spectral data.

tesliper.datawork.atoms

Converters between string and integer representations of atoms.

Functions

<code>atomic_number(element)</code>	Returns atomic number of given element.
<code>symbol_of_element(element)</code>	Returns symbol of given element.
<code>validate_atoms(atoms)</code>	Checks if given <i>atoms</i> represent a list of valid atom identifiers (symbols or atomic numbers).

Classes

<code>Atom(value)</code>	An enumeration that maps symbols of atoms to respective atomic numbers.
--------------------------	---

class `tesliper.datawork.atoms.Atom(value)`

An enumeration that maps symbols of atoms to respective atomic numbers.

This enumeration is introduced for your convenience: whenever you need to reference an atom by its atomic number, you may use appropriate symbol-value of this Enum instead. Providing e.g. `Atom.Au` rather than an integer 79 for Au's atomic number is probably a bit easier and definitely more readable.

`tesliper.datawork.atoms.symbol_of_element(element: Union[int, str]) → str`

Returns symbol of given element. If *element* is a symbol of an element already, it is capitalized and returned (so input's letters case doesn't matter).

Parameters `element (int or str)` – element's atomic number

Returns symbol of an element

Return type str

Raises

- **ValueError** – when *element* is not a whole number or cannot be converted to integer
- **TypeError** – if *element* cannot be interpreted as integer
- **InvalidElementError** – if *element* is not an atomic number of any known element

`tesliper.datawork.atoms.atomic_number(element: Union[int, str]) → int`

Returns atomic number of given element. If *element* is an atomic number already, it is returned without change.

Parameters *element* (*str* or *int*) – element’s symbol or atomic number (letters case doesn’t matter if string is given)

Returns atomic number of an element

Return type int

Raises

- **InvalidElementError** – when *element* cannot be converted to element’s atomic number
- **TypeError** – if *element* cannot be interpreted as integer or string

`tesliper.datawork.atoms.validate_atoms(atoms: Union[int, str, List[Union[str, int]]]) → List[int]`

Checks if given *atoms* represent a list of valid atom identifiers (symbols or atomic numbers). Returns list of atomic numbers of those atoms if it does or rises an exception if it doesn’t.

Parameters *atoms* (*int*, *str* or *iterable of int or str*) – Atoms to validate. Atoms as space-separated string are also accepted.

Returns List of given atoms’ atomic numbers.

Return type list of int

Raises **InvalidElementError** – if *atoms* cannot be interpreted as list of atoms’ identifiers

tesliper.datawork.energies

Functions for calculating populations of conformers based on their relative energies using Boltzmann distribution.

Module Attributes

<code>BOLTZMANN</code>	Value of Boltzmann constant in kcal/(mol*K).
<code>HARTREE_TO_KCAL_PER_MOL</code>	Multiply by this factor to convert from Hartree/mol to kcal/mol.

Functions

<code>calculate_deltas(energies)</code>	Calculates energy difference between each conformer and lowest energy conformer.
<code>calculate_min_factors(energies[, t])</code>	Calculates list of conformers' Boltzmann factors respective to lowest
<code>calculate_populations(energies[, t])</code>	Calculates Boltzmann distribution of conformers of given energies.

`tesliper.datawork.energies.BOLTZMANN = 0.0019872041`

Value of Boltzmann constant in kcal/(mol*K).

`tesliper.datawork.energies.HARTREE_TO_KCAL_PER_MOL = 627.5095`

Multiply by this factor to convert from Hartree/mol to kcal/mol.

`tesliper.datawork.energies.calculate_deltas(energies)`

Calculates energy difference between each conformer and lowest energy conformer.

Parameters **energies** (*numpy.ndarray or iterable of float*) – List of conformers energies.

Returns List of energy differences from lowest energy.

Return type `numpy.ndarray`

`tesliper.datawork.energies.calculate_min_factors(energies, t=298.15)`

Calculates list of conformers' Boltzmann factors respective to lowest energy conformer in system of given temperature.

Boltzmann factor of two states is defined as:

$$\frac{F(\text{state}_1)}{F(\text{state}_2)} = e^{\frac{E_2 - E_1}{kt}}$$

where E_1 and E_2 are energies of states 1 and 2, k is Boltzmann constant, $k = 0.0019872041 \text{ kcal}/(\text{mol} * \text{K})$, and t is temperature of the system.

energies [`numpy.ndarray` or `iterable`] List of conformers energies in kcal/mol units.

t [`float`, optional] Temperature of the system in K, defaults to 298,15 K.

numpy.ndarray List of conformers' Boltzmann factors respective to lowest energy conformer.

`tesliper.datawork.energies.calculate_populations(energies, t=298.15)`

Calculates Boltzmann distribution of conformers of given energies.

Parameters

- **energies** (*numpy.ndarray or iterable*) – List of conformers energies in kcal/mol units.
- **t** (*float, optional*) – Temperature of the system in K, defaults to 298,15 K.

Returns List of conformers populations calculated as Boltzmann distribution.

Return type `numpy.ndarray`

tesliper.datawork.geometry

Conformers geometry-related functions, primarily an RMSD sieve implementation.

This module provides an implementation of RMSD sieve, allowing for easy mathematical comparison of conformers' geometry and filtering out similar ones, based on user-provided "threshold of similarity".

Functions

<code>calc_rmsd(a, b)</code>	Compute RMSD (round-mean-square deviation) of two conformers (or sets of them).
<code>center(a)</code>	Zero-center all given conformers by subtracting their centroids.
<code>drop_atoms(values, atoms, discarded)</code>	Filters given values, returning those corresponding to atoms not specified as discarded.
<code>find_atoms(atoms, find[, reverse])</code>	Get indices of wanted atoms.
<code>fixed_windows(series, size)</code>	Simple, vectorized implementation of basic sliding window.
<code>get_triangular(m)</code>	Find <i>m</i> th triangular number.
<code>get_triangular_base(n)</code>	Find which <i>m</i> th triangular number <i>n</i> is.
<code>is_triangular(n)</code>	Checks if number <i>n</i> is triangular.
<code>kabsch_rotate(a, b)</code>	Minimize RMSD of conformers <i>a</i> and <i>b</i> by rotating molecule <i>a</i> onto <i>b</i> .
<code>pyramid_windows(series)</code>	Produces windows of shrinking sizes, from full sequence to last element only.
<code>rmsd_sieve(geometry, windows[, threshold])</code>	Compare conformers' geometry to keep only those that differ at least by a given threshold.
<code>select_atoms(values, indices)</code>	Filter given values to contain values only corresponding to atoms on given indices.
<code>stretching_windows(values, size[, ...])</code>	Implements a sliding window of a variable size, where values in each window are at most <i>size</i> bigger than the lowest value in given window.
<code>take_atoms(values, atoms, wanted)</code>	Filters given values, returning those corresponding to atoms specified as <i>wanted</i> .

`tesliper.datawork.geometry.find_atoms(atoms: Union[Sequence[int], numpy.ndarray], find: Union[int, Iterable[int], numpy.ndarray], reverse: bool = False) → numpy.ndarray`

Get indices of wanted atoms.

Parameters

- **atoms** (*Sequence of int or numpy.ndarray*) – List of atoms represented by their atomic numbers.
- **find** (*int, Sequence of int, or numpy.ndarray*) – Element or list of elements, represented by their atomic numbers, which indices should be find in *atoms* array.
- **reverse** (*bool*) – If True, indices of atoms NOT specified in *find* will be returned.

Returns Indices of found elements.

Return type numpy.ndarray

`tesliper.datawork.geometry.select_atoms(values: Union[Sequence, numpy.ndarray], indices: Union[Sequence[int], numpy.ndarray]) → numpy.ndarray`

Filter given values to contain values only corresponding to atoms on given indices. Recognizes if given values are a list of values for one or for many conformers, but it must be in shape (A, N) or (C, A, N) respectively.

`tesliper.datawork.geometry.take_atoms(values: Union[Sequence, numpy.ndarray], atoms: Union[Sequence[int], numpy.ndarray], wanted: Union[int, Iterable[int], numpy.ndarray]) → numpy.ndarray`

Filters given values, returning those corresponding to atoms specified as *wanted*. Roughly equivalent to: `>>> numpy.take(values, numpy.nonzero(numpy.equal(atoms, wanted))[0], 1)` but returns empty array, if no atom in *atoms* matches *wanted* atom. If *wanted* is list of elements, `numpy.isin` is used instead of `numpy.equal`.

Parameters

- **values** (*Sequence or numpy.ndarray*) – array of values; it should be one-dimensional list of values or n-dimensional array of shape (conformers, values[, coordinates[, other]])
- **atoms** (*Sequence of int or numpy.ndarray*) – list of atoms in molecule, given as atomic numbers; order should be the same as corresponding values for each conformer
- **wanted** (*int or Iterable of int or numpy.ndarray*) – atomic number of wanted atom, or a list of those

Returns values trimmed to corresponding to desired atoms only; preserves original dimension information

Return type `numpy.ndarray`

`tesliper.datawork.geometry.drop_atoms(values: Union[Sequence, numpy.ndarray], atoms: Union[Iterable[int], numpy.ndarray], discarded: Union[int, Iterable[int], numpy.ndarray]) → numpy.ndarray`

Filters given values, returning those corresponding to atoms not specified as discarded. Roughly equivalent to: `>>> numpy.take(values, numpy.nonzero(~numpy.equal(atoms, discarded))[0], 1)` If *wanted* is list of elements, `numpy.isin` is used instead of `numpy.equal`.

Parameters

- **values** (*Sequence or numpy.ndarray*) – array of values; it should be one-dimensional list of values or n-dimensional array of shape (conformers, values[, coordinates[, other]])
- **atoms** (*Iterable of int or numpy.ndarray*) – list of atoms in molecule, given as atomic numbers; order should be the same as corresponding values for each conformer
- **discarded** (*int or Iterable of int or numpy.ndarray*) – atomic number of discarded atom, or a list of those

Returns values trimmed to corresponding to desired atoms only; preserves original dimension information

Return type `numpy.ndarray`

`tesliper.datawork.geometry.is_triangular(n: int) → bool`

Checks if number *n* is triangular.

Notes

If n is the m th triangular number, then $n = m(m+1)/2$. Solving for m using the quadratic formula: $m = (\sqrt{8n+1} - 1) / 2$, so n is triangular if and only if $8n+1$ is a perfect square.

Parameters n (*int*) – number to check

Returns True if number n is triangular, else False

Return type bool

`tesliper.datawork.geometry.get_triangular_base(n : int) → int`

Find which m th triangular number n is.

`tesliper.datawork.geometry.get_triangular(m : int) → int`

Find m th triangular number.

`tesliper.datawork.geometry.center(a : Union[Sequence[Sequence[float]],
Sequence[Sequence[Sequence[float]]]]) →
Union[Sequence[Sequence[float]],
Sequence[Sequence[Sequence[float]]]]`

Zero-center all given conformers by subtracting their centroids. Accepts single molecule or list of conformers.

`tesliper.datawork.geometry.kabsch_rotate(a : Union[Sequence[Sequence[float]],
Sequence[Sequence[Sequence[float]]]], b :
Union[Sequence[Sequence[float]],
Sequence[Sequence[Sequence[float]]]]) → numpy.ndarray`

Minimize RMSD of conformers a and b by rotating molecule a onto b . Expects given representation of conformers to be zero-centered. Both a and b may be a single molecule or a set of conformers.

Parameters

- **a** (*[Sequence of]Sequence of Sequence of float*) – Set of points representing atoms, that will be rotated to best match reference.
- **b** (*[Sequence of]Sequence of Sequence of float*) – Set of points representing atoms of the reference molecule.

Returns Rotated set of points a .

Return type numpy.ndarray

Notes

Uses Kabsch algorithm, also known as Wahba's problem. See: https://en.wikipedia.org/wiki/Kabsch_algorithm and https://en.wikipedia.org/wiki/Wahba%27s_problem

`tesliper.datawork.geometry.calc_rmsd(a : Union[Sequence[Sequence[float]],
Sequence[Sequence[Sequence[float]]]], b :
Union[Sequence[Sequence[float]],
Sequence[Sequence[Sequence[float]]]]) → numpy.ndarray`

Compute RMSD (round-mean-square deviation) of two conformers (or sets of them).

Parameters

- **a** (*[Sequence of]Sequence of Sequence of float*) – Set of points representing atoms or list thereof.
- **b** (*[Sequence of]Sequence of Sequence of float*) – Set of points representing atoms or list thereof.

Returns Value of RMSD of two conformers or list of values, if list of conformers given.

Return type float or numpy.ndarray

Notes

https://en.wikipedia.org/wiki/Root-mean-square_deviation_of_atomic_positions

`tesliper.datawork.geometry.fixed_windows(series: Sequence, size: int) → numpy.ndarray`

Simple, vectorized implementation of basic sliding window. Produces a list of windows of given *size* from given *series*.

Parameters

- **series** (*sequence*) – Series of data, of which sliding window view is requested.
- **size** (*int*) – Number of data points in the window. Must be a positive integer.

Returns List of indices, corresponding to values in the original array, that form a window

Return type numpy.ndarray

Raises

- **ValueError** – if non-positive integer given as window size
- **TypeError** – if non-integer value given as window size

Notes

Implementation inspired by <https://towardsdatascience.com/fast-and-robust-sliding-window-vectorization-with-numpy-3ad950ed>

`tesliper.datawork.geometry.stretching_windows(values: Sequence[float], size: Union[int, float],
keep_hermits: bool = False, hard_bound: bool = False)
→ Iterator[numpy.ndarray]`

Implements a sliding window of a variable size, where values in each window are at most *size* bigger than the lowest value in given window. Values yielded are `np.ndarray`s of indices of sorted values, that constitute each window.

When window reaches a border, that is an end of the *values* array or a gap between values that is larger than given *size*, it is “squeezed”, when pressed against the border, producing subsequences of the first view that touches a border. This is usefull, when one wants to form a window for each value in the original array.

```
>>> list(stretching_windows([1, 2, 3, 4, 7, 8], 3))  
[[0, 1, 2], [1, 2, 3], [2, 3], [4, 5]]
```

This “soft” right bound may be “hardened” by passing `hard_bound=True` as a parameter to a function call. A window will than move immediately to the border’s other side.

```
>>> list(stretching_windows([1, 2, 3, 4, 7, 8], 3), hard_bound=True)  
[[0, 1, 2], [1, 2, 3], [4, 5]]
```

Windows of size 1, called hermits, are by default ignored.

```
>>> arr = [1, 2, 10, 20, 22]  
>>> list(stretching_windows(arr, 5))  
[[0, 1], [3, 4]]
```

If such behavior is not desired, it may be turned off with `keep_hermits = True`. One must remember that, when a bound is “soft”, the last window is always a hermit.

```
>>> list(stretching_windows(arr, 5, keep_hermits=True))
[[0, 1], [1], [2], [3, 4], [4]]
>>> list(stretching_windows(arr, 5, keep_hermits=True, hard_bound=True))
[[0, 1], [2], [3, 4]]
```

Parameters

- **values** (*Sequence of float*) – List of values, on which sliding window view is requested.
- **size** (*int or float*) – Maximum difference of smallest and largest values inside each window.
- **keep_hermits** (*bool*) – If windows of size one should be yielded (True) or omitted (False). False by default.
- **hard_bound** (*bool*) – How window should behave close to borders. With hard bound (True) it will move to the other side of border as soon, as it is reached. With soft bound (False) it will “squeeze” when pressed against the border, producing subsequences of the first view that includes border value. False by default.

Yields *np.array of int* – List of indices, corresponding to sorted values in the original array, that form a window.

Raises ValueError – If given *size* is not a positive number.

`tesliper.datawork.geometry.pyramid_windows(series: Sequence) → Iterator[numpy.ndarray]`

Produces windows of shrinking sizes, from full sequence to last element only.

This function yields `numpy.ndarray`s with indices that may be used to index an original sequence (assuming original sequence is `numpy.ndarray` as well). The first window yielded represents a whole *series* sequence and each consecutive window is reduced by the first element, leaving only the last element in the final window. This allows for easy setup of efficient calculations in symmetric each-to-each relationship.

```
>>> series = [3, 6, 3, 5, 7]
>>> for window in pyramid_windows(series):
...     print(window)
[0 1 2 3 4]
[1 2 3 4]
[2 3 4]
[3 4]
[4]
```

Parameters series (*sequence*) – Sequence of elements, for which windows should be generated.

Yields *np.ndarray(dtype=int)* – Windows as `np.ndarray` of indices.

`tesliper.datawork.geometry.rmsd_sieve(geometry: Sequence[Sequence[Sequence[float]]], windows: Iterable[Sequence[int]], threshold: float = 1) → numpy.ndarray`

Compare conformers’ geometry to keep only those that differ at least by a given threshold.

This function calculates how similar conformers are one to another, using a RMSD measure, that is a root-mean-square deviation of atomic positions, and signalizes which of the conformers are duplicates, according to a given similarity threshold. Returned array of booleans may be treated as “originality” indicators for each

conformer: True means given conformer has distinct structure, False means given conformer is similar to some other conformer marked as “original”.

The measure of conformers’ similarity, the *threshold* parameter, is a minimum value of RMSD needed to consider two conformers different. In other words, if two conformers give a RMSD value that is lower then *threshold*, one of them will be marked as similar, producing a False in the output array.

To lower a computational expense, similarity measurement is performed in “chunks”, using a sliding window technique. Windows consist of a portion of conformers from the original data, or more precisely, indices of conformers that should be included in the particular window. First item from the window is compared to all the others that are in the same window, and if any of them is similar to the reference item, it is marked as duplicate (not “original”). The process is repeated for each window.

The windows itself should be provided by user as *windows* parameter. This provides a flexibility in the process: you may choose to sacrifice accuracy to lower necessary computational time or vice versa. You may also choose a different moving window strategy or reject it altogether, and calculate one-to-each similarity in the whole set. Iterables of windows accepted by this function may be generated with one of the dedicated moving window functions: *stretching_windows()*, *fixed_windows()*, or *pyramid_windows()*. Refer to their documentation for more information.

Parameters

- **geometry** (*sequence of sequence of sequence of float*) – A list of conformers, where each conformer is represented by a sequence of coordinates in 3-dimensional space. It is assumed that order of atoms in each conformers’ representation is identical.
- **windows** (*iterable of sequence of int*) – An iterable of windows, where each window is a list of indices. Comparison of RMSD values will be performed inside each window.
- **threshold** (*float*) – Minimum RMSD value to consider two compared conformers different.

Returns Array of booleans for each conformer: True if conformer’s structure is “original” and should be kept, False if it is a duplicate of other, “original” structure (at least according to *threshold* given), and should be discarded.

Return type np.ndarray(dtype=bool)

tesliper.datawork.intensities

Optical activity to signal intensity converters.

Functions

<i>dip_to_ir</i> (values, frequencies)	Calculates signal intensity of IR spectrum.
<i>dip_to_uv</i> (values, wavelengths)	Calculates signal intensity of UV spectrum.
<i>osc_to_uv</i> (values)	Calculates signal intensity of UV spectrum.
<i>rot_to_ecd</i> (values, wavelengths)	Calculates signal intensity of ECD spectrum.
<i>rot_to_vcd</i> (values, frequencies)	Calculates signal intensity of VCD spectrum.

`tesliper.datawork.intensities.dip_to_ir(values: numpy.ndarray, frequencies: numpy.ndarray) → numpy.ndarray`

Calculates signal intensity of IR spectrum.

Parameters

- **values** (*numpy.ndarray*) – Dipole strength values extracted from gaussian output files.

- **frequencies** (*numpy.ndarray*) – Frequencies extracted from gaussian output files.

Returns List of calculated intensity values.

Return type *numpy.ndarray*

`tesliper.datawork.intensities.rot_to_vcd(values: numpy.ndarray, frequencies: numpy.ndarray) → numpy.ndarray`

Calculates signal intensity of VCD spectrum.

Parameters

- **values** (*numpy.ndarray*) – Rotator strength values extracted from gaussian output files.
- **frequencies** (*numpy.ndarray*) – Frequencies extracted from gaussian output files.

Returns List of calculated intensity values.

Return type *numpy.ndarray*

`tesliper.datawork.intensities.osc_to_uv(values: numpy.ndarray) → numpy.ndarray`

Calculates signal intensity of UV spectrum.

Parameters **values** (*numpy.ndarray*) – Oscillator strength values extracted from gaussian output files.

Returns List of calculated intensity values.

Return type *numpy.ndarray*

`tesliper.datawork.intensities.rot_to_ecd(values: numpy.ndarray, wavelengths: numpy.ndarray) → numpy.ndarray`

Calculates signal intensity of ECD spectrum.

Parameters

- **values** (*numpy.ndarray*) – Rotator strength values extracted from gaussian output files.
- **wavelengths** (*numpy.ndarray*) – Wavelengths extracted from gaussian output files.

Returns List of calculated intensity values.

Return type *numpy.ndarray*

`tesliper.datawork.intensities.dip_to_uv(values: numpy.ndarray, wavelengths: numpy.ndarray) → numpy.ndarray`

Calculates signal intensity of UV spectrum.

Parameters

- **values** (*numpy.ndarray*) – Dipole strength values extracted from gaussian output files.
- **wavelengths** (*numpy.ndarray*) – Wavelengths extracted from gaussian output files.

Returns List of calculated intensity values.

Return type *numpy.ndarray*

tesliper.datawork.spectra

Functions that deal with spectra and spectral data.

Functions

<code>calculate_average(values, populations)</code>	Calculates weighted average of <i>values</i> , where <i>populations</i> are used as weights.
<code>calculate_spectra(frequencies, intensities, ...)</code>	Calculates spectrum for each individual conformer.
<code>convert_band(value, from_genre, to_genre)</code>	Convert one representation of band to another.
<code>count_imaginary(frequencies)</code>	Finds number of imaginary frequencies of each conformer.
<code>find_imaginary(frequencies)</code>	Finds all conformers with imaginary frequency values.
<code>find_offset(ax, ay, bx, by[, upscale])</code>	Finds value, by which the spectrum should be shifted along x-axis to best overlap with the first spectrum.
<code>find_scaling(a, b)</code>	Find factor by which values <i>b</i> should be scaled to best match values <i>a</i> .
<code>gaussian(intensities, frequencies, abscissa, ...)</code>	Gaussian fitting function for spectra calculation.
<code>idx_offset(a, b)</code>	Calculate offset by which <i>b</i> should be shifted to best overlap with <i>a</i> .
<code>lorentzian(intensities, frequencies, ...)</code>	Lorentzian fitting function for spectra calculation.
<code>unify_abscissa(ax, ay, bx, by[, upscale])</code>	Interpolate one of the given spectra to have the same points density as the other given spectrum.

`tesliper.datawork.spectra.count_imaginary(frequencies: numpy.ndarray)`

Finds number of imaginary frequencies of each conformer.

Parameters **frequencies** – List of conformers' frequencies. Array with one dimension is interpreted as list of frequencies for single conformer.

Returns Number of imaginary frequencies of each conformer.

Return type `numpy.ndarray`

Raises **ValueError** – If input array has more than 2 dimensions.

`tesliper.datawork.spectra.find_imaginary(frequencies: numpy.ndarray)`

Finds all conformers with imaginary frequency values.

Parameters **frequencies** – List of conformers' frequencies.

Returns List of the indices of conformers with imaginary frequency values.

Return type `numpy.ndarray`

Raises **ValueError** – If input array has more than 2 dimensions.

`tesliper.datawork.spectra.gaussian(intensities: numpy.ndarray, frequencies: numpy.ndarray, abscissa: numpy.ndarray, width: Union[int, float]) → numpy.ndarray`

Gaussian fitting function for spectra calculation.

Parameters

- **intensities** – Appropriate values extracted from gaussian output files.
- **frequencies** – Frequencies extracted from gaussian output files.
- **abscissa** – List of wavelength/wave number points on spectrum x axis.

- **width** – Number representing half width of peak at 1/e its maximum height.

Returns List of calculated intensity values.

Return type numpy.ndarray

Raises ValueError – If given width is not greater than zero. If *intensities* and *frequencies* are not of the same shape.

`tesliper.datawork.spectra.lorentzian(intensities: numpy.ndarray, frequencies: numpy.ndarray, abscissa: numpy.ndarray, width: Union[int, float]) → numpy.ndarray`

Lorentzian fitting function for spectra calculation.

Parameters

- **intensities** – Appropriate values extracted from gaussian output files.
- **frequencies** – Frequencies extracted from gaussian output files.
- **abscissa** – List of wavelength/wave number points on spectrum x axis.
- **width** – Number representing half width of peak at half its maximum height.

Returns List of calculated intensity values.

Return type numpy.ndarray

Raises ValueError – If given width is not greater than zero. If *intensities* and *frequencies* are not of the same shape.

`tesliper.datawork.spectra.calculate_spectra(frequencies: numpy.ndarray, intensities: numpy.ndarray, abscissa: numpy.ndarray, width: Union[int, float], fitting: Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, float], numpy.ndarray])`

Calculates spectrum for each individual conformer.

Parameters

- **frequencies** – List of conformers' frequencies in cm⁻¹. Should be of shape (number_of_conformers, number_of_frequencies).
- **intensities** – List of calculated signal intensities for each conformer. Should be of same shape as frequencies.
- **abscissa** – List of points on x axis in output spectrum in cm⁻¹.
- **width** (*int or float*) – Number representing peak width in cm⁻¹, used by fitting function.
- **fitting** (*function*) – Function, which takes intensities, frequencies, abscissa, hwhm as parameters and returns numpy.array of calculated spectrum points.

Returns Array of intensity values for each conformer.

Return type numpy.ndarray

Raises ValueError – If given width is not greater than zero. If *intensities* and *frequencies* are not of the same shape.

`tesliper.datawork.spectra.calculate_average(values: Union[Sequence[Union[int, float]], numpy.ndarray], populations: Union[Sequence[Union[int, float]], numpy.ndarray]) → numpy.ndarray`

Calculates weighted average of *values*, where *populations* are used as weights.

Parameters

- **values** – List of values for each conformer, should be of shape (N, M), where N is number of conformers and M is number of values.
- **populations** – List of conformers' populations, should be of shape (N,) where N is number of conformers. Should add up to 1.

Returns weighted arithmetic mean of values given.

Return type numpy.ndarray

Raises ValueError – If parameters of non-matching shape were given.

`tesliper.datawork.spectra.idx_offset(a: Sequence[Union[int, float]], b: Sequence[Union[int, float]]) → int`

Calculate offset by which *b* should be shifted to best overlap with *a*. Both *a* and *b* should be sets of points, interpreted as spectral data. Returned offset is a number of data points, by which *b* should be moved relative to *a*, to get the best overlap of given spectra.

Parameters

- **a** – y values of the first spectrum.
- **b** – y values of the second spectrum.

Returns Offset, in number of data points, by which spectrum *b* should be shifted to best match spectrum *a*. Positive value means it should be shifted to the right and negative value means it should be shifted to the left of *a*.

Return type int

Notes

The best overlap is found by means of cross-correlation of given spectra.

`tesliper.datawork.spectra.unify_abcissa(ax: Sequence[Union[int, float]], ay: Sequence[Union[int, float]], bx: Sequence[Union[int, float]], by: Sequence[Union[int, float]], upscale: bool = True) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]`

Interpolate one of the given spectra to have the same points density as the other given spectrum.

Which spectra should be interpolated is determined based on the density of points of both spectra, by default more loosely spaced spectrum is interpolated to match spacing of the other spectrum. This may be changed by passing `upscale=False` to the function call.

Parameters

- **ax** – Abscissa of the first spectrum.
- **ay** – Values of the first spectrum.
- **bx** – Abscissa of the second spectrum.
- **by** – Values of the second spectrum.
- **upscale** – If interpolation should be done on more loosely spaced spectrum (default). When set to False, spectrum with lower resolution will be treated as reference.

Returns Spectra, one unchanged and one interpolated, as a tuple of numpy arrays of x and y values. I.e. `tuple(ax, ay, new_bx, new_by)` or `tuple(new_ax, new_ay, bx, by)`, depending on values of *upscale* parameter.

Return type tuple of np.arrays of numbers

`tesliper.datawork.spectra.find_offset(ax: Sequence[Union[int, float]], ay: Sequence[Union[int, float]],
bx: Sequence[Union[int, float]], by: Sequence[Union[int, float]],
upscale: bool = True) → float`

Finds value, by which the spectrum should be shifted along x-axis to best overlap with the first spectrum. If resolution of spectra is not identical, one of them will be interpolated to match resolution of the other one. By default interpolation is done on the lower-resolution spectra. This can be changed by passing `upscale = False` to function call.

Parameters

- **ax** – Abscissa of the first spectrum.
- **ay** – Values of the first spectrum.
- **bx** – Abscissa of the second spectrum.
- **by** – Values of the second spectrum.
- **upscale** – If interpolation should be done on more loosely spaced spectrum (default). When set to False, spectrum with lower resolution will be treated as reference for density of data points.

Returns Value, by which second spectrum should be shifted, in appropriate units.

Return type float

`tesliper.datawork.spectra.find_scaling(a: Sequence[Union[int, float]], b: Sequence[Union[int, float]])
→ float`

Find factor by which values *b* should be scaled to best match values *a*.

Parameters

- **a** – y values of the first spectrum.
- **b** – y values of the second spectrum.

Returns Scaling factor for *b* values.

Return type float

Notes

If scaling factor cannot be reasonably given, i.e. when *b* is an empty list or list of zeros or NaNs, `1.0` is returned. Values lower than 1% of maximum are ignored.

`tesliper.datawork.spectra.convert_band(value: Union[float, numpy.ndarray], from_genre: str, to_genre:
str) → Union[float, numpy.ndarray]`

Convert one representation of band to another.

Parameters

- **value** – Value(s) to convert.
- **from_genre** – Genre specifying a representation of band of input data. Should be one of: 'freq', 'wavelen', 'ex_en'.
- **to_genre** – Genre specifying a representation of band, to which you want to convert. Should be one of: 'freq', 'wavelen', 'ex_en'.

Returns Requested representation of bands. If *from_genre* is same as *to_genre*, then simply *value* is returned.

Return type float or np.ndarray

3.8.2 tesliper.exceptions

Project-specific errors.

Exceptions

<i>InconsistentDataError</i>	Raised to signalize problems with conformers' data consistency.
<i>InvalidElementError</i>	Used by tesliper to indicate, that value cannot be interpreted as an element.
<i>InvalidStateError</i>	Used by ParserBase class to signalize problems when handling states.
<i>TesliperError</i>	Base class for Exceptions used by tesliper library.

exception tesliper.exceptions.TesliperError

Base class for Exceptions used by tesliper library.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception tesliper.exceptions.InconsistentDataError

Raised to signalize problems with conformers' data consistency. Subclasses TeslaError.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception tesliper.exceptions.InvalidStateError

Used by ParserBase class to signalize problems when handling states. Subclasses TeslaError and ValueError.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception tesliper.exceptions.InvalidElementError

Used by tesliper to indicate, that value cannot be interpreted as an element. Subclasses TeslaError and ValueError.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

3.8.3 tesliper.extraction

Classes for reading and parsing files.

Abstract Base Class for parsers, as well as concrete parser implementations are defined in this subpackage. It also contains a *Soxhlet* class that is designed to orchestrate batch data extraction.

Modules

<code>tesliper.extraction.gaussian_parser</code>	Parser for Gaussian output files.
<code>tesliper.extraction.parameters_parser</code>	Parser for reading spectra calculation parameters from file.
<code>tesliper.extraction.parser_base</code>	This module contains a definition of Abstract Base Class for file parsers.
<code>tesliper.extraction.soxhlet</code>	A tool for batch parsing files from specified directory.
<code>tesliper.extraction.spectra_parser</code>	Parser for spectra files.

tesliper.extraction.gaussian_parser

Parser for Gaussian output files.

Classes

<code>GaussianParser()</code>	Parser for extracting data from human-readable output files from Gaussian computational chemistry software (.log and .out files).
-------------------------------	---

class tesliper.extraction.gaussian_parser.GaussianParser

Parser for extracting data from human-readable output files from Gaussian computational chemistry software (.log and .out files).

This class implements methods for reading information about conducted calculations' parameters, molecule energy, structure optimization, and calculation of spectral properties. It's use is as straightforward as:

```
>>> parser = GaussianParser()
>>> with open('path/to/file.out') as file:
>>>     data = parser.parse(file)
```

Dictionary with data extracted is also stored as `data` attribute of instance used for parsing. Each key in said dictionary is a name of its value data type, called from now on a 'data genre' (to avoid confusion with Python's data type). Below is a full list of data genres recognized by this parser, with their description:

freq [list of floats, available from freq job] harmonic vibrational frequencies (cm^{-1})

mass [list of floats, available from freq job] reduced masses (AMU)

frc [list of floats, available from freq job] force constants (mDyne/A)

iri [list of floats, available from freq job] IR intensities (KM/mole)

dip [list of floats, available from freq=VCD job] dipole strengths ($10^{-40} \text{esu}^2 \cdot \text{cm}^2$)

rot [list of floats, available from freq=VCD job] rotational strengths ($10^{-44} \text{esu}^2 \cdot \text{cm}^2$)

emang [list of floats, available from freq=VCD job] E-M angle = Angle between electric and magnetic dipole transition moments (deg)

depolarp [list of floats, available from freq=Raman job] depolarization ratios for plane incident light

depolaru [list of floats, available from freq=Raman job] depolarization ratios for unpolarized incident light

ramanactiv [list of floats, available from freq=Raman job] Raman scattering activities (A^4/AMU)

ramact [list of floats, available from freq=ROA job] Raman scattering activities (A^{**4}/AMU)

depp [list of floats, available from freq=ROA job] depolarization ratios for plane incident light

depu [list of floats, available from freq=ROA job] depolarization ratios for unpolarized incident light

alpha2 [list of floats, available from freq=ROA job] Raman invariants $\text{Alpha2} = \alpha^{**2}$ (A^{**4}/AMU)

beta2 [list of floats, available from freq=ROA job] Raman invariants $\text{Beta2} = \beta(\alpha)^{**2}$ (A^{**4}/AMU)

alphag [list of floats, available from freq=ROA job] ROA invariants $\text{AlphaG} = \alpha G' (10^{**4} A^{**5}/\text{AMU})$

gamma2 [list of floats, available from freq=ROA job] ROA invariants $\text{Gamma2} = \beta(G')^{**2} (10^{**4} A^{**5}/\text{AMU})$

delta2 [list of floats, available from freq=ROA job] ROA invariants $\text{Delta2} = \beta(A)^{**2}, (10^{**4} A^{**5}/\text{AMU})$

raman1 [list of floats, available from freq=ROA job] Far-From-Resonance Raman intensities =ICPu/SCPu(180) (K)

roa1 [list of floats, available from freq=ROA job] ROA intensities =ICPu/SCPu(180) (10^{**4} K)

cid1 [list of floats, available from freq=ROA job] CID=(ROA/Raman)* 10^{**4} =ICPu/SCPu(180)

raman2 [list of floats, available from freq=ROA job] Far-From-Resonance Raman intensities =ICPd/SCPd(90) (K)

roa2 [list of floats, available from freq=ROA job] ROA intensities =ICPd/SCPd(90) (10^{**4} K)

cid2 [list of floats, available from freq=ROA job] CID=(ROA/Raman)* 10^{**4} =ICPd/SCPd(90)

raman3 [list of floats, available from freq=ROA job] Far-From-Resonance Raman intensities =DCPI(180) (K)

roa3 [list of floats, available from freq=ROA job] ROA intensities =DCPI(180) (10^{**4} K)

cid3 [list of floats, available from freq=ROA job] CID=(ROA/Raman)* 10^{**4} =DCPI(180)

rc180 [list of floats, available from freq=ROA job] RC180 = degree of circularity

wavelen [list of floats, available from td job] excitation energies (nm)

ex_en [list of floats, available from td job] excitation energies (eV)

eemang [list of floats, available from td job] E-M angle = Angle between electric and magnetic dipole transition moments (deg)

vdip [list of floats, available from td job] dipole strengths (velocity)

ldip [list of floats, available from td job] dipole strengths (length)

vrot [list of floats, available from td job] rotatory strengths (velocity) in cgs (10^{**-40} erg-esu-cm/Gauss)

lrot [list of floats, available from td job] rotatory strengths (length) in cgs (10^{**-40} erg-esu-cm/Gauss)

vosc [list of floats, available from td job] oscillator strengths

losc [list of floats, available from td job] oscillator strengths

transitions [list of lists of lists of (int, int, float), available from td job] transitions (first to second) and their coefficients (third)

scf [float, always available] SCF energy

zpe [float, available from freq job] Sum of electronic and zero-point Energies (Hartree/Particle)

ten [float, available from freq job] Sum of electronic and thermal Energies (Hartree/Particle)

ent [float, available from freq job] Sum of electronic and thermal Enthalpies (Hartree/Particle)

gib [float, available from freq job] Sum of electronic and thermal Free Energies (Hartree/Particle)

zpecorr [float, available from freq job] Zero-point correction (Hartree/Particle)

tencorr [float, available from freq job] Thermal correction to Energy (Hartree/Particle)

entcorr [float, available from freq job] Thermal correction to Enthalpy (Hartree/Particle)

gibcorr [float, available from freq job] Thermal correction to Gibbs Free Energy (Hartree/Particle)

command [str, always available] command used for calculations

normal_termination [bool, always available] true if Gaussian job seem to exit normally, false otherwise

optimization_completed [bool, available from opt job] true if structure optimization was performed successfully

version [str, always available] version of Gaussian software used

charge [int, always available] molecule's charge

multiplicity [int, always available] molecule's spin multiplicity

input_atoms [list of str, always available] input atoms as a list of atoms' symbols

input_geom [list of lists of floats, always available] input geometry as X, Y, Z coordinates of atoms

stoichiometry [str, always available] molecule's stoichiometry

last_read_atoms [list of ints, always available] molecule's atoms as atomic numbers

last_read_geom [list of lists of floats, always available] molecule's geometry (last one found in file) as X, Y, Z coordinates of atoms

optimized_atoms [list of ints, available from successful opt job] molecule's atoms read from optimized geometry as atomic numbers

optimized_geom [list of lists of floats, available from successful opt job] optimized geometry as X, Y, Z coordinates of atoms

data
Data extracted during last parsing.

Type dict

parse(*lines*) → dict
Parses content of Gaussian output file and returns dictionary of found data.

Parameters **lines** (*iterator*) – Gaussian output file in a form iterable by lines of text. It may be a file handle, a list of strings, an io.StringIO instance, or similar. Please note that it should not be just a string instance, as it is normally iterated by a character, not by a line.

Returns Dictionary of extracted data.

Return type dict

initial(*line: str*) → None
First step of parsing Gaussian output file. It populates parser.data dictionary with these data genes: 'normal_termination', 'version', 'command', 'charge', 'multiplicity', 'input_geom'. Optionally, 'optimization_completed' genre is added if optimization was requested in calculation job.

Parameters **line** (*str*) – Line of text to parse.

wait(*line: str*) → None
This function searches for lines of text triggering other parsing states. It also updates a parser.data dictionary with 'normal_termination', 'scf', 'stoichiometry' data genres.

Parameters **line** (*str*) – Line of text to parse.

geometry(*line: str*) → None

Function for extracting information about molecule standard orientation geometry from Gaussian output files. It updates parser.data dictionary with 'last_read_atoms' and 'last_read_geom' data genres.

Parameters **line** (*str*) – Line of text to parse.

optimization(*line: str*) → None

This method scans optimization data in Gaussian output file, updating parser.data dictionary with 'stoichiometry', 'scf', 'optimization_completed', 'optimized_atoms', and 'optimized_geom' data genres (last two via [geometry\(\)](#) method).

Parameters **line** (*str*) – Line of text to parse.

frequencies(*line: str*) → None

Responsible for extracting harmonic vibrations-related data and information about molecule's energy.

Parameters **line** (*str*) – Line of text to parse.

excited(*line: str*) → None

Responsible for extracting electronic transitions-related data from Gaussian output file. Updates parser.data dictionary with 'ldip', 'losc', 'vdip', 'vosc', 'vrot', 'eemang', 'lrot', 'wavelen', 'ex_en', and 'transitions' data genres.

Parameters **line** (*str*) – Line of text to parse.

add_state(*state: Callable, name: str = "", trigger: str = ""*)

Register callable as parser's state.

This method registers a callable under *name* key in [states](#) dictionary. If *trigger* parameter is given, it is registered under the same key in [triggers](#) dictionary.

Parameters

- **state** (*Callable*) – callable, that is to be registered as parser's state
- **name** (*str, optional*) – name under which the callable should be registered; defaults to callable.__name__
- **trigger** (*str, optional*) – string with regular expression, that will be compiled with re module

Returns callable object registered as state

Return type Callable

remove_state(*name: str*)

Removes the state from parser's registered states.

Parameters **name** (*str*) – name of state, that should be unregistered

Raises [InvalidStateError](#) – if no callable was registered under the name 'name'

static state(*state=None, trigger=None*)

Convenience decorator for registering a method as parser's state. It can be with or without 'trigger' parameter, like this:

```
>>> @ParserBase.state
... def method(self, arg): pass
```

or

```
>>> @ParserBase.state(trigger='triggering regex')
... def method(self, arg): pass
```

This function marks a method *state* as parser’s state by defining `is_state` attribute on said method and setting its values to `True`. If *trigger* is given, it is stored in method’s attribute *trigger*. During instantiation of *ParserBase*’s subclass, methods marked as states are registered under `method.__name__` key in its *states* (and possibly *triggers*) attribute. It is meaningless if used outside of *ParserBase*’s subclass definition.

Parameters

- **state** (*Callable*) – callable, that is to be registered as parser’s state
- **trigger** (*str*, *optional*) – string with regular expression, that will be compiled with `re` module

Returns callable object registered as state if ‘state’ was given or decorator if only ‘trigger’ was given

Return type *Callable*

Raises

- **TypeError** – if no arguments given
- **InvalidStateError** – if *state* argument is not callable

property workhorse: *Callable*

Callable marked as a current state used by parser object.

Setter can take a callable or a string as a parameter. If name as string is passed to setter, it will be translated to a method registered as state. If no method was registered under this name, *InvalidStateError* will be raised. No other checks are performed when argument is callable.

tesliper.extraction.parameters_parser

Parser for reading spectra calculation parameters from file.

Functions

<i>fitting</i> (<i>s</i>)	Get fitting function mentioned in a given string <i>s</i> , ignoring anything else.
<i>quantity</i> (<i>s</i>)	Convert to float first occurrence of float-looking part of string <i>s</i> , ignoring anything else.

Classes

<i>ParametersParser</i> ()	Parser for configuration files containing spectra calculation parameters.
----------------------------	---

`tesliper.extraction.parameters_parser.quantity`(*s*: *str*) → float

Convert to float first occurrence of float-looking part of string *s*, ignoring anything else. Raise `configparser.ParsingError` if float cannot be found.

Parameters *s* (*str*) – string containing a float

Returns extracted float value

Return type float

`tesliper.extraction.parameters_parser.fitting(s: str) → Callable`

Get fitting function mentioned in a given string *s*, ignoring anything else. Raise `configparser.ParsingError` if known function name cannot be found.

Parameters *s* (*str*) – string containing name of fitting function

Returns an identified fitting function

Return type callable

class `tesliper.extraction.parameters_parser.ParametersParser`

Parser for configuration files containing spectra calculation parameters.

Configuration file should be in format similar to .ini files: a list of key-value pairs, separated with “=” or “:”, one pair for line. Standard parameters (width, start, stop, step, and fitting) will be converted to appropriate data type, i.e. float or function reference. If parameter value cannot be converted to its target type, it will be ignored and warning will be emitted. Any other (unexpected) parameters are included in the output and left as they are.

The parser is case-insensitive and knows some alias names of expected parameters: for instance, “hwhm”, “half width of band in half height”, “half width at half maximum” will be all recognized as “width” parameter. If you wish to add custom aliases, update `ParametersParser.ALIASES` dictionary with appropriate “alias”: “target” pair.

Notes

`ParametersParser` is using Python’s `configparser`, so it will parse files that contain a section header, enclosed in braces. However, the section name will be ignored and there may be only one such section, otherwise an exception is raised.

optionxform(*optionstr: str*) → str

Translates option names to desired form - lowercase and standard wording, as defined in `ALIASES`.

property parameters: dict

Dictionary of parameters for calculating spectra extracted from parsed file and converted to appropriate type.

parse(*source: Union[str, pathlib.Path]*)

Parse given source file to get stored parameters.

Parameters *source* (*str* or *Path*) – Path to file with calculations’ parameters.

Returns Parsed parameters.

Return type dict

tesliper.extraction.parser_base

This module contains a definition of Abstract Base Class for file parsers.

Classes

<code>ParserBase()</code>	Abstract Base Class for parsers implemented as finite state machines.
---------------------------	---

class tesliper.extraction.parser_base.ParserBase

Abstract Base Class for parsers implemented as finite state machines.

This base class defines some methods to organize work parsers implemented as finite state machines: automates registration of methods and functions as parser's states, manages its execution, and registers derived class as parser used for certain type of files (which registry is used by *Soxhlet* object).

The default parsing flow goes as follow:

1. method `parse()` is called with file handle as argument;
2. method `initial()` is set as a 'workhorse'
3. 'workhorse' is called for consecutive lines in file handle
4. `initial()` checks if any registered trigger matches current line
5. `workhorse()` is changed to method associated with first matching trigger
6. calling 'workhorse' on consecutive lines continues
7. `parse()` returns dictionary with extracted values

To make this possible, each method marked as state should return dictionary (or sequence convertible to dict) and handle changing 'workhorse' to next appropriate state. To mark a method as parser's state use `ParserBase.state` decorator in class definition or add a state directly to parser instance using 'add_state' method.

When subclassing `ParserBase`, one should implement `initial()` and `parse()` methods. Those abstract methods implement basic functionality, described above. See methods' documentation for more details. If you wish not to use default `ParserBase`'s protocol, simply override those methods to your liking. Values for class attributes `extensions` and `purpose` should also be provided.

To register class derived from `ParserBase` for use by *Soxhlet* object, simply set `purpose` class attribute to name, under which class should be registered. Setting it to one of names already defined (e.g. 'gaussian') will override the default parser used by *Soxhlet* object.

states

Dictionary of parser states, created automatically on object instantiation from object methods marked as states; method name is used as a key by default.

Type dict

triggers

Dictionary of triggers for parser states, created automatically on object instantiation from object methods marked as states with triggers; key for a particular state trigger should be the same as state's key in `states` dictionary.

Type dict

abstract property extensions

File extensions that should be considered compatible with a parser subclassing *ParserBase*. It will be used by *Soxhlet* to identify which files to parse when reading files in batch. Should be a class attribute with a tuple of str, where each element is a file extension. May also be an empty tuple, if files discovery feature is not needed for the parser.

abstract property purpose

An identifier for a parser subclassing *ParserBase*. It allows *tesliper* to pick a correct parser for each parsing task. A falsy value, i.e. an empty string or *None* prevents the parser from being registered for use by *tesliper*. If custom subclass uses a *purpose* already known, e.g. “gaussian” or “spectra”, it will override the original parser for this purpose.

property workhorse: Callable

Callable marked as a current state used by parser object.

Setter can take a callable or a string as a parameter. If name as string is passed to setter, it will be translated to a method registered as state. If no method was registered under this name, *InvalidStateError* will be raised. No other checks are performed when argument is callable.

add_state(state: Callable, name: str = "", trigger: str = "")

Register callable as parser’s state.

This method registers a callable under *name* key in *states* dictionary. If *trigger* parameter is given, it is registered under the same key in *triggers* dictionary.

Parameters

- **state** (Callable) – callable, that is to be registered as parser’s state
- **name** (str, optional) – name under which the callable should be registered; defaults to callable.__name__
- **trigger** (str, optional) – string with regular expression, that will be compiled with re module

Returns callable object registered as state

Return type Callable

remove_state(name: str)

Removes the state from parser’s registered states.

Parameters **name** (str) – name of state, that should be unregistered

Raises *InvalidStateError* – if no callable was registered under the name ‘name’

abstract initial(line: str) → dict

An initial parser state.

A default implementation checks if any of defined triggers matches a line and sets an associated state as parser’s workhorse, if it does. This is an abstract method and should be overridden in subclass. Its default implementation can be used, however, by calling *super().initial(line)* in subclass’s method.

Notes

`initial()` method is always registered as parser's state.

Parameters `line` (*str*) – currently parsed line

Returns empty dictionary

Return type dict

abstract `parse(lines: Iterable) → dict`

Parses consecutive elements of iterable and returns data found as dictionary.

Dictionary with extracted data is updated with workhorse's return value, so all states should return dictionary or compatible sequence. This is an abstract method and should be overridden in subclass. Its default implementation can be used, however, by calling `data = super().parse(lines)` in subclass's method.

Notes

After execution - either successful or interrupted by exception - `workhorse` is set back to `initial()` method.

Parameters `lines` (*Iterable*) – iterable (i.e. file handle), that will be parsed, line by line

Returns dictionary with data extracted by parser

Return type dict

Raises `InvalidStateError` – if dictionary can't be updated with state's return value

static `state(state=None, trigger=None)`

Convenience decorator for registering a method as parser's state. It can be with or without 'trigger' parameter, like this:

```
>>> @ParserBase.state
... def method(self, arg): pass
```

or

```
>>> @ParserBase.state(trigger='triggering regex')
... def method(self, arg): pass
```

This function marks a method `state` as parser's state by defining `is_state` attribute on said method and setting its values to `True`. If `trigger` is given, it is stored in method's attribute `trigger`. During instantiation of `ParserBase`'s subclass, methods marked as states are registered under `method.__name__` key in its `states` (and possibly `triggers`) attribute. It is meaningless if used outside of `ParserBase`'s subclass definition.

Parameters

- **state** (*Callable*) – callable, that is to be registered as parser's state
- **trigger** (*str, optional*) – string with regular expression, that will be compiled with `re` module

Returns callable object registered as state if 'state' was given or decorator if only 'trigger' was given

Return type Callable

Raises

- **TypeError** – if no arguments given
- **InvalidStateError** – if *state* argument is not callable

tesliper.extraction.soxhlet

A tool for batch parsing files from specified directory.

Classes

<code>Soxhlet</code> ([<i>path</i> , <i>purpose</i> , <i>wanted_files</i> , ...])	A tool for data extraction from files in specific directory.
--	--

```
class tesliper.extraction.soxhlet.Soxhlet(path: Optional[Union[str, pathlib.Path]] = None, purpose:  
                                           str = 'gaussian', wanted_files: Optional[Iterable[Union[str,  
                                           pathlib.Path]]] = None, extension: Optional[str] = None,  
                                           recursive: bool = False)
```

A tool for data extraction from files in specific directory. Typical use:

```
>>> s = Soxhlet('absolute/path_to/working/directory')  
>>> data = s.extract()
```

Parameters

- **path** (*str* or *pathlib.Path*) – String representing absolute path to directory containing files, which will be the subject of data extraction.
- **purpose** (*str*) – Determines which from registered parsers should be used for extraction. *purposes* supported out-of-the-box are “gaussian”, “spectra”, and “parameters”.
- **wanted_files** (*list of str or pathlib.Path objects, optional*) – List of files, that should be loaded for further extraction. If omitted, all output files present in directory will be processed.
- **extension** (*str, optional*) – A string representing file extension of output files, that should be parsed. If omitted, Soxhlet will try to resolve it based on contents of directory given in *path* parameter.
- **recursive** (*bool*) – If True, given *path* will be searched recursively, extracting data from subdirectories, otherwise subdirectories are ignored and only files placed directly in *path* will be parsed.

Raises

- **FileNotFoundError** – If *path* passed as argument to constructor doesn’t exist or is not a directory.
- **ValueError** – If no parser is registered for given *purpose*.

property all_files

List of all files present in directory bounded to Soxhlet instance. If its *recursive* attribute is True, also files from subdirectories are included.

property files

List of all wanted files available in given directory. If *wanted_files* is not specified, evaluates to all files in said directory. If Soxhlet object’s *recursive* attribute is True, also files from subdirectories are included.

property wanted_files: Optional[Set[str]]

Set of files that are desired for data extraction, stored as filenames without an extension. Any iterable of strings or Path objects is transformed to this form.

```
>>> s = Soxhlet()
>>> s.wanted_files = [Path("./dir/file_one.out"), Path("./dir/file_two.out")]
>>> s.wanted_files
{"file_one", "file_two"}
```

May also be set to None or other “falsy” value, in such case it is ignored.

property output_files: List[pathlib.Path]

List of (sorted by file name) gaussian output files from files list associated with Soxhlet instance.

filter_files(*ext: Optional[str] = None*) → List[pathlib.Path]

Filters files from filenames list.

Filters file names in list associated with *Soxhlet* object instance. It returns list of file names ending with provided ext string, representing file extension and starting with any of filenames associated with instance as wanted_files if those were provided.

Parameters *ext* (*str*) – Strings representing file extension.

Returns List of filtered filenames as strings.

Return type list

Raises **ValueError** – If parameter *ext* is not given and attribute *extension* in None.

guess_extension() → str

Tries to figure out which extension should be assumed.

Looks for files, which names end with one of the extensions defined by currently used parser. Returns extension that matches as the only one. Raises an exception if extension cannot be easily guessed.

Returns The extension of files that are present in filenames list, which current parser can parse.

Return type str

Raises

- **ValueError** – If more than one type of files declared by a current parser as possibly compatible is present in list of filenames.
- **FileNotFoundError** – If none of files declared by a current parser as possibly compatible are present in list of filenames.
- **TypeError** – If current parser does not declare any compatible file extensions.

extract_iter() → Generator[Tuple[str, dict], None, None]

Extracts data from files associated with *Soxhlet* instance (*via path and wanted_files* attributes), using a current parser (determined by a *purpose* provided on *Soxhlet*’s instantiation). Implemented as generator. If Soxhlet instance’s *recursive* attribute is True, also files from subdirectories are parsed.

Yields *tuple* – Two item tuple with name of parsed file as first and extracted data as second item, for each file associated with Soxhlet instance.

extract() → dict

Extracts data from files associated with *Soxhlet* instance (*via path and wanted_files* attributes), using a current parser (determined by a *purpose* provided on *Soxhlet*’s instantiation). If Soxhlet.*recursive* attribute is True, also files from subdirectories are parsed.

Returns dictionary of extracted data, with name of parsed file as key and data as value, for each file associated with Soxhlet instance.

Return type dict of dicts

parse_one(*source: Union[str, pathlib.Path]*) → Any

Parse one file using current parser (determined by a *purpose* provided on [Soxhlet](#)’s instantiation) and return extracted data.

Parameters **source** (*str or Path*) – Path or Path-like object to a file. May be given as an absolute path or relative to the Soxhlet.path.

Returns Data in a format that current parser provides.

Return type any

Raises **FileNotFoundError** – If no *source* file is found.

tesliper.extraction.spectra_parser

Parser for spectra files.

Classes

[SpectraParser](#)()

Parser for files containing spectral data.

class tesliper.extraction.spectra_parser.SpectraParser

Parser for files containing spectral data. It can parse .txt (in “x y” format) and .csv files, returning an numpy.ndarray with loaded spectrum. Parsing process may be customized by specifying what delimiter of values should be expected and in which column x- and y-values are, if there are more than 2 columns of data. If file contains any header, it is ignored.

parse(*filename: Union[str, pathlib.Path]*, *delimiter: Optional[str] = None*, *xcolumn: int = 0*, *ycolumn: int = 1*) → numpy.ndarray

Loads spectral data from file to numpy.array. Currently supports only .txt, .xy, and .csv files.

Parameters

- **filename** (*str*) – path to file containing spectral data
- **delimiter** (*str, optional*) – character used to delimit columns in file, defaults to whitespace
- **xcolumn** (*int, optional*) – column, that should be used as points on x axis, defaults to 0 (first column)
- **ycolumn** (*int, optional*) – column, that should be used as values on y axis, defaults to 1 (second column)

Returns two-dimensional numpy array ([[x-values], [y-values]]) of data type float

Return type numpy.array

initial(*filename: str*)

An initial parser state.

A default implementation checks if any of defined triggers matches a line and sets an associated state as parser’s workhorse, if it does. This is an abstract method and should be overridden in subclass. Its default implementation can be used, however, by calling `super().initial(line)` in subclass’s method.

Notes

`initial()` method is always registered as parser's state.

Parameters `line (str)` – currently parsed line

Returns empty dictionary

Return type dict

parse_txt(*file: pathlib.Path*)

Loads spectral data from .txt or .xy file to numpy.array.

Parameters

- **file** (*str*) – path to file containing spectral data
- **delimiter** (*str, optional*) – character used to delimit columns in file, defaults to whitespace
- **xcolumn** (*int, optional*) – column, that should be used as points on x axis, defaults to 0 (first column)
- **ycolumn** (*int, optional*) – column, that should be used as values on y axis, defaults to 1 (second column)

Returns

- *numpy.array* – two-dimensional numpy array ([[x-values], [y-values]]) of data type 'float'
- *Rises*
- *—*
- *ValueError* – if file passed was read to end, but no spectral data was found; this includes columns' numbers out of range and usage of inappropriate delimiter

parse_csv(*file: pathlib.Path*)

Loads spectral data from csv file to numpy.array.

Parameters

- **file** (*str*) – path to file containing spectral data
- **delimiter** (*str, optional*) – character used to delimit columns in file, defaults to ','
- **xcolumn** (*int, optional*) – column, that should be used as points on x axis, defaults to 0 (first column)
- **ycolumn** (*int, optional*) – column, that should be used as values on y axis, defaults to 1 (second column)

Returns two-dimensional numpy array ([[x-values], [y-values]]) of data type 'float'

Return type numpy.array

parse_spc(*file*)

Loads spectral data from spc file to numpy.array.

Notes

This method is not implemented yet, it will raise an error when called.

Parameters **file** (*str*) – path to file containing spectral data

Returns two-dimensional numpy array ([[x-values], [y-values]]) of data type ‘float’

Return type numpy.array

Raises **NotImplementedError** – Whenever called, as this functionality is not implemented yet.

add_state(*state: Callable, name: str = "", trigger: str = ""*)

Register callable as parser’s state.

This method registers a callable under *name* key in **states** dictionary. If *trigger* parameter is given, it is registered under the same key in **triggers** dictionary.

Parameters

- **state** (*Callable*) – callable, that is to be registered as parser’s state
- **name** (*str, optional*) – name under which the callable should be registered; defaults to `callable.__name__`
- **trigger** (*str, optional*) – string with regular expression, that will be compiled with `re` module

Returns callable object registered as state

Return type Callable

remove_state(*name: str*)

Removes the state from parser’s registered states.

Parameters **name** (*str*) – name of state, that should be unregistered

Raises **InvalidStateError** – if no callable was registered under the name ‘name’

static state(*state=None, trigger=None*)

Convenience decorator for registering a method as parser’s state. It can be with or without ‘trigger’ parameter, like this:

```
>>> @ParserBase.state
... def method(self, arg): pass
```

or

```
>>> @ParserBase.state(trigger='triggering regex')
... def method(self, arg): pass
```

This function marks a method *state* as parser’s state by defining `is_state` attribute on said method and setting its values to `True`. If *trigger* is given, it is stored in method’s attribute *trigger*. During instantiation of *ParserBase*’s subclass, methods marked as states are registered under `method.__name__` key in its **states** (and possibly **triggers**) attribute. It is meaningless if used outside of *ParserBase*’s subclass definition.

Parameters

- **state** (*Callable*) – callable, that is to be registered as parser’s state
- **trigger** (*str, optional*) – string with regular expression, that will be compiled with `re` module

Returns callable object registered as state if ‘state’ was given or decorator if only ‘trigger’ was given

Return type Callable

Raises

- **TypeError** – if no arguments given
- **InvalidStateError** – if *state* argument is not callable

property workhorse: Callable

Callable marked as a current state used by parser object.

Setter can take a callable or a string as a parameter. If name as string is passed to setter, it will be translated to a method registered as state. If no method was registered under this name, *InvalidStateError* will be raised. No other checks are performed when argument is callable.

3.8.4 tesliper.glassware

Data containers.

Modules

<code>tesliper.glassware.array_base</code>	Core functionality of <i>dataArray</i> classes.
<code>tesliper.glassware.arrays</code>	Implements <i>dataArray</i> -like objects for handling arrayed data.
<code>tesliper.glassware.conformers</code>	A tesliper's main data storage.
<code>tesliper.glassware.spectra</code>	Objects representing spectra.

tesliper.glassware.array_base

Core functionality of *dataArray* classes.

This module implements the base class for *dataArray*s and its core functionality, namely validation of array-like data, along with some helper functions. To implement a *dataArray*-like container, subclass the *ArrayBase* class and use one of the *ArrayProperty* classes to create a validated array-like instance attribute for your new class. You should also provide *associated_genres* class attribute to signalize, which genres this new *dataArray*-like class should be used for.

The most basic example may look like this:

```
>>> class MydataArray(ArrayBase):
...     associated_genres = ("foo",)
...     filenames = ArrayProperty(dtype=str)
...     values = ArrayProperty(check_against="filenames")
...     def __init__(genre, filenames, values, allow_data_inconsistency=False):
...         super().__init__(genre, filenames, values, allow_data_inconsistency)
```

```
>>> foo_array = MydataArray("foo", ["a", "b", "c"], values=[1, 2, 3])
```

This definition would be almost a re-implementation of what *ArrayBase* already provides, but is a good starting point for explanation, so let's elaborate on it a little. *ArrayBase* expects 4 parameters on initialization of its subclass: *genre* is a genre of data stored, *filenames* is a list of conformer identifiers, *values* is - not surprisingly - a list of data values

for each conformer, and `allow_data_inconsistency` is a boolean flag that controls process of validation of array-like attributes.

`filenames` and `values` are [ArrayProperty](#) instances - values passed to the constructor as parameters of these names will be checked and validated, and stored as `numpy.ndarrays`. Moreover, `filenames` will be stored as strings, because we told the [ArrayProperty](#) this is our desired data type for this array-like attribute, using `dtype=str`. The default data type is `float`, so `values` will be converted to floats.

```
>>> foo_array.filenames
array(["a", "b", "c"], dtype=str)
>>> foo_array.values
array([1.0, 2.0, 3.0], dtype=float)
```

`check_against="filenames"` tells [ArrayProperty](#) to validate `values` using `filenames` as a reference for desired shape of `values` array. If shape is different than shape of the reference, [InconsistentDataError](#) is raised. If you will deal with multidimensional data, you can utilize `check_depth` parameter to signalize that arrays should have identical shapes only to some certain depth, for example `check_depth=2` would accept arrays of shapes (10, 20) and (10, 20, 3) but would raise exception on arrays shaped (10,) and (10, 3). However, in our simple example it wouldn't make much sense to check more than default depth of 1, since `filenames` have only one dimension.

```
>>> MyDataArray("foo", ["a", "b", "c"], values=[1, 2, 3, 4])
Traceback (most recent call last):
...
InconsistentDataError: values and filenames must have the same shape up to 1 dimensions.
Arrays of shape (3,) and (4,) were given.
```

The above exception is also raised if values given to [ArrayProperty](#) are a jagged sequence, that is not all entries of the array have identical number of sub-entries. An example of jagged array would be `[[1, 2], [3]]`. Data in this format usually comes from reading calculations of different molecules rather than conformers, or from corrupted or incomplete output files, so it is not allowed by default. However, if you are sure that you want to work with such data, you can pass `allow_data_inconsistency=True` to your `MyDataArray` constructor and [ArrayProperty](#) will try to fill-in missing values, producing `numpy.ma.masked_array` or at least will ignore inconsistencies. You can chose the fill value by specifying `fill_value` parameter on [ArrayProperty](#) instantiation.

Finally we specify `associated_genres = ("foo",)`, which is the only thing in our example that's not already defined by [ArrayBase](#). This class attribute informs [Conformers](#) object that it should use this [ArrayBase](#) subclass to instantiate [DataArray](#)-like objects for data genres specified in `associated_genres`. It must be specified as a tuple of strings, but may be left empty, if no genre should be associated with this particular class. However, the main purpose of [ArrayBase](#) is to provide integration with [Conformers](#) machinery - if you wish to use [ArrayProperty](#)'s validation features only, you may safely use it in a custom class. It may define `allow_data_inconsistency` attribute, but it is optional (False is assumed).

```
>>> class CustomDataHolder:
...     allow_data_inconsistency=True # class-level attribute will also work
...     points = ArrayProperty(fill_value=0)
...     def __init__(self, points):
...         self.points = points
...
>>> d = CustomDataHolder(points=((1,2,3),(1,2)))
>>> d.points
masked_array(
  data=[[1.0, 2.0, 3.0],
        [1.0, 2.0, --]],
  mask=[[False, False, False],
        [False, False,  True]],
```

(continues on next page)

(continued from previous page)

fill_value=0)

genre, *filenames*, *values*, and *allow_data_inconsistency* are stored on [ArrayBase](#) subclass automatically, if `super().__init__()` is called. However, if you introduce any new init parameters, you must bind them to the object by yourself. Moreover, if you wish to use [Conformers](#) automatic initialization of [ArrayBase](#) subclasses, you should name those additional parameters with a name of genre you'd like to be retrived or give them a default value, otherwise [Conformers.arrayed\(\)](#) won't know how to initialize such class.

Functions

find_best_shape (jagged)	Find shape of an array, that could fit arbitrarily deep, jagged, nested sequence of sequences.
flatten (items[, depth])	Yield items from any nested iterable as chain of values up to given <i>depth</i> .
longest_subsequences (sequences)	Finds lengths of longest subsequences on each level of given nested sequence.
mask (jagged)	Returns a <code>numpy.array</code> of booleans, of shape that best fits given jagged nested sequence <i>jagged</i> .
to_masked (jagged[, dtype, fill_value])	Convert jagged, arbitrarily deep, nested sequence to <code>numpy.ma.masked_array</code> with missing entries masked.

Classes

ArrayBase (genre, filenames, values[, ...])	Base class for data holding objects.
ArrayProperty (fget, <code>numpy.ndarray</code>] = None, ...)	Property, that validates array-like value given to its setter and stores it as <code>numpy.ndarray</code> .
CollapsibleArrayProperty (fget, ...)	ArrayProperty that stores only one value, if all entries are identical.
DependentParameter (name, kind, genre_getter, ...)	A parameter that depends on the genre of data array.
JaggedArrayProperty (fget, ...)	ArrayProperty for storing intentionally jagged arrays of data.

`tesliper.glassware.array_base.longest_subsequences`(*sequences*: `Sequence[Union[Any, Sequence[Union[Any, NestedSequence]]]]`) → `Tuple[int, ...]`

Finds lengths of longest subsequences on each level of given nested sequence. Each subsequence should have same number of nesting levels.

Parameters **sequences** (*sequence [of sequences [of...]]*) – Arbitrarily deep, nested sequence of sequences.

Returns Length of the longest subsequence for each nesting level as a tuple.

Return type tuple of ints

Notes

If nesting level is not identical in all subsequences, lengths are reported up to first level of non-iterable elements.

```
>>> longest_subsequences([[[1, 2]], [[1], 2]])  
(2,)
```

Examples

```
>>> longest_subsequences([[[1, 2]], [[1]]])  
(1, 2)  
>>> longest_subsequences([[[1, 2]], [[1], [1], [1]]])  
(3, 2)
```

`teslipiper.glassware.array_base.find_best_shape(jagged: Sequence[Union[Any, Sequence[Union[Any, NestedSequence]]]]) → Tuple[int, ...]`

Find shape of an array, that could fit arbitrarily deep, jagged, nested sequence of sequences. Reported size for each level of nesting is the length of the longest subsequence on this level.

Parameters `jagged` (*sequence [of sequences [of ...]]*) – Arbitrarily deep, nested sequence of sequences.

Returns Length of the longest subsequence for each nesting level as a tuple.

Return type tuple of ints

Notes

If nesting level is not identical in all subsequences, size is reported up to first level of non-iterable elements.

```
>>> find_best_shape([[[1, 2]], [[1], 2]])  
(2, 2)
```

Examples

```
>>> find_best_shape([[[1, 2]], [[1]]])  
(2, 1, 2)  
>>> find_best_shape([[[1, 2]], [[1], [1], [1]]])  
(2, 3, 2)
```

`teslipiper.glassware.array_base.flatten(items: Sequence[Union[Any, Sequence[Union[Any, NestedSequence]]]], depth: Optional[int] = None) → Iterator`

Yield items from any nested iterable as chain of values up to given *depth*. If *depth* is `None`, yielded sequence is completely flat.

Parameters

- **items** (*NestedSequence*) – Arbitrarily deep, nested sequence of sequences.
- **depth** (*int, optional*) – How deep should flattening be.

Yields *Any* – Values from *items* as flattened sequence.

`tesliper.glassware.array_base.mask(jagged: Sequence[Union[Any, Sequence[Union[Any, NestedSequence]]]]) → numpy.ndarray`

Returns a numpy.array of booleans, of shape that best fits given jagged nested sequence *jagged*. Each boolean value of the output indicates if corresponding value exists in *jagged*.

Parameters *jagged* (*sequence [of sequences [of ...]]*) – Arbitrarily deep, nested sequence of sequences.

Returns Array of booleans, of shape that best fits *jagged*, indicating if value of same index exist in *jagged*.

Return type numpy.array of bool

Notes

To use output as a mask of `numpy.ma.masked_array`, it should be inverted. `>>> np.ma.array(values, mask=~mask(jagged))`

Examples

```
>>> mask([[1, 2], [1]])
array([[True, True], [True, False]])
>>> mask([[1, 2], []])
array([[True, True], [False, False]])
>>> mask([[1], [], [[2, 3]]])
array([[[True, False], [False, False]], [[True, True], [False, False]]])
```

`tesliper.glassware.array_base.to_masked(jagged: Sequence[Union[Any, Sequence[Union[Any, NestedSequence]]]], dtype: Optional[type] = None, fill_value: Optional[Any] = None) → numpy.ma.core.MaskedArray`

Convert *jagged*, arbitrarily deep, nested sequence to `numpy.ma.masked_array` with missing entries masked.

Parameters

- **jagged** (*sequence [of sequences [of ...]]*) – Arbitrarily deep, nested sequence of sequences.
- **dtype** (*type, optional*) – Data type of the output. If *dtype* is `None`, the type of the data is figured out by numpy machinery.
- **fill_value** (*scalar, optional*) – Value used to fill in the masked values when necessary. If `None`, a default based on the data-type is used.

Returns Given *jagged* converted to `numpy.ma.masked_array` with missing entries masked.

Return type `numpy.ma.core.MaskedArray`

Raises **ValueError** – If *jagged* sequence has inconsistent number of dimensions.

Examples

```
>>> to_masked([[1, 2], [1]])
array(data=[[1, 2], [1, --]], mask=[[True, True], [True, False]])
>>> to_masked([1, [1]])
Traceback (most recent call last):
ValueError: Cannot convert to masked array: jagged sequence has inconsistent
number of dimensions.
```

```
class tesliper.glassware.array_base.ArrayProperty(fget: typing.Optional[typing.Callable[[typing.Any],
numpy.ndarray]] = None, fset:
typing.Optional[typing.Callable[[typing.Any,
typing.Sequence], None]] = None, fdel:
typing.Optional[typing.Callable[[typing.Any],
None]] = None, doc: typing.Optional[str] = None,
dtype: type = <class 'float'>, check_against:
typing.Optional[str] = None, check_depth: int = 1,
fill_value: typing.Any = 0, fsan: typ-
ing.Optional[typing.Callable[[typing.Sequence],
typing.Sequence]] = None)
```

Property, that validates array-like value given to its setter and stores it as `numpy.ndarray`.

Value given to property setter is:

1. (optionally) sanitized with user-provided sanitizer function;
2. (optionally) compared with another array-like attribute of the owner regarding their shape;
3. transformed to `numpy.ndarray` of desired data type;
4. stored in owner's `__dict__`.

Setting, getting and deletion of the value may be customized using standard [setter](#), [getter](#) and [deleter](#) decorators. Additionally, [ArrayProperty](#) provides an [ArrayProperty.sanitizer](#) decorator. If sanitizer function is provided, it is called as a first step of data validation and should return sanitized array-like value (given original value as a positional parameter).

Validation regarding shape of the value is triggered if `check_against` parameter is provided. It should be a name of owner's other array-like attribute as a string. Shape of the value is then compared to the shape of this reference attribute. If shapes are not identical up to the first `check_depth` dimensions, [InconsistentDataError](#) is raised.

Value is always transformed to `numpy.ndarray` of specified `dtype` (float by default.) If such conversion cannot be done because value is a jagged array, [InconsistentDataError](#) will be raised. However, if owner allows for data inconsistency by defining `owner.allow_data_inconsistency = True`, non-matching shapes will be ignored and jagged arrays will be padded with `fill_value` and stored as `numpy.ma.masked_array`.

Parameters

- **fget** – Custom getter for attribute. Default one just returns the stored value.
- **fset** – Custom setter for attribute. Default one stores validated values in instance's `__dict__`.
- **fdel** – Custom deleter for attribute. Deleting attribute is not supported by default.
- **doc** – Attribute's docstring.
- **dtype** – Data type of elements of this array-like attribute.

- **check_against** – Which other instance’s attribute should be used as a reference for array’s shape. If shape of this attribute and reference attribute’s are different, an exception is raised. Only first *check_depth* dimensions are compared.
- **check_depth** – How many dimensions should be compared when checking shape of the array.
- **fill_value** – If values are a jagged array and *instance.allow_data_inconsistency* is *True*, this value will be passed to *numpy.ma.masked_array* constructor as a *fill_value*.
- **fsan** – Custom sanitizer for attribute. “Sanitizer” is here understood as a function that transforms value received by the setter, before the value is validated (checked for corectness) and stored on the instance. *fsan* should return a sanitized value.

getter(*fget*: *Optional[Callable[[Any], Sequence]]*)

Descriptor to change the getter on an *ArrayProperty*.

setter(*fset*: *Optional[Callable[[Any, Sequence], None]]*)

Descriptor to change the setter on an *ArrayProperty*.

deleter(*fdel*: *Optional[Callable[[Any], None]]*)

Descriptor to change the deleter on an *ArrayProperty*.

sanitizer(*fsan*: *Optional[Callable[[Sequence], Sequence]]*)

Descriptor to change the sanitizer on an *ArrayProperty*. Function given as parameter should take one positional argument and return sanitized values. If any sanitizer is provided, it is always called with *values* given to *ArrayProperty* setter. Sanitation is performed before *.check_input()* is called.

check_shape(*instance*: *Any*, *values*: *Sequence*)

Raises an error if *values* have different shape than attribute specified as *check_against*.

check_input(*instance*: *Any*, *values*: *Sequence*) → *numpy.ndarray*

Checks if *values* given to setter have same length as attribute specified with *check_against*.

Parameters

- **instance** – Instance of owner class.
- **values** – Values to validate.

Returns Validated values.

Return type *numpy.ndarray*

Raises

- **ValueError** – If *check_against* is not *None* and list of given values have different length than *getattr(instance, check_against)*. If given list of values cannot be converted to *dtype* type.
- **InconsistentDataError** – If *values* is list of lists of varying size and instance doesn’t allow data inconsistency.

```
class tesliper.glassware.array_base.JaggedArrayProperty(fget: typing.Optional[typing.Callable[[typing.Any],
numpy.ndarray]] = None, fset: typing.Optional[typing.Callable[[typing.Any,
typing.Sequence], None]] = None, fdel:
typing.Optional[typing.Callable[[typing.Any],
None]] = None, doc: typing.Optional[str]
= None, dtype: type = <class 'float'>,
check_against: typing.Optional[str] =
None, check_depth:
int = 1, fill_value: typing.Any = 0, fsan: typing.Optional[typing.Callable[[typing.Sequence],
typing.Sequence]] = None)
```

[ArrayProperty](#) for storing intentionally jagged arrays of data. [InconsistentDataError](#) is only raised if [ArrayProperty.check_shape\(\)](#) fails. Given values are converted to masked array and expanded as needed, regardless value of `allow_data_inconsistency` attribute.

Parameters

- **fget** – Custom getter for attribute. Default one just returns the stored value.
- **fset** – Custom setter for attribute. Default one stores validated values in instance's `__dict__`.
- **fdel** – Custom deleter for attribute. Deleting attribute is not supported by default.
- **doc** – Attribute's docstring.
- **dtype** – Data type of elements of this array-like attribute.
- **check_against** – Which other instance's attribute should be used as a reference for array's shape. If shape of this attribute and reference attribute's are different, an exception is raised. Only first `check_depth` dimensions are compared.
- **check_depth** – How many dimensions should be compared when checking shape of the array.
- **fill_value** – If values are a jagged array and `instance.allow_data_inconsistency` is `True`, this value will be passed to `numpy.ma.masked_array` constructor as a `fill_value`.
- **fsan** – Custom sanitizer for attribute. “Sanitizer” is here understood as a function that transforms value received by the setter, before the value is validated (checked for correctness) and stored on the instance. `fsan` should return a sanitized value.

check_input(*instance: Any, values: Sequence*) → `numpy.ndarray`

Checks if *values* given to setter have same length as attribute specified with `check_against`.

Parameters

- **instance** – Instance of owner class.
- **values** – Values to validate.

Returns Validated values.

Return type `numpy.ndarray`

Raises

- **ValueError** – If `check_against` is not `None` and list of given values have different length than `getattr(instance, check_against)`. If given list of values cannot be converted to `dtype` type.
- **InconsistentDataError** – If `values` is list of lists of varying size and instance doesn't allow data inconsistency.

check_shape(*instance: Any, values: Sequence*)

Raises an error if `values` have different shape than attribute specified as `check_against`.

deleter(*fdel: Optional[Callable[[Any], None]]*)

Descriptor to change the deleter on an [ArrayProperty](#).

getter(*fget: Optional[Callable[[Any], Sequence]]*)

Descriptor to change the getter on an [ArrayProperty](#).

sanitizer(*fsan: Optional[Callable[[Sequence], Sequence]]*)

Descriptor to change the sanitizer on an [ArrayProperty](#). Function given as parameter should take one positional argument and return sanitized values. If any sanitizer is provided, it is always called with `values` given to [ArrayProperty](#) setter. Sanitation is performed before `.check_input()` is called.

setter(*fset: Optional[Callable[[Any, Sequence], None]]*)

Descriptor to change the setter on an [ArrayProperty](#).

```
class tesliper.glassware.array_base.CollapsibleArrayProperty(fget: typing.Optional[typing.Callable[[typing.Any, numpy.ndarray]] = None, fset: typing.Optional[typing.Callable[[typing.Any, typing.Sequence], None]] = None, fdel: typing.Optional[typing.Callable[[typing.Any], None]] = None, doc: typing.Optional[str] = None, dtype: type = <class 'float'>, check_against: typing.Optional[str] = None, check_depth: int = 1, fill_value: typing.Any = 0, fsan: typing.Optional[typing.Callable[[typing.Sequence], typing.Sequence]] = None, strict: bool = False)
```

[ArrayProperty](#) that stores only one value, if all entries are identical.

Parameters

- **fget** – Custom getter for attribute. Default one just returns the stored value.
- **fset** – Custom setter for attribute. Default one stores validated values in instance's `__dict__`.
- **fdel** – Custom deleter for attribute. Deleting attribute is not supported by default.
- **doc** – Attribute's docstring.
- **dtype** – Data type of elements of this array-like attribute.
- **check_against** – Which other instance's attribute should be used as a reference for array's shape. If shape of this attribute and reference attribute's are different, an exception is raised. Only first `check_depth` dimensions are compared.

- **check_depth** – How many dimensions should be compared when checking shape of the array.
- **fill_value** – If values are a jagged array and `instance.allow_data_inconsistency` is `True`, this value will be passed to `numpy.ma.masked_array` constructor as a `fill_value`.
- **fsan** – Custom sanitizer for attribute. “Sanitizer” is here understood as a function that transforms value received by the setter, before the value is validated (checked for correctness) and stored on the instance. `fsan` should return a sanitized value.
- **strict** – Boolean flag indicating if `check_input()` should disallow values that are not all identical. If `strict` is `True` it will raise `InconsistentDataError` when setter is given such values. Defaults to `False`.

check_shape(*instance: Any, values: Sequence*)

Raises an error if *values* have different shape than attribute specified as `check_against`. Accepts values with size of first dimension equal to 1, even if it is not identical to the size of the first dimension of said attribute.

check_input(*instance: Any, values: Union[Sequence, Any]*) → `numpy.ndarray`

If given *values* is not iterable or is of type `str` it is returned without change. Otherwise it is validated using `ArrayProperty.check_input()`, and collapsed to single value if all values are identical. If values are non-uniform and instance doesn’t allow data inconsistency, `InconsistentDataError` is raised.

Parameters

- **instance** – Instance of owner class.
- **values** – Values to validate.

Returns Validated array or single value.

Return type `numpy.ndarray` or any

Raises

- **ValueError** – If `ArrayProperty.check_against` is not `None` and list of given values have different length than `getattr(instance, ArrayProperty.check_against)`. If given list of values cannot be converted to `ArrayProperty.dtype` type.
- **InconsistentDataError** – If *values* is list of lists of varying size and instance doesn’t allow data inconsistency. If property is declared as `strict`, given *values* are non-uniform and instance doesn’t allow data inconsistency.

deleter(*fdel: Optional[Callable[[Any], None]]*)

Descriptor to change the deleter on an `ArrayProperty`.

getter(*fget: Optional[Callable[[Any], Sequence]]*)

Descriptor to change the getter on an `ArrayProperty`.

sanitizer(*fsan: Optional[Callable[[Sequence], Sequence]]*)

Descriptor to change the sanitizer on an `ArrayProperty`. Function given as parameter should take one positional argument and return sanitized values. If any sanitizer is provided, it is always called with *values* given to `ArrayProperty` setter. Sanitation is performed before `.check_input()` is called.

setter(*fset: Optional[Callable[[Any, Sequence], None]]*)

Descriptor to change the setter on an `ArrayProperty`.

class `tesliper.glassware.array_base.ArrayBase`(*genre: str, filenames: Sequence[str], values: Sequence, allow_data_inconsistency: bool = False*)

Base class for data holding objects.

It provides an automatic registration of its subclasses as a *DataArray*-like representations of all *associated_genres* declared by said subclass. A subclass should provide an *associated_genres* class attribute, even if it's not supposed to be directly instantiated with data for any genre, it should be an empty tuple in such case. Otherwise, *associated_genres* should be a tuple of genre names as strings.

This base class provides the most basic set of attributes, a *DataArray*-like object should implement, listed in the Parameters section.

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers' identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see *ArrayProperty* for details).

abstract property associated_genres

Genres associated with subclassing class.

Should be provided by subclass as class-level attribute. It will be used to determine what class to use to represent data of particular genre when requested *via* *Conforemrs.arrayed()* method. May be an empty sequence, if subclass is not intended to be used directly by *tesliper*'s machinery.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

```
class tesliper.glassware.array_base.DependentParameter(name: str, kind: inspect._ParameterKind,
                                                    genre_getter: Callable[[str], str], *, default:
                                                    Any, annotation: Any)
```

A parameter that depends on the genre of data array. It provides a *_genre_getter* callable attribute that is used to provide a name of data genre that should be used for this parameter.

It is hashable as the original *inspect.Parameter*, however it must be remembered that Python hashes functions based on their identity.

property genre_getter

Should be a function that given a genre of data array being instantiated, returns a genre that should be used for this parameter.

classmethod from_parameter(parameter: inspect.Parameter, genre_getter: Callable[[str], str])

Casts given *inspect.Parameter* instance to this class.

empty

alias of *inspect._empty*

```
replace(*, name=<class 'inspect._void'>, kind=<class 'inspect._void'>, genre_getter=<class
'inspect._void'>, annotation=<class 'inspect._void'>, default=<class 'inspect._void'>)
```

Creates a customized copy of the *DependentParameter*.

tesliper.glassware.arrays

Implements *DataArray*-like objects for handling arrayed data.

DataArray-like objects are concrete implementations of *ArrayBase* base class that collect specific data for multiple conformers and provide an easy access to genre-specific functionality. Instances of *DataArray* subclasses are produced by the *Conformers.arrayed()* method and Tesliper's subscription mechanism.

Classes

<i>Averagable</i> ()	Mix-in for <i>DataArray</i> subclasses, that may be averaged based on populations of conformers.
<i>Bands</i> (genre, filenames, values[, ...])	Special kind of data array for band values, to which spectral data or activities correspond.
<i>BooleanArray</i> (genre, filenames, values[, ...])	For handling data of <code>bool</code> type.
<i>DataArray</i> (genre, filenames, values[, ...])	Base class for data holding objects.
<i>ElectronicActivities</i> (genre, filenames, ...)	For handling electronic spectral activity data.
<i>ElectronicData</i> (genre, filenames, values, wavenum)	For handling electronic data that is not a spectral activity.
<i>Energies</i> (genre, filenames, values[, t, ...])	For handling data about the energy of conformers.
<i>FilenamesArray</i> ([genre, filenames, values, ...])	Special case of <i>DataArray</i> , holds only filenames.
<i>FloatArray</i> (genre, filenames, values[, ...])	For handling data of <code>float</code> type.
<i>Geometry</i> (genre, filenames, values, atoms[, ...])	For handling information about geometry of conformers.
<i>InfoArray</i> (genre, filenames, values[, ...])	For handling data of <code>str</code> type.
<i>IntegerArray</i> (genre, filenames, values[, ...])	For handling data of <code>int</code> type.
<i>ScatteringActivities</i> (genre, filenames, ...)	For handling scattering spectral activity data.
<i>ScatteringData</i> (genre, filenames, values, freq)	For handling scattering data that is not a spectral activity.
<i>SpectralActivities</i> (genre, filenames, values)	Base class for spectral activities genres.
<i>SpectralData</i> (genre, filenames, values[, ...])	Base class for spectral data genres, that are not spectral activities.
<i>Transitions</i> (genre, filenames, values[, ...])	For handling information about electronic transitions from ground to excited state contributing to each band.
<i>VibrationalActivities</i> (genre, filenames, ...)	For handling electronic spectral activity data.
<i>VibrationalData</i> (genre, filenames, values, freq)	For handling vibrational data that is not a spectral activity.

class tesliper.glassware.arrays.**DataArray**(genre: str, filenames: Sequence[str], values: Sequence, allow_data_inconsistency: bool = False)

Base class for data holding objects.

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers' identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see *ArrayProperty* for details).

classmethod **get_init_params**() → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class tesliper.glassware.arrays.**IntegerArray**(*genre: str, filenames: Sequence[str], values: Sequence, allow_data_inconsistency: bool = False*)

For handling data of `int` type.

Table 19: Genres associated with this class:

charge	multiplicity
--------	--------------

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers' identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

classmethod **get_init_params()** → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class tesliper.glassware.arrays.**FloatArray**(*genre: str, filenames: Sequence[str], values: Sequence, allow_data_inconsistency: bool = False*)

For handling data of `float` type.

Table 20: Genres associated with this class:

zpecorr	tencorr	entcorr	gibcorr
---------	---------	---------	---------

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers' identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

classmethod **get_init_params()** → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class tesliper.glassware.arrays.**InfoArray**(*genre: str, filenames: Sequence[str], values: Sequence, allow_data_inconsistency: bool = False*)

For handling data of `str` type.

Table 21: Genres associated with this class:

command	stoichiometry
---------	---------------

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

classmethod `get_init_params()` → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class `tesliper.glassware.arrays.FilenamesArray`(*genre: str = 'filenames', filenames: Union[Sequence, numpy.ndarray] = (), values: Optional[Any] = None, allow_data_inconsistency: bool = False*)

Special case of [DataArray](#), holds only filenames. *values* property returns same as *filenames* and ignores any value given to its setter. Only genre associated with this class is *filenames* pseudo-genre.

Parameters

- **genre** (*str*) – Name of genre, should be ‘filenames’.
- **filenames** (*numpy.ndarray(dtype=str)*) – List of filenames of gaussian output files, from which data were extracted.
- **values** (*numpy.ndarray(dtype=str)*) – Always returns same as *filenames*.

property values

Property, that validates array-like value given to its setter and stores it as `numpy.ndarray`.

Value given to property setter is:

1. (optionally) sanitized with user-provided sanitizer function;
2. (optionally) compared with another array-like attribute of the owner regarding their shape;
3. transformed to `numpy.ndarray` of desired data type;
4. stored in owner’s `__dict__`.

Setting, getting and deletion of the value may be customized using standard [setter](#), [getter](#) and [deleter](#) decorators. Additionally, [ArrayProperty](#) provides an [ArrayProperty.sanitizer](#) decorator. If sanitizer function is provided, it is called as a first step of data validation and should return sanitized array-like value (given original value as a positional parameter).

Validation regarding shape of the value is triggered if *check_against* parameter is provided. It should be a name of owner’s other array-like attribute as a string. Shape of the value is then compared to the shape of this reference attribute. If shapes are not identical up to the first *check_depth* dimensions, [InconsistentDataError](#) is raised.

Value is always transformed to `numpy.ndarray` of specified *dtype* (float by default.) If such conversion cannot be done because value is a jagged array, [InconsistentDataError](#) will be raised. However, if owner allows for data inconsistency by defining `owner.allow_data_inconsistency = True`, non-matching shapes will be ignored and jagged arrays will be padded with *fill_value* and stored as `numpy.ma.masked_array`.

classmethod `get_init_params()` → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class `tesliper.glassware.arrays.BooleanArray`(*genre: str, filenames: Sequence[str], values: Sequence, allow_data_inconsistency: bool = False*)

For handling data of bool type.

Table 22: Genres associated with this class:

normal_termination	optimization_completed
--------------------	------------------------

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers' identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

classmethod `get_init_params()` → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class `tesliper.glassware.arrays.Energies`(*genre, filenames, values, t=298.15, allow_data_inconsistency=False*)

For handling data about the energy of conformers.

Table 23: Genres associated with this class:

scf	zpe	ten	ent	gib
-----	-----	-----	-----	-----

Parameters

- **genre** (*str*) – genre of energy.
- **filenames** (*numpy.ndarray(dtype=str)*) – List of filenames of gaussian output files, from which data were extracted.
- **values** (*numpy.ndarray(dtype=float)*) – Energy value for each conformer.
- **t** (*int or float*) – Temperature of calculated state in K.

property `as_kcal_per_mol`

Energy values converted to kcal/mol.

property `deltas`

Calculates energy difference between each conformer and lowest energy conformer. Converts energy to kcal/mol.

Returns List of energy differences from lowest energy in kcal/mol.

Return type `numpy.ndarray`

property min_factors

Calculates list of conformers' Boltzmann factors respective to lowest energy conformer in system.

Notes

Boltzmann factor of two states is defined as: $F(\text{state}_1)/F(\text{state}_2) = \exp((E_1 - E_2)/kt)$ where E_1 and E_2 are energies of states 1 and 2, k is Boltzmann constant, $k = 0.0019872041 \text{ kcal}/(\text{mol}\cdot\text{K})$, and t is temperature of the system.

Returns List of conformers' Boltzmann factors respective to lowest energy conformer.

Return type `numpy.ndarray`

property populations

Calculates Boltzmann distribution of conformers.

Returns List of conformers populations calculated as Boltzmann distribution.

Return type `numpy.ndarray`

calculate_populations(*t*)

Calculates conformers' Boltzmann distribution in given temperature.

Parameters *t* (*int* or *float*) – Temperature of calculated state in K.

classmethod `get_init_params()` → `Dict[str, Union[str, inspect.Parameter]]`

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → `Dict[str, Any]`

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class `tesliper.glassware.arrays.Averagable`

Mix-in for `DataArray` subclasses, that may be averaged based on populations of conformers.

average_conformers(*energies*) → `tesliper.glassware.arrays.DataArray`

A method for averaging values by population of conformers.

Parameters *energies* (*Energies* or *iterable*) – Object with *populations* and *genre* attributes, containing respectively: list of populations values as `numpy.ndarray` and string specifying energy type. Alternatively, list of weights for each conformer.

Returns New instance of `DataArray`'s subclass, on which *average* method was called, containing averaged values.

Return type `DataArray`

Raises **TypeError** – If creation of an instance based on its `__init__` signature is impossible.

class `tesliper.glassware.arrays.Bands`(*genre*: *str*, *filenames*: *Sequence[str]*, *values*: *Sequence*, *allow_data_inconsistency*: *bool* = *False*)

Special kind of data array for band values, to which spectral data or activities correspond. Provides an easy way to convert values between their different representations: frequency, wavelength, and excitation energy.

Table 24: Genres associated with this class:

freq	wavelen	ex_en
------	---------	-------

Parameters

- **genre** – Name of the data genre that *values* represent.

- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

property freq

Values converted to frequencies in cm^{-1} .

property frequencies

Values converted to frequencies in cm^{-1} . A convenience alias for [Bands.frequencies](#).

property wavelen

Values converted to wavelengths in nm.

property wavelengths

Values converted to wavelengths in nm. A convenience alias for [Bands.wavelen](#).

property ex_en

Values converted to excitation energy in eV.

property excitation_energy

Values converted to excitation energy in eV. A convenience alias for [Bands.ex_en](#).

property imaginary

Finds number of imaginary frequencies of each conformer.

Returns Number of imaginary frequencies of each conformer.

Return type numpy.ndarray

find_imaginary()

Reports number of imaginary frequencies of each conformer that has any.

Returns Dictionary of {filename: number-of-imaginary-frequencies} for each conformer with at least one imaginary frequency.

Return type dict

classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class tesliper.glassware.arrays.**SpectralData**(*genre: str, filenames: Sequence[str], values: Sequence, allow_data_inconsistency: bool = False*)

Base class for spectral data genres, that are not spectral activities.

When subclassed, one of the attributes: [freq](#) or [wavelen](#) should be overridden with a concrete setter and getter - use of [ArrayProperty](#) is recommended. The other one may use implementation from this base class by call to `super().freq` or `super().wavelen` to get converted values.

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.

- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

abstract property spectra_type

Type of spectra, that genres associated with [SpectralData](#)’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

abstract property freq

Bands values converted to frequencies in cm^{-1} . If [wavelen](#) is provided, this may be overridden with a simple call to `super()`:

```
@property
def freq(self):
    return super().freq() # values converted to cm-1
```

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for [freq](#).

abstract property wavelen

Bands values converted to wavelengths in nm. If [freq](#) is provided, this may be overridden with a simple call to `super()`:

```
@property
def wavelen(self):
    return super().wavelen() # values converted to nm
```

property wavelengths

Bands values converted to wavelengths in nm. A convenience alias for [wavelen](#).

classmethod `get_init_params()` → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class `tesliper.glassware.arrays.VibrationalData(genre, filenames, values, freq, allow_data_inconsistency=False)`

For handling vibrational data that is not a spectral activity.

Table 25: Genres associated with this class:

mass	frc	emang
------	-----	-------

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **freq** – Frequency for each value in each conformer in cm^{-1} units.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

property spectra_type

Type of spectra, that genres associated with *SpectralData*’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for *freq*.

classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

property wavelen

Bands values converted to wavelengths in nm.

property wavelengths

Bands values converted to wavelengths in nm. A convenience alias for *wavelen*.

class tesliper.glassware.arrays.**ScatteringData**(*genre*, *filenames*, *values*, *freq*, *t*=298.15, *laser*=532, *allow_data_inconsistency*=False)

For handling scattering data that is not a spectral activity.

Table 26: Genres associated with this class:

depolarp	depolaru	depp	depu	alpha2
beta2	alphag	gamma2	delta2	cid1
cid2	cid3	rc180		

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **freq** – Frequency for each value in each conformer in cm^{-1} units.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see *ArrayPropety* for details).

property spectra_type

Type of spectra, that genres associated with *SpectralData*’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for *freq*.

classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

property wavelen

Bands values converted to wavelengths in nm.

property wavelengths

Bands values converted to wavelengths in nm. A convenience alias for [wavelen](#).

```
class tesliper.glassware.arrays.ElectronicData(genre, filenames, values, wavelen,
                                              allow_data_inconsistency=False)
```

For handling electronic data that is not a spectral activity.

Table 27: Genres associated with this class:

eemang

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

property spectra_type

Type of spectra, that genres associated with [SpectralData](#)’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

property freq

Bands values converted to frequencies in cm^{-1} . If [wavelen](#) is provided, this may be overridden with a simple call to `super()`:

```
@property
def freq(self):
    return super().freq() # values converted to cm^(-1)
```

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for [freq](#).

```
classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter]]
```

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

```
get_repr_args() → Dict[str, Any]
```

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

property wavelengths

Bands values converted to wavelengths in nm. A convenience alias for [wavelen](#).

```
class tesliper.glassware.arrays.SpectralActivities(genre: str, filenames: Sequence[str], values:
                                                  Sequence, allow_data_inconsistency: bool =
                                                  False)
```

Base class for spectral activities genres.

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.

- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

property intensities

Converts spectral activity calculated by quantum chemistry software to signal intensity.

Returns Signal intensities for each conformer.

Return type `numpy.ndarray`

Raises **NotImplementedError** – if genre does not provide values conversion to intensities.

average_conformers(*energies*) → [tesliper.glassware.arrays.DataArray](#)

A method for averaging values by population of conformers.

Parameters **energies** ([Energies](#) or *iterable*) – Object with **populations** and **genre** attributes, containing respectively: list of populations values as `numpy.ndarray` and string specifying energy type. Alternatively, list of weights for each conformer.

Returns New instance of `DataArray`’s subclass, on which *average* method was called, containing averaged values.

Return type [DataArray](#)

Raises **TypeError** – If creation of an instance based on its `__init__` signature is impossible.

abstract property freq

Bands values converted to frequencies in cm^{-1} . If *wavelen* is provided, this may be overridden with a simple call to `super()`:

```
@property
def freq(self):
    return super().freq() # values converted to cm^(-1)
```

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for *freq*.

classmethod **get_init_params**() → `Dict[str, Union[str, inspect.Parameter]]`

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → `Dict[str, Any]`

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

abstract property spectra_type

Type of spectra, that genres associated with [SpectralData](#)’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

abstract property wavelen

Bands values converted to wavelengths in nm. If *freq* is provided, this may be overridden with a simple call to `super()`:

```
@property
def wavelen(self):
    return super().wavelen() # values converted to nm
```

property wavelengths

Bands values converted to wavelengths in nm. A convenience alias for *wavelen*.

```
class tesliper.glassware.arrays.VibrationalActivities(genre, filenames, values, freq,  
                                                    allow_data_inconsistency=False)
```

For handling electronic spectral activity data.

Table 28: Genres associated with this class:

iri	dip	rot
-----	-----	-----

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **freq** – Frequency for each value in each conformer in cm^{-1} units.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see `ArrayProperty` for details).

average_conformers(*energies*) → *tesliper.glassware.arrays.DataArray*

A method for averaging values by population of conformers.

Parameters **energies** (*Energies* or *iterable*) – Object with *populations* and *genre* attributes, containing respectively: list of populations values as `numpy.ndarray` and string specifying energy type. Alternatively, list of weights for each conformer.

Returns New instance of `DataArray`’s subclass, on which *average* method was called, containing averaged values.

Return type *DataArray*

Raises **TypeError** – If creation of an instance based on its `__init__` signature is impossible.

calculate_spectra(*start, stop, step, width, fitting*)

Calculates spectrum for each individual conformer.

Parameters

- **start** (*int* or *float*) – Number representing start of spectral range in relevant units.
- **stop** (*int* or *float*) – Number representing end of spectral range in relevant units.
- **step** (*int* or *float*) – Number representing step of spectral range in relevant units.
- **width** (*int* or *float*) – Number representing half width of maximum peak height.
- **fitting** (*function*) – Function, which takes spectral data, freqs, abscissa, width as parameters and returns `numpy.array` of calculated, non-corrected spectrum points.

Returns Calculated spectrum.

Return type *SingleSpectrum*

Raises **ValueError** – If given *start*, *stop*, and *step* values would produce an empty or one-element sequence; i.e. if *start* is greater than *stop* or if *start* - *stop* < *step*, assuming *step* is a positive value.

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for *freq*.

classmethod `get_init_params()` → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

property intensities

Converts spectral activity calculated by quantum chemistry software to signal intensity.

Returns Signal intensities for each conformer.

Return type numpy.ndarray

Raises `NotImplementedError` – if genre does not provide values conversion to intensities.

property spectra_type

Type of spectra, that genres associated with *SpectralData*’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

property wavelen

Bands values converted to wavelengths in nm.

property wavelengths

Bands values converted to wavelengths in nm. A convenience alias for *wavelen*.

class `tesliper.glassware.arrays.ScatteringActivities`(*genre, filenames, values, freq, t=298.15, laser=532, allow_data_inconsistency=False*)

For handling scattering spectral activity data.

Table 29: Genres associated with this class:

ramanactiv	ramact	raman1	roa1
raman2	roa2	raman3	roa3

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **freq** – Frequency for each value in each conformer in cm^{-1} units.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see `ArrayPropety` for details).

property intensities

Converts spectral activity calculated by quantum chemistry software to signal intensity.

Returns Signal intensities for each conformer.

Return type numpy.ndarray

Raises `NotImplementedError` – if genre does not provide values conversion to intensities.

average_conformers(*energies*) → *tesliper.glassware.arrays.DataArray*

A method for averaging values by population of conformers.

Parameters *energies* (*Energies* or *iterable*) – Object with populations and genre attributes, containing respectively: list of populations values as numpy.ndarray and string specifying energy type. Alternatively, list of weights for each conformer.

Returns New instance of `DataArray`’s subclass, on which *average* method was called, containing averaged values.

Return type *DataArray*

Raises **TypeError** – If creation of an instance based on its `__init__` signature is impossible.

calculate_spectra(*start, stop, step, width, fitting*)

Calculates spectrum for each individual conformer.

Parameters

- **start** (*int or float*) – Number representing start of spectral range in relevant units.
- **stop** (*int or float*) – Number representing end of spectral range in relevant units.
- **step** (*int or float*) – Number representing step of spectral range in relevant units.
- **width** (*int or float*) – Number representing half width of maximum peak height.
- **fitting** (*function*) – Function, which takes spectral data, freqs, abscissa, width as parameters and returns `numpy.array` of calculated, non-corrected spectrum points.

Returns Calculated spectrum.

Return type *SingleSpectrum*

Raises **ValueError** – If given *start*, *stop*, and *step* values would produce an empty or one-element sequence; i.e. if *start* is greater than *stop* or if *start* - *stop* < *step*, assuming *step* is a positive value.

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for *freq*.

classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

property spectra_type

Type of spectra, that genres associated with *SpectralData*’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

property wavelen

Bands values converted to wavelengths in nm.

property wavelengths

Bands values converted to wavelengths in nm. A convenience alias for *wavelen*.

class tesliper.glassware.arrays.ElectronicActivities(*genre, filenames, values, wavelen, allow_data_inconsistency=False*)

For handling electronic spectral activity data.

Table 30: Genres associated with this class:

vdip	ldip	vrot	lrot	vosc	losc
------	------	------	------	------	------

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.

- **values** – Sequence of values for *genre* for each conformer in *filenames*.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see [ArrayProperty](#) for details).

property intensities

Converts spectral activity calculated by quantum chemistry software to signal intensity.

Returns Signal intensities for each conformer.

Return type `numpy.ndarray`

Raises **NotImplementedError** – if *genre* does not provide values conversion to intensities.

calculate_spectra(*start, stop, step, width, fitting*)

Calculates spectrum for each individual conformer.

Parameters

- **start** (*int* or *float*) – Number representing start of spectral range in relevant units.
- **stop** (*int* or *float*) – Number representing end of spectral range in relevant units.
- **step** (*int* or *float*) – Number representing step of spectral range in relevant units.
- **width** (*int* or *float*) – Number representing half width of maximum peak height.
- **fitting** (*function*) – Function, which takes spectral data, freqs, abscissa, width as parameters and returns `numpy.array` of calculated, non-corrected spectrum points.

Returns Calculated spectrum.

Return type [SingleSpectrum](#)

Raises **ValueError** – If given *start*, *stop*, and *step* values would produce an empty or one-element sequence; i.e. if *start* is greater than *stop* or if *start* - *stop* < *step*, assuming *step* is a positive value.

average_conformers(*energies*) → [tesliper.glassware.arrays.DataArray](#)

A method for averaging values by population of conformers.

Parameters **energies** ([Energies](#) or *iterable*) – Object with *populations* and *genre* attributes, containing respectively: list of populations values as `numpy.ndarray` and string specifying energy type. Alternatively, list of weights for each conformer.

Returns New instance of `DataArray`'s subclass, on which *average* method was called, containing averaged values.

Return type [DataArray](#)

Raises **TypeError** – If creation of an instance based on its `__init__` signature is impossible.

property freq

Bands values converted to frequencies in cm^{-1} . If *wavelen* is provided, this may be overridden with a simple call to `super()`:

```
@property
def freq(self):
    return super().freq() # values converted to cm^(-1)
```

property frequencies

Bands values converted to frequencies in cm^{-1} . A convenience alias for [freq](#).

classmethod `get_init_params()` → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

property `spectra_type`

Type of spectra, that genres associated with *SpectralData*’s subclass relate to. Should be a class-level attribute with value of either “vibrational”, “electronic”, or “scattering”.

property `wavelengths`

Bands values converted to wavelengths in nm. A convenience alias for *wavelen*.

class `tesliper.glassware.arrays.Transitions`(*genre: str, filenames: Sequence[str], values: Sequence[Sequence[Sequence[Tuple[int, int, float]]]], allow_data_inconsistency: bool = False*)

For handling information about electronic transitions from ground to excited state contributing to each band.

Data is stored in three attributes: *ground*, *excited*, and *values*, which are respectively: list of ground state electronic subshells, list of excited state electronic subshells, and list of coefficients of transitions from corresponding ground to excited subshell. Each of these arrays is of shape (conformers, bands, max_transitions), where ‘max_transitions’ is a highest number of transitions contributing to single band across all bands of all conformers.

Table 31: Genres associated with this class:

transitions

values

List of coefficients of each transition. It is a 3-dimensional of shape (conformers, bands, max_transitions).

Type `numpy.ndarray(dtype=float)`

ground

List of ground state electronic subshells, stored as integers assigned to them by used quantum computations program. It is a 3-dimensional array of shape (conformers, bands, max_transitions).

Type `numpy.ndarray(dtype=int)`

excited

List of excited state electronic subshells, stored as integers assigned to them by used quantum computations program. It is a 3-dimensional array of shape (conformers, bands, max_transitions).

Type `numpy.ndarray(dtype=int)`

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers’ identifiers.
- **values** (*list of lists of lists of tuples of (int, int, float)*) – Transitions data (ground and excited state electronic subshell and coefficient of transition from former to latter) for each transition of each band of each conformer.
- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see `ArrayPropety` for details).

static unpack_values(*values*: Sequence[Sequence[Sequence[Tuple[int, int, float]]]])

Unpack transitions data stored as list of tuples of (ground, excited, coefficient) to separate lists for each information piece, keeping original dimensionality (conformers, bands, transitions).

Parameters *values* (list of lists of lists of tuples of (int, int, float)) – Transitions data (ground and excited state electronic subshell and coefficient of transition from former to latter) for each transition of each band of each conformer.

Returns

- list of lists of lists of int,
- list of lists of lists of int,
- list of lists of lists of float – Transitions data separated to lists of ground, excited, and coefficients, for each transition of each band of each conformer.

property coefficients: numpy.ndarray

Coefficients of each transition, alias for *values*.

property contribution: numpy.ndarray

Contribution of each transition to given band, calculated as $2 * \text{coef}^2$. To get values in percent, multiply by 100.

property indices_highest: Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Indices of coefficients of highest contribution to band in form that can be used in numpy's advanced indexing mechanism.

property highest_contribution: Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]

Electronic transitions data limited to transition of highest contribution to each band. Returns tuple with 4 arrays: ground and excited state electronic subshell, coefficient of transition from former to latter, and its contribution, for each band of each conformer.

classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

class teslipper.glassware.arrays.**Geometry**(*genre*: str, *filenames*: Sequence[str], *values*: Sequence[Sequence[Sequence[float]]], *atoms*: Union[Sequence[Union[int, str]], Sequence[Sequence[Union[int, str]]]], *allow_data_inconsistency*: bool = False)

For handling information about geometry of conformers.

Table 32: Genres associated with this class:

last_read_geom	input_geom	optimized_geom
----------------	------------	----------------

Parameters

- **genre** – Name of the data genre that *values* represent.
- **filenames** – Sequence of conformers' identifiers.
- **values** – List of x, y, z coordinated for each conformer, for each atom.

- **allow_data_inconsistency** – Flag signaling if instance should allow data inconsistency (see `ArrayPropety` for details). False by default.
- **atoms** – List of atomic numbers representing atoms in conformer, one for each coordinate. Should be a list of integers or list of strings, that can be interpreted as integers or symbols of atoms. May also be a list of such lists - one list of atoms for each conformer. All those lists should be identical in such case, otherwise `InconsistentDataError` is raised. Only one list of atoms is stored in either case.

get_repr_args() → Dict[str, Any]

Returns dictionary that can be used as keyword-value pairs to instantiate identical object.

classmethod get_init_params() → Dict[str, Union[str, inspect.Parameter, *tesliper.glassware.array_base.DependentParameter*]]

Returns parameters used to instantiate this class. *genre* is a genre of data array that is to be instantiated.

tesliper.glassware.conformers

A tesliper's main data storage.

Classes

Conformers(*args[, ...])

Container for data extracted from quantum chemical software output files.

class tesliper.glassware.conformers.Conformers(*args, allow_data_inconsistency: bool = False, temperature_of_the_system: float = 298.15, **kwargs)

Container for data extracted from quantum chemical software output files.

Data for each file is stored in the underlying `OrderedDict`, under the key of said file's name. Its values are dictionaries with genres name (as key) and appropriate data pairs. Beside this, its essential functionality is transformation of stored data to corresponding `DataArray` objects with use of `arrayed()` method. It provides some control over this transformation, especially in terms of including/excluding particular conformers' data on creation of new `DataArray` instance. This type of control is here called trimming. Trimming can be achieved by use of various *trim* methods defined in this class or by direct changes to *kept* attribute. See its documentation for more information.

primary_genres

Class attribute. Data genres considered most important, used as default when checking for conformers completeness (see `trim_incomplete()` method).

Notes

Inherits from `collections.OrderedDict`.

Parameters

- ***args** – list of arguments for creation of underlying dictionary
- **allow_data_inconsistency** (*bool*, *optional*) – specifies if data inconsistency should be allowed in created `DataArray` object instances, defaults to False
- **temperature_of_the_system** (*float*, *optional*) – Temperature of the system in Kelvin units, must be zero or higher. Defaults to room temperature = 298.15 K.

- ****kwargs** – list of arbitrary keyword arguments for creation of underlying dictionary

property temperature: float

Temperature of the system expressed in Kelvin units.

Value of this parameter is passed to *data arrays* created with the *arrayed()* method, provided that the target data array class supports a parameter named *t* in it's constructor.

New in version 0.9.1.

Raises ValueError – if set to a value lower than zero.

clear()

Remove all items from the Conformers instance.

popitem(*last=True*)

Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

move_to_end(*key, last=True*)

Move an existing element to the end (or beginning if last==False).

Raises KeyError if the element does not exist.

copy() → a shallow copy of conformers

property kept

List of booleans, one for each conformer stored, defining if particular conformers data should be included in corresponding DataArray instance, created by *arrayed()* method. It may be changed by use of trim methods, by setting its value directly, or by modification of the underlying list. For the first option refer to those methods documentation, for rest see the Examples section.

Returns List of booleans, one for each conformer stored, defining if particular conformers data should be included in corresponding DataArray instance.

Return type list of bool

Raises

- **TypeError** – If assigned values is not a sequence. If elements of given sequence are not one of types: bool, int, str.
- **ValuesError** – If number of given boolean values doesn't match number of contained conformers.
- **KeyError** – If any of given string values is not in underlying dictionary keys.
- **IndexError** – If any of given integer values is not in range $0 \leq i < \text{number of conformers}$.

Examples

New list of values can be set in a few ways. Firstly, it is the most straightforward to just assign a new list of boolean values to the *kept* attribute. This list should have the same number of elements as the number of conformers contained. A ValueError is raised if it doesn't.

```
>>> c = Conformers(one={}, two={}, tree={})
>>> c.kept
[True, True, True]
>>> c.kept = [False, True, False]
```

(continues on next page)

(continued from previous page)

```
>>> c.kept
[False, True, False]
>>> c.kept = [False, True, False, True]
Traceback (most recent call last):
...
ValueError: Must provide boolean value for each known conformer.
4 values provided, 3 expected.
```

Secondly, list of filenames of conformers intended to be kept may be given. Only these conformers will be kept. If given filename is not in the underlying Conformers' dictionary, `KeyError` is raised.

```
>>> c.kept = ['one']
>>> c.kept
[True, False, False]
>>> c.kept = ['two', 'other']
Traceback (most recent call last):
...
KeyError: Unknown conformers: other.
```

Thirdly, list of integers representing conformers indices may be given. Only conformers with specified indices will be kept. If one of given integers can't be translated to conformer's index, `IndexError` is raised. Indexing with negative values is not supported currently.

```
>>> c.kept = [1, 2]
>>> c.kept
[False, True, True]
>>> c.kept = [2, 3]
Traceback (most recent call last):
...
IndexError: Indexes out of bounds: 3.
```

Fourthly, assigning `True` or `False` to this attribute will mark all conformers as kept or not kept respectively.

```
>>> c.kept = False
>>> c.kept
[False, False, False]
>>> c.kept = True
>>> c.kept
[True, True, True]
```

Lastly, list of kept values may be modified by setting its elements to `True` or `False`. It is advised against, however, as mistake such as `c.kept[:2] = [True, False, False]` will break some functionality by forcibly changing size of `kept` list.

Notes

Type of the first element of given sequence is used for dynamic dispatch.

update(*other=None, **kwargs*)

Works like `dict.update`, but if key is already present, it updates dictionary associated with given key rather than assigning new value. Keys of dictionary passed as positional parameter (or additional keyword arguments given) should be conformers' identifiers and its values should be dictionaries of {"genre": values} for those conformers.

Please note, that values of status genres like 'optimization_completed' and 'normal_termination' will be updated as well for such key, if are present in given new values.

arrayed(*genre: str, full: bool = False, strict: bool = True, **kwargs*) →

Union[*tesliper.glassware.arrays.DataArray*, *tesliper.glassware.arrays.Energies*, *tesliper.glassware.arrays.FloatArray*, *tesliper.glassware.arrays.FilenamesArray*, *tesliper.glassware.arrays.InfoArray*, *tesliper.glassware.arrays.BooleanArray*, *tesliper.glassware.arrays.IntegerArray*, *tesliper.glassware.arrays.Bands*, *tesliper.glassware.arrays.VibrationalData*, *tesliper.glassware.arrays.ScatteringData*, *tesliper.glassware.arrays.ElectronicData*, *tesliper.glassware.arrays.VibrationalActivities*, *tesliper.glassware.arrays.ScatteringActivities*, *tesliper.glassware.arrays.ElectronicActivities*, *tesliper.glassware.arrays.Transitions*, *tesliper.glassware.arrays.Geometry*]

Lists requested data and returns as appropriate *DataArray* instance.

New in version 0.9.1: The *strict* parameter.

Parameters

- **genre** – String representing data genre. Must be one of known genres.
- **full** – Boolean indicating if full set of data should be taken, ignoring any trimming conducted earlier. Defaults to `False`.
- **strict** – Boolean indicating if additional kwargs that doesn't match signature of data array's constructor should cause an exception as normally (`strict = True`) or be silently ignored (`strict = False`). Defaults to `True`.
- **kwargs** – Additional keyword parameters passed to data array constructor. Any explicitly given parameters will take precedence over automatically retrieved and default values.

Returns Arrayed data of desired genre as appropriate *DataArray* object.

Return type *DataArray*

Notes

For now, the special "filenames" genre always ignores *kwargs*.

by_index(*index: int*) → dict

Returns data for conformer on desired index.

key_of(*index: int*) → str

Returns name of conformer associated with given index.

index_of(*key: str*) → int

Return index of given key.

has_genre(*genre: str, ignore_trimming: bool = False*) → bool

Checks if any of stored conformers contains data of given genre.

Parameters

- **genre** (*str*) – Name of genre to test.
- **ignore_trimming** (*bool*) – If all known conformers should be considered (*ignore_trimming = True*) or only kept ones (*ignore_trimming = False*, default).

Returns Boolean value indicating if any of stored conformers contains data of genre in question.

Return type bool

has_any_genre(*genres: Iterable[str], ignore_trimming: bool = False*) → bool

Checks if any of stored conformers contains data of any of given genres.

Parameters

- **genres** (*iterable of str*) – List of names of genres to test.
- **ignore_trimming** (*bool*) – If all known conformers should be considered (*ignore_trimming = True*) or only kept ones (*ignore_trimming = False*, default).

Returns Boolean value indicating if any of stored conformers contains data of any of genres in question.

Return type bool

all_have_genres(*genres: Iterable[str], ignore_trimming: bool = False*) → bool

Checks if all stored conformers contains data of given genres.

Parameters

- **genres** (*iterable of str*) – List of names of genres to test.
- **ignore_trimming** (*bool*) – If all known conformers should be considered (*ignore_trimming = True*) or only kept ones (*ignore_trimming = False*, default).

Returns Boolean value indicating if each stored conformers contains data of all genres in question.

Return type bool

trim_incomplete(*wanted: Optional[Iterable[str]] = None, strict: bool = False*) → None

Mark incomplete conformers as “not kept”.

Conformers that does not contain one or more data genres specified as *wanted* will be marked as “not kept”. If *wanted* parameter is not given, it evaluates to [primary_genres](#). If no conformer contains all *wanted* genres, conformers that match the specification most closely are kept. The “closeness” is defined by number of conformer’s genres matching *wanted* genres in the first place (the more, the better) and the position of particular genre in *wanted* list in the second place (the closer to the beginning, the better). This “match closest” behaviour may be turned off by setting parameter *strict* to True. In such case, only conformers containing all *wanted* genres will be kept.

Parameters

- **wanted** – List of data genres used as completeness reference. If not given, evaluates to [primary_genres](#).

- **strict** – Indicates if all *wanted* genres must be present in the kept conformers (**strict=True**) or if “match closest” mechanism should be used as a fallback (**strict=False**, this is the default).

Notes

Conformers previously marked as “not kept” will not be affected.

trim_imaginary_frequencies() → None

Mark all conformers with imaginary frequencies as “not kept”.

Notes

Conformers previously marked as “not kept” will not be affected. Conformers that doesn’t contain “freq” genre will be treated as not having imaginary frequencies.

trim_non_matching_stoichiometry(*wanted: Optional[str] = None*) → None

Mark all conformers with stoichiometry other than *wanted* as “not kept”. If not given, *wanted* evaluates to the most common stoichiometry.

Parameters wanted – Only conformers with same stoichiometry will be kept. Evaluates to the most common stoichiometry if not given.

Notes

Conformers previously marked as “not kept” will not be affected. Conformers that doesn’t contain stoichiometry data are always treated as non-matching.

trim_not_optimized() → None

Mark all conformers that failed structure optimization as “not kept”.

Notes

Conformers previously marked as “not kept” will not be affected. Conformers that doesn’t contain optimization data are always treated as optimized.

trim_non_normal_termination() → None

Mark all conformers, which calculation job did not terminate normally, as “not kept”.

Notes

Conformers previously marked as “not kept” will not be affected. Conformers that doesn’t contain data regarding their calculation job’s termination are always treated as terminated abnormally.

trim_inconsistent_sizes() → None

Mark as “not kept” all conformers that contain any iterable data genre, that is of different length, than in case of majority of conformers.

Examples

```
>>> c = Conformers(
...     one={'a': [1, 2, 3]},
...     two={'a': [1, 2, 3]},
...     three={'a': [1, 2, 3, 4]}
... )
>>> c.kept
[True, True, True]
>>> c.trim_inconsistent_sizes()
>>> c.kept
[True, True, False]
```

Notes

Conformers previously marked as “not kept” will not be affected.

trim_to_range(*genre: str, minimum: Union[int, float] = -inf, maximum: Union[int, float] = inf, attribute: str = 'values'*) → None

Marks as “not kept” all conformers, which numeric value of data of specified genre is outside of the range specified by *minimum* and *maximum* values.

Parameters

- **genre** – Name of genre that should be compared to specified minimum and maximum values.
- **minimum** – Minimal accepted value - every conformer, which genre value evaluates to less than *minimum* will be marked as “not kept”. Defaults to `float(-inf)`.
- **maximum** – Maximal accepted value - every conformer, which genre value evaluates to more than *maximum* will be marked as “not kept”. Defaults to `float(inf)`.
- **attribute** – Attribute of DataArray of specified *genre* that contains one-dimensional array of numeric values. defaults to “*values*”.

Raises

- **AttributeError** – If DataArray associated with *genre* genre has no attribute *attribute*.
- **ValueError** – If data retrieved from specified genre’s attribute is not in the form of one-dimensional array.
- **TypeError** – If comparison cannot be made between elements of specified genre’s attribute and *minimum* or *maximum* values.

Notes

Conformers previously marked as “not kept” will not be affected.

trim_rmsd(*threshold: typing.Union[int, float], window_size: typing.Optional[typing.Union[int, float]], geometry_genre: str = 'last_read_geom', energy_genre: str = 'scf', ignore_hydrogen: bool = True, moving_window_strategy: typing.Callable = <function stretching_windows>*) → None

Marks as “not kept” all conformers that are identical with some other conformer, judging by a provided RMSD threshold.

To minimize computation cost, conformers are compared inside windows, that is a subsets of the original list of conformers. Those windows are generated by the `moving_window_strategy` function. The recommended strategy, and a default value, is `stretching_windows()`, but other are also available: `fixed_windows()` and `pyramid_windows()`. This function will be called with list of energies for conformers compared and (if it is not None) `window_size` parameter.

With default `moving_window_strategy` conformers, which energy difference (dE) is higher than given `window_size` are always treated as different, while those with dE smaller than `window_size` and RMSD value smaller than given `threshold` are considered identical. From two identical conformers, the one with lower energy is “kept”, and the other is discarded (marked as “not kept”).

Notes

RMSD threshold and size of the energy window should be chosen depending on the parameters of conformers’ set: number of conformers, size of the conformer, its lability, etc. However, `threshold` of 0.5 angstrom and `window_size` of 5 to 10 kcal/mol is a good place to start if in doubt.

Parameters

- **threshold** (*int or float*) – Maximum RMSD value to consider conformers identical.
- **window_size** (*int or float*) – Size of the energy window, in kcal/mol, inside which RMSD matrix is calculated. Essentially, a difference in conformers’ energy, after which conformers are always considered different.
- **geometry_genre** (*str*) – Genre of geometry used to calculate RMSD matrix. “last_read_geom” is default.
- **energy_genre** (*str*) – Genre of energy used to sort and group conformers into windows of given energy size. “scf” is used by default.
- **ignore_hydrogen** (*bool*) – If hydrogen atom should be discarded before RMSD calculation. Defaults to True.
- **moving_window_strategy** (*callable*) – Function that generates windows, inside which RMSD comparisons is performed.

Raises

- **InconsistentDataError** – If requested genres does not provide the same set of conformers.
- **ValueError** – When called with `ignore_hydrogen=True` but requested Geometry. atoms cannot be collapsed to 1-D array.

select_all() → None

Marks all conformers as ‘kept’. Equivalent to `conformers.kept = True`.

reject_all() → None

Marks all conformers as ‘not kept’. Equivalent to `conformers.kept = False`.

kept_keys(*indices: bool = False*) → `tesliper.glassware.conformers._KeptKeysView`

Equivalent of `dict.keys()` but gives view only on conformers marked as “kept”. Returned view may also provide information on conformers index in its Conformers instance if requested with `indices=True`.

```
>>> c = Conformers(c1={"g": 0.1}, c2={"g": 0.2}, c3={"g": 0.3})
>>> c.kept = [True, False, True]
>>> list(c.kept_keys())
["c1", "c3"]
```

(continues on next page)

(continued from previous page)

```
>>> list(c.kept_keys(indices=True))
[(0, "c1"), (2, "c3")]
```

Parameters *indices* (*bool*) – If resulting Conformers view should also provide index of each conformer. Defaults to False.

Returns View of kept conformers.

Return type `_KeptKeysView`

kept_values(*indices: bool = False*) → `tesliper.glassware.conformers._KeptValuesView`

Equivalent of `dict.values()` but gives view only on conformers marked as “kept”. Returned view may also provide information on conformers index in its Conformers instance if requested with `indices=True`.

```
>>> c = Conformers(c1={"g": 0.1}, c2={"g": 0.2}, c3={"g": 0.3})
>>> c.kept = [True, False, True]
>>> list(c.kept_values())
[{"g": 0.1}, {"g": 0.3}]
>>> list(c.kept_values(indices=True))
[(0, {"g": 0.1}), (2, {"g": 0.3})]
```

Parameters *indices* (*bool*) – If resulting Conformers view should also provide index of each conformer. Defaults to False.

Returns View of kept conformers.

Return type `_KeptValuesView`

fromkeys(*value=None*)

Create a new ordered dictionary with keys from iterable and values set to value.

get(*key, default=None, /*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

kept_items(*indices: bool = False*) → `tesliper.glassware.conformers._KeptItemsView`

Equivalent of `dict.items()` but gives view only on conformers marked as “kept”. Returned view may also provide information on conformers index in its Conformers instance if requested with `indices=True`.

```
>>> c = Conformers(c1={"g": 0.1}, c2={"g": 0.2}, c3={"g": 0.3})
>>> c.kept = [True, False, True]
>>> list(c.kept_items())
[("c1", {"g": 0.1}), ("c3", {"g": 0.3})]
>>> list(c.kept_items(indices=True))
[(0, "c1", {"g": 0.1}), (2, "c3", {"g": 0.3})]
```

Parameters *indices* (*bool*) – If resulting Conformers view should also provide index of each conformer. Defaults to False.

Returns View of kept conformers.

Return type `_KeptItemsView`

keys() → a set-like object providing a view on D's keys

pop(*k*, *d*) → *v*, remove specified key and return the corresponding value. If key is not found, *d* is returned if given, otherwise `KeyError` is raised.

setdefault(*key*, *default=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

values() → an object providing a view on D's values

property untrimmed: `tesliper.glassware.conformers.Conformers`

Temporally remove trimming. Implemented as context manager to use with python's 'with' keyword.

Examples

```
>>> c = Conformers(one={}, two={}, tree={})
>>> c.kept = [False, True, False]
>>> with c.untrimmed:
>>>     c.kept
[True, True, True]
>>> c.kept
[False, True, False]
```

trimmed_to(*blade*: *Union[Sequence[bool], Sequence[str], Sequence[int], bool]*) →

`tesliper.glassware.conformers.Conformers`

Temporally set trimming blade to given one. Implemented as context manager to use with python's 'with' keyword.

Parameters blade (*bool or sequence of bool, str, or int*) – Temporary trimming blade. To better understand how blade setting works, see `Conformers.kept` documentation.

Examples

```
>>> c = Conformers(one={}, two={}, tree={})
>>> c.kept = [True, True, False]
>>> with c.trimmed_to([1, 2]):
>>>     c.kept
[False, True, True]
>>> c.kept
[True, True, False]
```

property inconsistency_allowed: `tesliper.glassware.conformers.Conformers`

Temporally sets `Conformers`' 'allow_data_inconsistency' attribute to true. Implemented as context manager to use with python's 'with' keyword.

Examples

```
>>> c = Conformers(...)
>>> with c.inconsistency_allowed:
>>>     # do stuff here while c.allow_data_inconsistency is True
>>>     c.allow_data_inconsistency
True
>>> c.allow_data_inconsistency
False
```

tesliper.glassware.spectra

Objects representing spectra.

Classes

<i>SingleSpectrum</i> (genre, values, abscissa[, ...])	Represents a single spectrum: experimental, averaged from set of conformers, or calculated for only one conformer.
<i>Spectra</i> (genre, filenames, values, abscissa)	Represents a collection of spectra calculated for a number of conformers.

class tesliper.glassware.spectra.**SingleSpectrum**(genre: str, values: Sequence[float], abscissa: Sequence[float], width: float = 0.0, fitting: str = 'n/a', scaling: float = 1.0, offset: float = 0.0, filenames: Optional[Sequence[str]] = None, averaged_by: Optional[str] = None)

Represents a single spectrum: experimental, averaged from set of conformers, or calculated for only one conformer.

Notes

Calling `len()` on this class' instance will show a number of data points in the spectrum.

Parameters

- **genre** (str) – Name of data genre that this object represents.
- **values** (Sequence[float]) – List of intensity values for each point on the x-axis.
- **abscissa** (Sequence[float]) – List of x-axis values.
- **width** (float, optional) – Full width at half maximum used to calculate spectrum, if applies. Provided for the record only, by default 0.0.
- **fitting** (str, optional) – Name of the fitting function used to calculate spectrum, if applies. Provided for the record only, by default “n/a”.
- **scaling** (float, optional) – Multiplier for correction of signal intensity, by default 1.0.
- **offset** (float, optional) – Correction of the spectrum's shift. Positive value indicates a bathochromic shift, negative value indicates a hypsochromic shift. By default 0.0.

- **filenames** (*Optional[Sequence[str]], optional*) – List of identifiers of conformers that were used to calculate average spectrum, if applies.
- **averaged_by** (*Optional[str], optional*) – Energies genre used to calculate average spectrum, if applies.

property spectra_type

‘vibrational’, ‘electronic’, or ‘scattering’.

Type Returns type of spectra

property units: Dict[str, str]

Units in which spectral data is stored. It provides a unit for width, start, stop, step, *x*, and *y*. abscissa and values are stored in the same units as *x* and *y* respectively.

property scaling: Union[int, float]

A factor for correcting the scale of spectra. Setting it to new value changes the *y* attribute as well. It should be an int or float.

property offset: Union[int, float]

A factor for correcting the shift of spectra. Positive value indicates a bathochromic shift, negative value indicates a hypsochromic shift. Setting it to new value changes the *x* attribute as well. It should be an int or float.

property x: numpy.ndarray

Spectra’s x-values corrected by adding its *offset* to abscissa.

property y: numpy.ndarray

Spectra’s y-values corrected by multiplying its values by *scaling*.

scale_to(spectrum: tesliper.glassware.spectra.SingleSpectrum) → None

Establishes a scaling factor to best match a scale of the *spectrum* values.

Parameters *spectrum* (*SingleSpectrum*) – This spectrum’s y-axis values will be treated as a reference. If *spectrum* has its own scaling factor, it will be taken into account.

shift_to(spectrum: tesliper.glassware.spectra.SingleSpectrum) → None

Establishes an offset factor to best match given *spectrum*.

Parameters *spectrum* (*SingleSpectrum*) – This spectrum will be treated as a reference. If *spectrum* has its own offset factor, it will be taken into account.

class tesliper.glassware.spectra.Spectra(*genre: str, filenames: Sequence[str], values: Sequence[Sequence[float]], abscissa: Sequence[float], width: float = 0.0, fitting: str = 'n/a', scaling: float = 1.0, offset: float = 0.0, allow_data_inconsistency: bool = False*)

Represents a collection of spectra calculated for a number of conformers.

Changed in version 0.9.1: Corrected `len()` behavior.

Notes

Calling `len()` on this class' instance will show how many conformers' spectra it contains.

Parameters

- **genre** (*str*) – Name of data genre that this object represents.
- **filenames** (*Optional[Sequence[str]], optional*) – List of conformers' identifiers that were used to calculate spectra.
- **values** (*Sequence[float]*) – List of intensity values for each point on the x-axis.
- **abscissa** (*Sequence[float]*) – List of x-axis values.
- **width** (*float, optional*) – Full width at half maximum used to calculate spectra. Provided for the record only, by default 0.0.
- **fitting** (*str, optional*) – Name of the fitting function used to calculate spectra. Provided for the record only, by default "n/a".
- **scaling** (*float, optional*) – Multiplier for correction of signal intensity, by default 1.0.
- **offset** (*float, optional*) – Correction of the spectra's shift. Positive value indicates a bathochromic shift, negative value indicates a hypsochromic shift. By default 0.0.
- **allow_data_inconsistency** (*bool, optional*) – Flag signaling if instance should allow data inconsistency (see `ArrayProperty` for details).

property offset: `Union[int, float]`

A factor for correcting the shift of spectra. Positive value indicates a bathochromic shift, negative value indicates a hypsochromic shift. Setting it to new value changes the `x` attribute as well. It should be an `int` or `float`.

property scaling: `Union[int, float]`

A factor for correcting the scale of spectra. Setting it to new value changes the `y` attribute as well. It should be an `int` or `float`.

property spectra_type

'vibrational', 'electronic', or 'scattering'.

Type Returns type of spectra

property units: `Dict[str, str]`

Units in which spectral data is stored. It provides a unit for `width`, `start`, `stop`, `step`, `x`, and `y`. `abscissa` and `values` are stored in the same units as `x` and `y` respectively.

property x: `numpy.ndarray`

Spectra's x-values corrected by adding its `offset` to `abscissa`.

property y: `numpy.ndarray`

Spectra's y-values corrected by multiplying its values by `scaling`.

average(*energies: tesliper.glassware.arrays.Energies*) → *tesliper.glassware.spectra.SingleSpectrum*

A method for averaging spectra by population of conformers. If this object is empty, averaged spectrum will be a flat line at 0.0 intensity.

Parameters energies (*Energies*) – Object with populations and `genre` attributes containing respectively: list of populations values as `numpy.ndarray` and string specifying energy genre.

Returns Averaged spectrum.

Return type *SingleSpectrum*

scale_to(*spectrum*: *tesliper.glassware.spectra.SingleSpectrum*, *average_by*:
Optional[*tesliper.glassware.arrays.Energies*] = None) → None

Establishes a scaling factor to best match a scale of the *spectrum* values. An average spectrum is calculated prior to calculating the factor. If *average_by* is given, it is used to average by population of each conformer. Otherwise an arithmetic average of spectra is calculated, which may lead to inaccurate results.

Parameters

- **spectrum** (*SingleSpectrum*) – This spectrum’s y-axis values will be treated as a reference. If *spectrum* has its own scaling factor, it will be taken into account.
- **average_by** (*Energies*, optional) – Energies object, used to calculate average spectrum prior to calculating the factor. If not given, a simple arithmetic average of the spectra will be calculated.

shift_to(*spectrum*: *tesliper.glassware.spectra.SingleSpectrum*, *average_by*:
Optional[*tesliper.glassware.arrays.Energies*] = None) → None

Establishes an offset factor to best match given *spectrum*. An average spectrum is calculated prior to calculating the factor. If *average_by* is given, it is used to average by population of each conformer. Otherwise an arithmetic average of spectra is calculated, which may lead to inaccurate results.

Parameters

- **spectrum** (*SingleSpectrum*) – This spectrum will be treated as a reference. If *spectrum* has its own offset factor, it will be taken into account.
- **average_by** (*Energies*, optional) – Energies object, used to calculate average spectrum prior to calculating the factor. If not given, a simple arithmetic average of the spectra will be calculated.

3.8.5 tesliper.tesliper

Provides a facade-like interface for easy access to *tesliper*’s functionality.

There are some conventions that are important to note:

- *tesliper* stores multiple data entries of various types for each conformer. To prevent confusion with Python’s data type and with data itself, *tesliper* refers to specific kinds of data as “genres”. Genres in code are represented by specific strings, used as identifiers. To learn about data genres known to *tesliper*, see documentation for *GaussianParser*, which lists them.
- *tesliper* identifies conformers using stem of an extracted file (i.e. its filename without extension). When files with identical names are extracted in course of subsequent *Tesliper.extract()* calls or in recursive extraction using *tesliper_object.extract(recursive=True)*, they are treated as data for one conformer. This enables to join data from subsequent calculations steps, e.g. geometry optimization, vibrational spectra simulation, and electronic spectra simulation. Please note that if specific data genre is available from more than one calculation job, only recently extracted values will be stored.
- *tesliper* was designed to deal with multiple conformers of single molecule and may not work properly when used to process data concerning different molecules (i.e. having different number of atoms, different number of degrees of freedom, etc.). If you want to use it for such purpose anyway, you may set *Tesliper.conformers.allow_data_inconsistency* to True. *tesliper* will then stop complaining and try to do its best.

Classes

<code>Tesliper([input_dir, output_dir, ...])</code>	This class is a main access point to <code>tesliper</code> 's functionality.
---	--

class `tesliper.tesliper.Tesliper`(*input_dir: str = '.', output_dir: str = '.', wanted_files: Optional[Iterable[Union[str, pathlib.Path]]] = None, quantum_software: str = 'gaussian')*

This class is a main access point to `tesliper`'s functionality. It allows you to extract data from specified files, provides a proxy to the trimming functionality, gives access to data in form of specialized arrays, enables you to calculate and average desired spectra, and provides an easy way to export data.

Most basic use might look like this:

```
>>> tslr = Tesliper()
>>> tslr.extract()
>>> tslr.calculate_spectra()
>>> tslr.average_spectra()
>>> tslr.export_averaged()
```

This extracts data from files in the current working directory, calculates available spectra using standard parameters, averages them using available energy values, and exports to current working directory in .txt format.

You can customize this process by specifying call parameters for used methods and modifying `Tesliper`'s configuration attributes:

- to change source directory or location of exported files instantiate `Tesliper` object with `input_dir` and `output_dir` parameters specified, respectively. You can also set appropriate attributes on the instance directly.
- To extract only selected files in `input_dir` use `wanted_files` init parameter. It should be given an iterable of filenames you want to parse. Again, you can also directly set an identically named attribute.
- To change parameters used for calculation of spectra, modify appropriate entries of `parameters` attribute.
- Use other export methods to export more data and specify `fmt` parameter in method's call to export to other file formats.

```
>>> tslr = Tesliper(input_dir="./myjob/optimization/", output_dir="./myjob/output/")
>>> tslr.wanted_files = ["one", "two", "three"] # only files with this names
>>> tslr.extract() # use tslr.input_dir as source
>>> tslr.extract(path="./myjob/vcd_sim/") # use other input_dir
>>> tslr.conformers.trim_not_optimized() # trimming out unwanted conformers
>>> tslr.parameters["vcd"].update({"start": 500, "stop": 2500, "width": 2})
>>> tslr.calculate_spectra(genres=["vcd"]) # we want only VCD spectrum
>>> tslr.average_spectra()
>>> tslr.export_averaged(mode="w") # overwrite previously exported files
>>> tslr.export_activities(fmt="csv") # save activities for analysis elsewhere
>>> tslr.output_dir = "./myjob/ecd_sim/"
>>> tslr.export_job_file( # prepare files for next step of calculations
...     route="# td=(singlets,nstates=80) B3LYP/Def2TZVP"
... )
```

When modifying `Tesliper.parameters` be careful to not delete any of the parameters. If you need to revert to standard parameters values, you can find them in `Tesliper.standard_parameters`.

```
>>> tslr.parameters["ir"] = {
...     "start": 500, "stop": 2500, "width": 2
... } # this will cause problems!
>>> tslr.parameters = tslr.standard_parameters # revert to default values
```

Trimming functionality, used in previous example in `tslr.conformers.trim_not_optimized()`, allows you to filter out conformers that shouldn't be used in further processing and analysis. You can trim off conformers that were not optimized, contain imaginary frequencies, or have other unwanted qualities. Conformers with similar geometry may be discarded using an RMSD sieve. For more information about trimming, please refer to the documentation of [Conformers](#) class.

For more exploratory analysis, [Tesliper](#) provides an easy way to access desired data as an instance of specialized [DataArray](#) class. Those objects implement a number of convenience methods for dealing with specific data genres. A more detailed information on [DataArray](#) see [arrays](#) module documentation. To get data in this form use `array = tslr["genre"]` where "genre" is string with the name of desired data genre. For more control over instantiation of [DataArray](#) you may use [Tesliper.conformers.arrayed](#) factory method.

```
>>> energies = tslr["gib"]
>>> energies.values
array([-304.17061762, -304.17232455, -304.17186735])
>>> energies.populations
array([0.0921304 , 0.56174031, 0.3461293 ])
>>> energies.full_name
'Thermal Free Energy'
```

Please note, that if some conformers do not provide values for a specific data genre, it will be ignored when retrieving data for [DataArray](#) instantiation, regardless if it were trimmed off or not.

```
>>> tslr = Tesliper()
>>> tslr.conformers.update([
>>> ...     ('one', {'gib': -304.17061762}),
>>> ...     ('two', {'gib': -304.17232455}),
>>> ...     ('three', {'gib': -304.17186735}),
>>> ...     ('four', {}))
>>> ... ])
>>> tslr.conformers.kept
[True, True, True, True]
>>> energies = tslr["gib"]
>>> energies.filename
array(['one', 'two', 'three'], dtype='<U5')
```

conformers

Container for data extracted from Gaussian output files. It provides trimming functionality, enabling to filter out conformers of unwanted qualities.

Type [Conformers](#)

spectra

Spectra calculated so far, using [calculate_spectra\(\)](#) method. Possible keys are spectra genres: "ir", "vcd", "uv", "ecd", "raman", and "roa". Values are [Spectra](#) instances with lastly calculated spectra of this genre.

Type dict of str: Spectra

averaged

Spectra averaged using available energies genres, calculated with last call to `average_spectra()` method. Keys are tuples of two strings: averaged spectra genre and energies genre used for averaging.

Type dict of str: (dict of str: float or callable)

experimental

Experimental spectra loaded from disk. Possible keys are spectra genres: “ir”, “vcd”, “uv”, “ecd”, “raman”, and “roa”. Values are *Spectra* instances with experimental spectra of this genre.

Type dict of str: Spectra

quantum_software

A name, lower case, of the quantum chemical computations software used to obtain data. Used by tesliper to figure out, which parser to use to extract data, if custom parsers are available. Only “gaussian” is supported out-of-the-box.

Type str

parameters

Parameters for calculation of each spectra genres: “ir”, “vcd”, “uv”, “ecd”, “raman”, and “roa”. Available parameters are:

- “start”: float or int, the beginning of the spectral range,
- “stop”: float or int, the end of the spectral range,
- “step”: float or int, step of the abscissa,
- “width”: float or int, width of the peak,
- “fitting”: callable, function used to simulate peaks as curves, preferably one of *datawork.gaussian* or *datawork.lorentzian*.

“start”, “stop”, and “step” expect its values to be in cm^{-1} units for vibrational and scattering spectra, and nm units for electronic spectra. “width” expects its value to be in cm^{-1} units for vibrational and scattering spectra, and eV units for electronic spectra.

Type dict of str: (dict of str: float or callable)

Parameters

- **input_dir** (*str or path-like object, optional*) – Path to directory containing files for extraction, defaults to current working directory.
- **output_dir** (*str or path-like object, optional*) – Path to directory for output files, defaults to current working directory.
- **wanted_files** (*list of str or list of Path, optional*) – List of files or file-names representing wanted files. If not given, all files are considered wanted. File extensions are ignored.
- **quantum_software** (*str*) – A name of the quantum chemical computations software used to obtain data. Used by tesliper to figure out, which parser to use, if custom parsers are available.

clear()

Remove all data from the instance.

property temperature: float

Temperature of the system expressed in Kelvin units.

Value of this parameter is passed to *data arrays* created with the *Conformers.arrayed()* method, provided that the target data array class supports a parameter named *t* in it's constructor.

New in version 0.9.1.

Raises ValueError – if set to a value lower than zero.

Notes

It's actually just a proxy to *self.conformers.temperature*.

property energies: Dict[str, tesliper.glassware.arrays.Energies]

Data for each energies' genre as *Energies* data array. Returned dictionary is of form {"genre": *Energies*} for each of the genres: "scf", "zpe", "ten", "ent", and "gib". If no values are available for a specific genre, an empty *Energies* array is produced as corresponding dictionary value.

```
>>> tslr = Tesliper()
>>> tslr.energies
{
  "scf": Energies(genre="scf", ...),
  "zpe": Energies(genre="zpe", ...),
  "ten": Energies(genre="ten", ...),
  "ent": Energies(genre="ent", ...),
  "gib": Energies(genre="gib", ...),
}
```

Returns Dictionary with genre names as keys and *Energies* data arrays as values.

Return type dict

property activities: Dict[str, Union[tesliper.glassware.arrays.VibrationalActivities, tesliper.glassware.arrays.ScatteringActivities, tesliper.glassware.arrays.ElectronicActivities]]

Data for default activities used to calculate spectra as appropriate *SpectralActivities* subclass. Returned dictionary is of form {"genre": *SpectralActivities*} for each of the genres: "dip", "rot", "vosc", "vrot", "raman1", and "roa1". If no values are available for a specific genre, an empty data array is produced as corresponding dictionary value.

```
>>> tslr = Tesliper()
>>> tslr.activities
{
  "dip": VibrationalActivities(genre="dip", ...),
  "rot": VibrationalActivities(genre="rot", ...),
  "vosc": ElectronicActivities(genre="vosc", ...),
  "vrot": ElectronicActivities(genre="vrot", ...),
  "raman1": ScatteringActivities(genre="raman1", ...),
  "roa1": ScatteringActivities(genre="roa1", ...),
}
```

Returns Dictionary with genre names as keys and *SpectralActivities* data arrays as values.

Return type dict

property wanted_files: Optional[Set[str]]

Set of files that are desired for data extraction, stored as filenames without an extension. Any iterable of strings or Path objects is transformed to this form.

```
>>> tslr = Tesliper()
>>> tslr.wanted_files = [Path("./dir/file_one.out"), Path("./dir/file_two.out")]
>>> tslr.wanted_files
{"file_one", "file_two"}
```

May also be set to None or other “falsy” value, in such case it is ignored.

property standard_parameters: Dict[str, Dict[str, Union[int, float, Callable]]]

Default parameters for spectra calculation for each spectra genre (ir, vcd, uv, ecd, raman, roa). This returns a dictionary, but in fact it is a convenience, read-only attribute, modifying it will have no persisting effect.

update(*other: Optional[Dict[str, dict]] = None, **kwargs*)

Update stored conformers with given data.

Works like dict.update, but if key is already present, it updates dictionary associated with given key rather than assigning new value. Keys of dictionary passed as positional parameter (or additional keyword arguments given) should be conformers’ identifiers and its values should be dictionaries of {"genre": values} for those conformers.

Please note, that values of status genres like ‘optimization_completed’ and ‘normal_termination’ will be updated as well for such key, if are present in given new values.

```
>>> tslr.conformers
Conformers([('one', {'scf': -100, 'stoichiometry': 'CH4'})])
>>> tslr.update(
...     {'one': {'scf': 97}, 'two': {'scf': 82, 'stoichiometry': 'CH4'}}
... )
>>> tslr.conformers
Conformers([
    ('one', {'scf': 97, 'stoichiometry': 'CH4'}),
    ('two', {'scf': 82, 'stoichiometry': 'CH4'}),
])
```

property input_dir: pathlib.Path

Directory, from which files should be read.

property output_dir: pathlib.Path

Directory, to which generated files should be written.

extract_iterate(*path: Optional[Union[str, pathlib.Path]] = None, wanted_files: Optional[Iterable[str]] = None, extension: Optional[str] = None, recursive: bool = False*) → Generator[Tuple[str, dict], None, None]

Extracts data from chosen Gaussian output files present in given directory and yields data for each conformer found.

Uses *Tesliper.input_dir* as source directory and *Tesliper.wanted_files* list of chosen files if these are not explicitly given as ‘path’ and ‘wanted_files’ parameters.

Parameters

- **path** (*str or pathlib.Path, optional*) – Path to directory, from which Gaussian files should be read. If not given or is None, *Tesliper.output_dir* will be used.

- **wanted_files** (*list of str, optional*) – Filenames (without a file extension) of conformers that should be extracted. If not given or is `None`, `Tesliper.wanted_files` will be used. If `Tesliper.wanted_files` is also `None`, all found Gaussian output files will be parsed.
- **extension** (*str, optional*) – Only files with given extension will be parsed. If omitted, `Tesliper` will try to guess the extension from contents of input directory.
- **recursive** (*bool*) – If `True`, also subdirectories are searched for files to parse, otherwise subdirectories are ignored. Defaults to `False`.

Yields *tuple* – Two item tuple with name of parsed file as first and extracted data as second item, for each Gaussian output file parsed.

extract (*path: Optional[Union[str, pathlib.Path]] = None, wanted_files: Optional[Iterable[str]] = None, extension: Optional[str] = None, recursive: bool = False*)

Extracts data from chosen Gaussian output files present in given directory.

Uses `Tesliper.input_dir` as source directory and `Tesliper.wanted_files` list of chosen files if these are not explicitly given as *path* and *wanted_files* parameters.

Parameters

- **path** (*str or pathlib.Path, optional*) – Path to directory, from which Gaussian files should be read. If not given or is `None`, `Tesliper.output_dir` will be used.
- **wanted_files** (*list of str, optional*) – Filenames (without a file extension) of conformers that should be extracted. If not given or is `None`, `Tesliper.wanted_files` will be used.
- **extension** (*str, optional*) – Only files with given extension will be parsed. If omitted, `Tesliper` will try to guess the extension from contents of input directory.
- **recursive** (*bool*) – If `True`, also subdirectories are searched for files to parse, otherwise subdirectories are ignored. Defaults to `False`.

load_parameters (*path: Union[str, pathlib.Path], spectra_genre: Optional[str] = None*) → dict

Load calculation parameters from a file.

Parameters

- **path** (*str or pathlib.Path, optional*) – Path to the file with desired parameters specification.
- **spectra_genre** (*str, optional*) – Genre of spectra that loaded parameters concerns. If given, should be one of “ir”, “vcd”, “uv”, “ecd”, “raman”, or “roa” – parameters for that spectra will be updated with loaded values. Otherwise no update is done, only parsed data is returned.

Returns Parameters read from the file.

Return type dict

Notes

For information on supported format of parameters configuration file, please refer to [ParametersParser](#) documentation.

load_experimental(*path*: Union[str, pathlib.Path], *spectrum_genre*: str) → *tesliper.glassware.spectra.SingleSpectrum*

Load experimental spectrum from a file. Data read from file is stored as *SingleSpectrum* instance in *Tesliper.experimental* dictionary under *spectrum_genre* key.

Parameters

- **path** (str or *pathlib.Path*) – Path to the file with experimental spectrum.
- **spectrum_genre** (str) – Genre of the experimental spectrum that will be loaded. Should be one of “ir”, “vcd”, “uv”, “ecd”, “raman”, or “roa”.

Returns Experimental spectrum loaded from the file.

Return type *SingleSpectrum*

calculate_single_spectrum(*genre*: str, *conformer*: Union[str, int], *start*: Optional[Union[int, float]] = None, *stop*: Optional[Union[int, float]] = None, *step*: Optional[Union[int, float]] = None, *width*: Optional[Union[int, float]] = None, *fitting*: Optional[Callable[[numpy.ndarray, numpy.ndarray, numpy.ndarray, float], numpy.ndarray]] = None) → *tesliper.glassware.spectra.SingleSpectrum*

Calculates spectrum for requested conformer.

‘start’, ‘stop’, ‘step’, ‘width’, and ‘fitting’ parameters, if given, will be used instead of the parameters stored in *Tesliper.parameters* attribute. ‘start’, ‘stop’, and ‘step’ values will be interpreted as cm⁻¹ for vibrational or scattering spectra/activities and as nm for electronic ones. Similarly, ‘width’ will be interpreted as cm⁻¹ or eV. If not given, values stored in appropriate *Tesliper.parameters* are used.

Parameters

- **genre** (str) – Spectra genre (or related spectral activities genre) that should be calculated. If given spectral activity genre, this genre will be used to calculate spectra instead of the default activities.
- **conformer** (str or int) – Conformer, specified as it’s identifier or it’s index, for which spectrum should be calculated.
- **start** (int or float, optional) – Number representing start of spectral range.
- **stop** (int or float, optional) – Number representing end of spectral range.
- **step** (int or float, optional) – Number representing step of spectral range.
- **width** (int or float, optional) – Number representing half width of maximum peak height.
- **fitting** (function, optional) – Function, which takes spectral data, freqs, abscissa, width as parameters and returns numpy.array of calculated, non-corrected spectrum points. Basically one of *datawork.gaussian* or *datawork.lorentzian*.

Returns Calculated spectrum.

Return type *SingleSpectrum*

calculate_spectra(*genres*: Iterable[str] = ()) → Dict[str, *tesliper.glassware.spectra.Spectra*]

Calculates spectra for each requested genre using parameters stored in *Tesliper.parameters* attribute.

Parameters **genres** (*iterable of str*) – List of spectra genres (or related spectral activities genres) that should be calculated. If given spectral activity genre, this genre will be used to calculate spectra instead of the default activities. If given empty sequence (default), all available spectra will be calculated using default activities.

Returns **dict of str** – Dictionary with calculated spectra genres as keys and *Spectra* objects as values.

Return type *Spectra*

get_averaged_spectrum(*spectrum: str, energy: str, temperature: Optional[float] = None*) → *tesliper.glassware.spectra.SingleSpectrum*

Average previously calculated spectra using populations derived from specified energies.

New in version 0.9.1: The optional *temperature* parameter.

Changed in version 0.9.1: If spectra needed for averaging was not calculated so far, it will try to calculate it instead of raising a *KeyError*.

Parameters

- **spectrum** (*str*) – Genre of spectrum that should be averaged. This spectrum should be previously calculated using *calculate_spectrum()* method.
- **energy** (*str*) – Genre of energies, that should be used to calculate populations of conformers. These populations will be used as weights for averaging.
- **temperature** (*float, optional*) – Temperature used for calculation of the Boltzmann distribution for spectra averaging. If not given, *Tesliper.temperature()* value is used.

Returns Calculated averaged spectrum.

Return type *SingleSpectrum*

Raises **ValueError** – If no data for calculation of requested spectrum is available.

average_spectra() → Dict[Tuple[str, str], *tesliper.glassware.spectra.SingleSpectrum*]

For each previously calculated spectra (stored in *Tesliper.spectra* attribute) calculate it's average using population derived from each available energies genre.

Returns Averaged spectrum for each previously calculated spectra and energies known as a dictionary. It's keys are tuples of genres used for averaging and values are *SingleSpectrum* instances (so this dictionary is of form {tuple("spectra", "energies"): *SingleSpectrum*}).

Return type dict

export_data(*genres: Sequence[str], fmt: str = 'txt', mode: str = 'x'*)

Saves specified data genres to disk in given file format.

File formats available by default are: "txt", "csv", "xlsx", "gjf". Note that not all formats may be compatible with every genre (e.g. only genres associated with *Geometry* may be exported to .gjf format). In such case genres unsupported by given format are ignored.

Files produced are written to *Tesliper.output_dir* directory with filenames automatically generated using adequate genre's name and conformers' identifiers. In case of "xlsx" format only one file is produced and different data genres are written to separate sheets. If there are no values for given genre, no files will be created for this genre.

Parameters

- **genres** (*list of str*) – List of genre names, that will be saved to disk.
- **fmt** (*str*) – File format of output files, defaults to "txt".

- **mode** (*str*) – Specifies how writing to file should be handled. May be one of: “a” (append to existing file), “x” (only write if file doesn’t exist yet), “w” (overwrite file if it already exists). Defaults to “x”.

export_energies(*fmt: str = 'txt', mode: str = 'x'*)

Saves energies and population data to disk in given file format.

File formats available by default are: “txt”, “csv”, “xlsx”. Files produced are written to [Tesliper.output_dir](#) directory with filenames automatically generated using adequate genre’s name and conformers’ identifiers. In case of “xlsx” format only one file is produced and different data genres are written to separate sheets.

Parameters

- **fmt** (*str*) – File format of output files, defaults to “txt”.
- **mode** (*str*) – Specifies how writing to file should be handled. May be one of: “a” (append to existing file), “x” (only write if file doesn’t exist yet), “w” (overwrite file if it already exists). Defaults to “x”.

export_spectral_data(*fmt: str = 'txt', mode: str = 'x'*)

Saves unprocessed spectral data to disk in given file format.

File formats available by default are: “txt”, “csv”, “xlsx”. Files produced are written to [Tesliper.output_dir](#) directory with filenames automatically generated using adequate genre’s name and conformers’ identifiers. In case of “xlsx” format only one file is produced and different data genres are written to separate sheets.

Parameters

- **fmt** (*str*) – File format of output files, defaults to “txt”.
- **mode** (*str*) – Specifies how writing to file should be handled. May be one of: “a” (append to existing file), “x” (only write if file doesn’t exist yet), “w” (overwrite file if it already exists). Defaults to “x”.

export_activities(*fmt: str = 'txt', mode: str = 'x'*)

Saves unprocessed spectral activities to disk in given file format.

File formats available by default are: “txt”, “csv”, “xlsx”. Files produced are written to [Tesliper.output_dir](#) directory with filenames automatically generated using adequate genre’s name and conformers’ identifiers. In case of “xlsx” format only one file is produced and different data genres are written to separate sheets.

Parameters

- **fmt** (*str*) – File format of output files, defaults to “txt”.
- **mode** (*str*) – Specifies how writing to file should be handled. May be one of: “a” (append to existing file), “x” (only write if file doesn’t exist yet), “w” (overwrite file if it already exists). Defaults to “x”.

export_spectra(*fmt: str = 'txt', mode: str = 'x'*)

Saves spectra calculated previously to disk in given file format.

File formats available by default are: “txt”, “csv”, “xlsx”. Files produced are written to [Tesliper.output_dir](#) directory with filenames automatically generated using adequate genre’s name and conformers’ identifiers. In case of “xlsx” format only one file is produced and different data genres are written to separate sheets.

Parameters

- **fmt** (*str*) – File format of output files, defaults to “txt”.

- **mode** (*str*) – Specifies how writing to file should be handled. May be one of: “a” (append to existing file), “x” (only write if file doesn’t exist yet), “w” (overwrite file if it already exists). Defaults to “x”.

export_averaged(*fmt: str = 'txt', mode: str = 'x'*)

Saves spectra calculated and averaged previously to disk in given file format.

File formats available by default are: “txt”, “csv”, “xlsx”. Files produced are written to *Tesliper.output_dir* directory with filenames automatically generated using adequate genre’s name and conformers’ identifiers. In case of “xlsx” format only one file is produced and different data genres are written to separate sheets.

Parameters

- **fmt** (*str*) – File format of output files, defaults to “txt”.
- **mode** (*str*) – Specifies how writing to file should be handled. May be one of: “a” (append to existing file), “x” (only write if file doesn’t exist yet), “w” (overwrite file if it already exists). Defaults to “x”.

export_job_file(*fmt: str = 'gjf', mode: str = 'x', geometry_genre: str = 'last_read_geom', **kwargs*)

Saves conformers to disk as job files for quantum chemistry software in given file format.

Currently only “gjf” format is provided, used by Gaussian software. Files produced are written to *Tesliper.output_dir* directory with filenames automatically generated using conformers’ identifiers.

Parameters

- **fmt** (*str*) – File format of output files, defaults to “gjf”.
- **mode** (*str*) – Specifies how writing to file should be handled. May be one of: “a” (append to existing file), “x” (only write if file doesn’t exist yet), “w” (overwrite file if it already exists). Defaults to “x”.
- **geometry_genre** (*str*) – Name of the data genre representing conformers’ geometry that should be used as input geometry. Please note that the default value “last_read_geom” is not necessarily an optimized geometry. Use “optimized_geom” if this is what you need.
- **kwargs** – Any additional keyword parameters are passed to the writer object, relevant to the *fmt* requested. Keyword supported by the default “gjf”-format *writer* are as follows:
 - route** A calculations route: keywords specifying calculations directives for quantum chemical calculations software.
 - link0** Dictionary with “link zero” commands, where each key is command’s name and each value is this command’s parameter.
 - comment** Contents of title section, i.e. a comment about the calculations.
 - post_spec** Anything that should be placed after conformer’s geometry specification. Will be written to the file as given.

serialize(*filename: str = '.tslr', mode: str = 'x'*) → None

Serialize instance of *Tesliper* object to a file in *output_dir*.

Parameters

- **filename** (*str*) – Name of the file, to which content will be written. Defaults to “.tslr”.
- **mode** (*str*) – Specifies how writing to file should be handled. Should be one of characters: “x” or “w”. “x” - only write if file doesn’t exist yet; “w” - overwrite file if it already exists. Defaults to “x”.

Raises **ValueError** – If given any other mode than “x” or “w”.

Notes

If `output_dir` is None, current working directory is assumed.

classmethod `load(source: Union[pathlib.Path, str]) → tesliper.tesliper.Tesliper`

Load serialized *Tesliper* object from given file.

Parameters `source` (*pathlib.Path* or *str*) – Path to the file with serialized *Tesliper* object.

Returns New instance of *Tesliper* class containing data read from the file.

Return type *Tesliper*

3.8.6 tesliper.writing

Objects for data serialization.

Aside from concrete implementations of *WriterBase*-derived classes for particular file formats, this module provides a `writer()` factory function that allows to dynamically retrieve particular writer objects. This function is used by *Tesliper* when exporting data to allow for use of user-provided *WriterBase* subclasses.

Modules

<code>tesliper.writing.csv_writer</code>	Data export to CSV format.
<code>tesliper.writing.gjf_writer</code>	Export of Gaussian input files (.gjf) for setting up new calculation step.
<code>tesliper.writing.serializer</code>	Serialization and deserialization of <i>Tesliper</i> objects.
<code>tesliper.writing.txt_writer</code>	Data export to text files.
<code>tesliper.writing.writer_base</code>	Interface for writing data to disk.
<code>tesliper.writing.xlsx_writer</code>	Data export to excel files.

tesliper.writing.csv_writer

Data export to CSV format.

Classes

<code>CsvWriter(destination[, mode, ...])</code>	Writes extracted or calculated data to .csv format files.
--	---

class `tesliper.writing.csv_writer.CsvWriter(destination: Union[str, pathlib.Path], mode: str = 'x', include_header: bool = True, dialect: Union[str, csv.Dialect] = 'excel', **fmtparams)`

Writes extracted or calculated data to .csv format files.

Parameters

- **destination** (*str* or *pathlib.Path*) – Directory, to which generated files should be written.

- **mode** (*str*) – Specifies how writing to file should be handled. Should be one of characters: ‘a’ (append to existing file), ‘x’ (only write if file doesn’t exist yet), or ‘w’ (overwrite file if it already exists).
- **include_header** (*bool*, *optional*) – Determines if file should contain a header with column names, True by default.
- **dialect** (*str* or *csv.Dialect*) – Name of a dialect or *csv.Dialect* object, which will be used by underlying *csv.writer*.
- **fmtparams** (*dict*, *optional*) – Additional formatting parameters for underlying *csv.writer* to use. For list of valid parameters consult *csv.Dialect* documentation.

_get_handle(*template: Union[str, string.Template]*, *template_params: dict*, *open_params: Optional[dict] = None*) → *Iterator[IO]*

Helper method for creating files. Given additional kwargs will be passed to *Path.open()* method. Implemented as context manager for use with *with* statement.

Parameters

- **template** (*str* or *string.Template*) – Template that will be used to generate filenames.
- **template_params** (*dict*) – Dictionary of {identifier: value} for *.make_name* method.
- **open_params** (*dict*, *optional*) – Arguments for *Path.open()* used to open file.

Yields

- *IO* – file handle, will be closed automatically after *with* statement exits
- *meta public:*

_iter_handles(*filenames: Iterable[str]*, *template: Union[str, string.Template]*, *template_params: dict*, *open_params: Optional[dict] = None*) → *Iterator[IO]*

Helper method for iteration over generated files. Given additional kwargs will be passed to *Path.open()* method.

Parameters

- **filenames** (*list of str*) – list of source filenames, used as value for *\${conf}* placeholder in *name_template*
- **template_params** (*dict*) – Dictionary of {identifier: value} for *.make_name* method.
- **open_params** (*dict*, *optional*) – arguments for *Path.open()* used to open file.

Yields

- *TextIO* – file handle, will be closed automatically on next iteration
- *meta public:*

generic(*data: List[Union[tesliper.glassware.arrays.DataArray, tesliper.glassware.arrays.IntegerArray, tesliper.glassware.arrays.FloatArray, tesliper.glassware.arrays.BooleanArray, tesliper.glassware.arrays.InfoArray]]*, *name_template: Union[str, string.Template] = '\${cat}.\${det}.\${ext}'*)

Writes generic data from multiple *DataArray*-like objects to a single file. Said objects should provide a single value for each conformer.

Parameters

- **data** – *DataArray* objects that are to be exported.
- **name_template** – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

energies(*energies*: *tesliper.glassware.arrays.Energies*, *corrections*: *Optional[tesliper.glassware.arrays.FloatArray]* = *None*, *name_template*: *Union[str, string.Template]* = *'distribution-\${genre}.\${ext}'*)

Writes Energies object to csv file. The output also contains derived values: populations, min_factors, deltas. Corrections are added only when explicitly given.

Parameters

- **energies** (*glassware.Energies*) – Energies objects that is to be serialized
- **corrections** (*glassware.DataArray*, *optional*) – *DataArray* objects containing energies corrections
- **name_template** (*str* or *string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

single_spectrum(*spectrum*: *tesliper.glassware.spectra.SingleSpectrum*, *name_template*: *Union[str, string.Template]* = *'\${cat}.\${genre}-\${det}.\${ext}'*)

Writes SingleSpectrum object to csv file.

Parameters

- **spectrum** (*glassware.SingleSpectrum*) – spectrum, that is to be serialized
- **name_template** (*str* or *string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

spectral_activities(*band*: *tesliper.glassware.arrays.SpectralActivities*, *data*: *List[tesliper.glassware.arrays.SpectralActivities]*, *name_template*: *Union[str, string.Template]* = *'\${conf}.\${cat}-\${det}.\${ext}'*)

Writes SpectralActivities objects to csv files (one file for each conformer).

Parameters

- **band** (*glassware.SpectralActivities*) – Object containing information about band at which transitions occur; it should be frequencies for vibrational data and wave-lengths or excitation energies for electronic data.
- **data** (*list of glassware.SpectralActivities*) – SpectralActivities objects that are to be serialized; all should contain information for the same set of conformers and correspond to given band. Assumes that all *data*'s elements have the same *spectra_type*, which is passed to the *name_template* as “det”.
- **name_template** (*str* or *string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

Raises ValueError – if *data* is an empty sequence

property destination: *pathlib.Path*

Directory, to which generated files should be written.

Raises FileNotFoundError – If given destination doesn't exist or is not a directory.

Type *pathlib.Path*

property dialect

Name of a dialect (as string) or csv.Dialect object, which will be used by csv.writer.

static distribute_data(data: List) → Tuple[Dict[str, List], Dict[str, Any]]

Sorts given data by genre category for use by specialized writing methods.

Returns

- **distr** (dict) – Dictionary with *DataArray*-like objects, sorted by their type. Each {key: value} pair is {name of the type in lowercase format: list of *DataArray* objects of this type}.
- **extras** (dict) – Spacial-case genres: extra information used by some writer methods when exporting data. Available {key: value} pairs (if given in *data*) are:

corrections: dict of {"energy genre": *FloatArray*},
 frequencies: *Bands*,
 wavelengths: *Bands*,
 excitation: *Bands*,
 stoichiometry: *InfoArray*,
 charge: *IntegerArray*,
 multiplicity: *IntegerArray*

property fmtparams

Dict of additional formatting parameters for csv.writer to use. For list of valid parameters consult csv.Dialect documentation.

Raises TypeError – if invalid parameter is given

geometry(geometry: tesliper.glassware.arrays.Geometry, charge:

Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None, multiplicity:
 Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None,
 name_template: Union[str, string.Template] = "")

Interface for writing single object with geometry of each conformer. Evoked when handling *Geometry* objects.

Parameters

- **geometry** – Positions of atoms in each conformer. Mandatory in custom implementation.
- **charge** – Value of each structure's charge. Mandatory in custom implementation.
- **multiplicity** – Value of each structure's multiplicity. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises NotImplementedError – Whenever called, this is an interface that should not be used directly.

make_name(template: Union[str, string.Template], conf: str = "", num: Union[str, int] = "", genre: str = "", cat: str = "", det: str = "", ext: str = "") → str

Create filename using given template and given or global values for known identifiers. The identifier should

be used in the template as "\${identifier}" where “identifier” is the name of identifier. Available names and their meaning are:

`${ext}` - appropriate file extension
`${conf}` - name of the conformer
`${num}` - number of the file according to internal counter
`${genre}` - genre of exported data
`${cat}` - category of produced output
`${det}` - category-specific detail

The `${ext}` identifier is filled with the value of Writers `extension` attribute if not explicitly given as parameter to this method’s call. Values for other identifiers should be provided by the caller.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames. It should contain only known identifiers, listed above.
- **conf** (*str*) – value for `${conf}` identifier, defaults to empty string.
- **num** (*str or int*) – value for `${str}` identifier, defaults to empty string.
- **genre** (*str*) – value for `${genre}` identifier, defaults to empty string.
- **cat** (*str*) – value for `${cat}` identifier, defaults to empty string.
- **det** (*str*) – value for `${det}` identifier, defaults to empty string.
- **ext** (*str*) – value for `${ext}` identifier, defaults to empty string.

Raises **ValueError** – If given template or string contains any unexpected identifiers.

Examples

Must be first subclassed and instantiated:

```
>>> class MyWriter(WriterBase):
>>>     extension = "foo"
>>> wrt = MyWriter("/path/to/some/directory/")
```

```
>>> wrt.make_name(template="somefile.${ext}")
"somefile.foo"
>>> wrt.make_name(template="${conf}.${ext}")
".foo" # conf is empty string by default
>>> wrt.make_name(template="${conf}.${ext}", conf="conformer")
"conformer.foo"
>>> wrt.make_name(template="Unknown_identifier_${bla}.${ext}")
Traceback (most recent call last):
ValueError: Unexpected identifiers given: bla.
```

property mode

Specifies how writing to file should be handled. Should be one of characters: “a”, “x”, or “w”. “a” - append to existing file; “x” - only write if file doesn’t exist yet; “w” - overwrite file if it already exists.

Raises **ValueError** – If given anything other than “a”, “x”, or “w”.

overview(*energies: Sequence[tesliper.glassware.arrays.Energies], frequencies: Optional[tesliper.glassware.arrays.Bands] = None, stoichiometry: Optional[tesliper.glassware.arrays.InfoArray] = None, name_template: Union[str, string.Template] = ""*)

Interface for generating an overview of known conformers: values of energies, number of imaginary frequencies, and stoichiometry for each conformer. Evoked when handling [Energies](#) objects.

Parameters

- **energies** – List of objects representing different energies genres for each conformer. Mandatory in custom implementation.
- **frequencies** – [Bands](#) of “freq” genre, with list of frequencies for each conformer. Mandatory in custom implementation. May be `None` when method evoked by handler.
- **stoichiometry** – Stoichiometry of each conformer. Mandatory in custom implementation. May be `None` when method evoked by handler.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

spectral_data(*band: tesliper.glassware.arrays.SpectralData, data: List[tesliper.glassware.arrays.SpectralData], name_template: Union[str, string.Template] = '\${conf}.\${cat}-\${det}.\${ext}'*)

Writes `SpectralData` objects to csv files (one file for each conformer).

Parameters

- **band** (*glassware.SpectralData*) – Object containing information about band at which transitions occur; it should be frequencies for vibrational data and wavelengths or excitation energies for electronic data.
- **data** (*list of glassware.SpectralData*) – `SpectralData` objects that are to be serialized; all should contain information for the same set of conformers and correspond to given band. Assumes that all *data*’s elements have the same *spectra_type*, which is passed to the *name_template* as “det”.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

Raises `ValueError` – if *data* is an empty sequence

write(*data: List*) → `None`

Writes [DataArray](#)-like objects to disk, decides how to write them based on the type of each object. If some types of given objects are not supported by this writer, data of this type is ignored and a warning is emitted.

Parameters *data* (*List*) – [DataArray](#)-like objects that should be written to disk.

spectra(*spectra: tesliper.glassware.spectra.Spectra, name_template: Union[str, string.Template] = '\${conf}.\${genre}.\${ext}'*)

Writes `Spectra` object to .csv files (one file for each conformer).

Parameters

- **spectra** (*glassware.Spectra*) – `Spectra` object, that is to be serialized.

- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

transitions(*transitions: tesliper.glassware.arrays.Transitions, wavelengths: tesliper.glassware.arrays.Bands, only_highest=True, name_template: Union[str, string.Template] = '\${conf}.\${cat}-\${det}.\${ext}'*)

Writes electronic transitions data to CSV files (one for each conformer).

Parameters

- **transitions** (*glassware.Transitions*) – Electronic transitions data that should be serialized.
- **wavelengths** (*glassware.ElectronicActivities*) – Object containing information about wavelength at which transitions occur.
- **only_highest** (*bool*) – Specifies if only transition of highest contribution to given band should be reported. If `False` all transition are saved to file. Defaults to `True`.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

tesliper.writing.gjf_writer

Export of Gaussian input files (.gjf) for setting up new calculation step.

Classes

<code>GjfWriter(destination[, mode, link0, route, ...])</code>	Generates Gaussian input files for each conformer given.
--	--

```
class tesliper.writing.gjf_writer.GjfWriter(destination: Union[str, pathlib.Path], mode: str = 'x',
                                             link0: Optional[Dict[str, Union[str, bool]]] = None, route:
                                             str = "", comment: str = 'No information provided.',
                                             post_spec: str = "")
```

Generates Gaussian input files for each conformer given.

Parameters

- **destination** (*str or pathlib.Path*) – Directory, to which generated files should be written.
- **mode** (*str, optional*) – Specifies how writing to file should be handled. Should be one of characters: “a” (append to existing file); “x” (only write if file doesn’t exist yet); or “w” (overwrite file if it already exists). Defaults to “x”.
- **link0** (*Dict[str, Union[str, bool]], optional*) – Link0 commands that should be included in generated files, as a dictionary of {“command”: “value”}. Refer to [link0](#) for more information. If omitted, no link0 commands are added.
- **route** (*str*) – Calculation directives for Gaussan, refer to the Gaussian documentation for information on how to construct the calculations route.
- **comment** (*str, optional*) – Additional text, describing the calculations, by default “No information provided.”

- **post_spec**(*str*, *optional*) – Additional specification written after the molecule specification, written to generated files as provided by the user (you need to take care of line breaks). If omitted, no additional specification is added.

geometry(*geometry*: `tesliper.glassware.arrays.Geometry`, *charge*: `Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None`, *multiplicity*: `Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None`, *name_template*: `Union[str, string.Template] = '${conf}.${ext}'`)

Write given conformers' geometries to multiple Gaussian input files.

Parameters

- **geometry** (`Geometry`) – `Geometry` object containing data for each conformer that should be exported as Gaussian input file.
- **charge** (`Union[IntegerArray, Sequence[int], int, None]`, *optional*) – Molecule's charge for each conformer. May be a sequence of values or one value that will be repeated for each conformer. By default 0 for each.
- **multiplicity** (`Union[IntegerArray, Sequence[int], int, None]`, *optional*) – Molecule's multiplicity for each conformer. May be a sequence of values or one value that will be repeated for each conformer. By default 1 for each.
- **name_template** (`Union[str, Template]`, *optional*) – Template that will be used to generate filenames, by default “\${conf}.\${ext}”. Refer to `make_name()` documentation for details on supported placeholders.

_get_handle(*template*: `Union[str, string.Template]`, *template_params*: `dict`, *open_params*: `Optional[dict] = None`) → `Iterator[IO]`

Helper method for creating files. Given additional kwargs will be passed to `Path.open()` method. Implemented as context manager for use with `with` statement.

Parameters

- **template** (*str* or `string.Template`) – Template that will be used to generate filenames.
- **template_params** (`dict`) – Dictionary of {identifier: value} for `.make_name` method.
- **open_params** (`dict`, *optional*) – Arguments for `Path.open()` used to open file.

Yields

- *IO* – file handle, will be closed automatically after `with` statement exits
- *meta public*:

_iter_handles(*filenames*: `Iterable[str]`, *template*: `Union[str, string.Template]`, *template_params*: `dict`, *open_params*: `Optional[dict] = None`) → `Iterator[IO]`

Helper method for iteration over generated files. Given additional kwargs will be passed to `Path.open()` method.

Parameters

- **filenames** (*list of str*) – list of source filenames, used as value for `${conf}` placeholder in `name_template`
- **template_params** (`dict`) – Dictionary of {identifier: value} for `.make_name` method.
- **open_params** (`dict`, *optional*) – arguments for `Path.open()` used to open file.

Yields

- *TextIO* – file handle, will be closed automatically on next iteration
- *meta public*:

property destination: `pathlib.Path`

Directory, to which generated files should be written.

Raises `FileNotFoundError` – If given destination doesn't exist or is not a directory.

Type `pathlib.Path`

static `distribute_data(data: List) → Tuple[Dict[str, List], Dict[str, Any]]`

Sorts given data by genre category for use by specialized writing methods.

Returns

- **distr** (*dict*) – Dictionary with *DataArray*-like objects, sorted by their type. Each {key: value} pair is {name of the type in lowercase format: list of *DataArray* objects of this type}.
- **extras** (*dict*) – Spacial-case genres: extra information used by some writer methods when exporting data. Available {key: value} pairs (if given in *data*) are:

corrections: dict of {"energy genre": *FloatArray*},
frequencies: *Bands*,
wavelengths: *Bands*,
excitation: *Bands*,
stoichiometry: *InfoArray*,
charge: *IntegerArray*,
multiplicity: *IntegerArray*

energies(*energies*: `tesliper.glassware.arrays.Energies`, *corrections*:
`Optional[tesliper.glassware.arrays.FloatArray] = None`, *name_template*: `Union[str,`
`string.Template] = ""`)

Interface for writing energies values, and optionally their corrections. Evoked when handling *Energies* objects.

Parameters

- **energies** – Conformers' energies. Mandatory in custom implementation.
- **corrections** – Correction of energies values. Mandatory in custom implementation. May be None when method evoked by handler.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

generic(*data*: `List[Union[tesliper.glassware.arrays.DataArray, tesliper.glassware.arrays.IntegerArray,`
`tesliper.glassware.arrays.FloatArray, tesliper.glassware.arrays.BooleanArray,`
`tesliper.glassware.arrays.InfoArray]]`, *name_template*: `Union[str, string.Template] = ""`)

Interface for writing generic data: any that provides one value for each conformer. Evoked when handling *DataArray*, *IntegerArray*, *FloatArray*, *BooleanArray*, or *InfoArray*.

Parameters **data** – List of objects that provide one value for each conformer.

Raises **NotImplementedError** – Whenever called, this is an interface that should not be used directly.

make_name(*template: Union[str, string.Template], conf: str = "", num: Union[str, int] = "", genre: str = "", cat: str = "", det: str = "", ext: str = ""*) → str

Create filename using given template and given or global values for known identifiers. The identifier should be used in the template as "\${identifier}" where “identifier” is the name of identifier. Available names and their meaning are:

\${ext} - appropriate file extension
 \${conf} - name of the conformer
 \${num} - number of the file according to internal counter
 \${genre} - genre of exported data
 \${cat} - category of produced output
 \${det} - category-specific detail

The \${ext} identifier is filled with the value of Writers *extension* attribute if not explicitly given as parameter to this method’s call. Values for other identifiers should be provided by the caller.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames. It should contain only known identifiers, listed above.
- **conf** (*str*) – value for \${conf} identifier, defaults to empty string.
- **num** (*str or int*) – value for \${str} identifier, defaults to empty string.
- **genre** (*str*) – value for \${genre} identifier, defaults to empty string.
- **cat** (*str*) – value for \${cat} identifier, defaults to empty string.
- **det** (*str*) – value for \${det} identifier, defaults to empty string.
- **ext** (*str*) – value for \${ext} identifier, defaults to empty string.

Raises **ValueError** – If given template or string contains any unexpected identifiers.

Examples

Must be first subclassed and instantiated:

```
>>> class MyWriter(WriterBase):
>>>     extension = "foo"
>>> wrt = MyWriter("/path/to/some/directory/")
```

```
>>> wrt.make_name(template="somefile.${ext}")
"somefile.foo"
>>> wrt.make_name(template="${conf}.${ext}")
".foo" # conf is empty string by default
>>> wrt.make_name(template="${conf}.${ext}", conf="conformer")
"conformer.foo"
>>> wrt.make_name(template="Unknown_identifier_${bla}.${ext}")
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
ValueError: Unexpected identifiers given: bla.
```

property mode

Specifies how writing to file should be handled. Should be one of characters: “a”, “x”, or “w”. “a” - append to existing file; “x” - only write if file doesn’t exist yet; “w” - overwrite file if it already exists.

Raises `ValueError` – If given anything other than “a”, “x”, or “w”.

overview(*energies: Sequence[tesliper.glassware.arrays.Energies], frequencies:*

Optional[tesliper.glassware.arrays.Bands] = None, stoichiometry:

Optional[tesliper.glassware.arrays.InfoArray] = None, name_template: Union[str, string.Template] = "")

Interface for generating an overview of known conformers: values of energies, number of imaginary frequencies, and stoichiometry for each conformer. Evoked when handling *Energies* objects.

Parameters

- **energies** – List of objects representing different energies genres for each conformer. Mandatory in custom implementation.
- **frequencies** – *Bands* of “freq” genre, with list of frequencies for each conformer. Mandatory in custom implementation. May be `None` when method evoked by handler.
- **stoichiometry** – Stoichiometry of each conformer. Mandatory in custom implementation. May be `None` when method evoked by handler.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

single_spectrum(*spectrum: tesliper.glassware.spectra.SingleSpectrum, name_template: Union[str, string.Template] = "")*

Interface for writing a single spectrum to disk: calculated for one conformer or averaged. Evoked when handling *SingleSpectrum* objects.

Parameters

- **spectrum** – Single calculated spectrum. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

spectra(*spectra: tesliper.glassware.spectra.Spectra, name_template: Union[str, string.Template] = "")*

Interface for writing a set of spectra of one type calculated for many conformers. Evoked when handling *Spectra* objects.

Parameters

- **spectra** – Spectra of one type calculated for multiple conformers. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

spectral_activities(*band*: `tesliper.glassware.arrays.Bands`, *data*: `List[tesliper.glassware.arrays.SpectralActivities]`, *name_template*: `Union[str, string.Template]` = "")

Interface for writing multiple objects with spectral activities (data that may be converted to signal intensity). Evoked when handling one of the: `VibrationalActivities`, `ElectronicActivities`, `ScatteringActivities` objects.

Parameters

- **band** – Band at which transitions occur for each conformer. Mandatory in custom implementation.
- **data** – List of objects representing different spectral activities genres. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

spectral_data(*band*: `tesliper.glassware.arrays.Bands`, *data*: `List[tesliper.glassware.arrays.SpectralData]`, *name_template*: `Union[str, string.Template]` = "")

Interface for writing multiple objects with spectral data that is not a spectral activity (cannot be converted to signal intensity). Evoked when handling one of the: `VibrationalData`, `ElectronicData`, `ScatteringData` objects.

Parameters

- **band** – Band at which transitions occur for each conformer. Mandatory in custom implementation.
- **data** – List of objects representing different spectral data genres (but not spectral activities). Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

transitions(*transitions*: `tesliper.glassware.arrays.Transitions`, *wavelengths*: `tesliper.glassware.arrays.Bands`, *only_highest*: `bool` = `True`, *name_template*: `Union[str, string.Template]` = "")

Interface for writing single object with electronic transitions data. Evoked when handling `Transitions` objects.

Parameters

- **transitions** – List of objects representing different spectral data genres (but not spectral_activities). Mandatory in custom implementation.
- **wavelengths** – Wavelengths at which transitions occur for each conformer. Mandatory in custom implementation.
- **only_highest** – Boolean flag indicating if all transitions should be written to disk or only these transition that contributes the most for each wavelength/ May be omitted in custom implementation.

- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

write(*data: List*) → None

Writes *DataArray*-like objects to disk, decides how to write them based on the type of each object. If some types of given objects are not supported by this writer, data of this type is ignored and a warning is emitted.

Parameters *data* (*List*) – *DataArray*-like objects that should be written to disk.

property link0: Dict[str, Union[str, bool]]

Link0 commands, in a form of {"command": "value"}, that will be placed in the beginning of each Gaussian input file created. If any *command* is an unknown keyword, an exception will be raised. Accepted *command* keywords are as follows:

Mem str specifying required memory

Chk str with file path

OldChk str with file path

SChk str with file path

RWF str with file path

OldMatrix str with file path

OldRawMatrix str with file path

Int str with spec

D2E str with spec

KJob str with link number and, optionally, space-separated number

Save boolean

ErrorSave boolean

NoSave boolean, same as ErrorSave

Subst str with link number and space-separated file path

Commands that provide a file path as a value may be parametrized for each conformer. You can put a placeholder inside a given string path, that will be parametrized when writing to file. See [make_name\(\)](#) to see available placeholders. You may use any of values listed there, however `${conf}` and `${num}` will probably be the most useful.

property route: str

Also known as *#lines*, specifies desired calculation type, model chemistry, and other options for Gaussian. If pound sign is missing, it is added in the beginning. For supported keywords and syntax refer to the Gaussian's documentation.

tesliper.writing.serializer

Serialization and deserialization of *Tesliper* objects.

Classes

<code>ArchiveLoader</code> (source[, encoding])	Class for deserialization of <i>Tesliper</i> objects.
<code>ArchiveWriter</code> (destination[, mode, encoding])	Class for serialization of <i>Tesliper</i> objects.

class tesliper.writing.serializer.**ArchiveWriter**(*destination: Union[str, pathlib.Path], mode: str = 'x', encoding: str = 'utf-8'*)

Class for serialization of *Tesliper* objects.

Structure of the produced archive:

```

.
├── arguments: {input_dir:str, output_dir:str, wanted_files=[str]}
├── parameters: {"ir": {params}, ..., "roa": {params}}
├── conformers
│   ├── arguments: {"allow_data_inconsistency": bool,
│   │               "temperature_of_the_system": float}
│   ├── filenames: [str]
│   ├── kept: [bool]
│   └── data
│       ├── filename_1: {genre:str: data}
│       ├── ...
│       └── filename_N: {genre:str: data}
├── spectra
│   ├── experimental
│   │   ├── spectra_genre_1: {attr_name: SingleSpectrum.attr}
│   │   ├── ...
│   │   └── spectra_genre_N: {attr_name: SingleSpectrum.attr}
│   ├── calculated
│   │   ├── spectra_genre_1: {attr_name: Spectra.attr}
│   │   ├── ...
│   │   └── spectra_genre_N: {attr_name: Spectra.attr}
│   └── averaged
│       ├── spectra_genre_1-energies-genre-1: {attr_name: SingleSpectrum.attr}
│       ├── ...
│       └── spectra_genre_N-energies-genre-N: {attr_name: SingleSpectrum.attr}

```

Parameters

- **destination** (*Union[str, Path]*) – Path to target file.
- **mode** (*str, optional*) – Specifies how writing to file should be handled. Should be one of characters: ‘a’ (append to existing file), ‘x’ (only write if file doesn’t exist yet), or ‘w’ (overwrite file if it already exists). Defaults to “x”.
- **encoding** (*str, optional*) – Encoding of the output, by default “utf-8”

property mode

Specifies how writing to file should be handled. Should be one of characters: “a”, “x”, or “w”. “a” - append to existing file; “x” - only write if file doesn’t exist yet; “w” - overwrite file if it already exists.

Raises ValueError – If given anything other than “a”, “x”, or “w”.

property destination: pathlib.Path

Directory, to which generated files should be written.

Raises FileNotFoundError – If given destination doesn’t exist or is not a directory.

Type pathlib.Path

jsonencode(*obj: Any, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw*) → bytes
json.dumps wrapper, that encodes JSON produced.

class tesliper.writing.serializer.ArchiveLoader(*source: Union[str, pathlib.Path], encoding: str = 'utf-8'*)

Class for deserialization of Tesliper objects.

Parameters

- **source** (*Union[str, Path]*) – Path to the source file.
- **encoding** (*str, optional*) – Source file encoding, by default “utf-8”.

property source: pathlib.Path

File, from which data should read.

Notes

If str given, it will be converted to pathlib.Path.

Raises FileNotFoundError – If given destination doesn’t exist.

Type pathlib.Path

jsondecode(*string: bytes, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw*) → Any
json.loads wrapper, that decodes bytes before parsing as JSON.

tesliper.writing.txt_writer

Data export to text files.

Classes

<i>TxtWriter</i> (<i>destination[, mode]</i>)	Writes extracted or calculated data to .txt format files.
---	---

class tesliper.writing.txt_writer.TxtWriter(*destination: Union[str, pathlib.Path], mode: str = 'x'*)

Writes extracted or calculated data to .txt format files.

Parameters

- **destination** (*str or pathlib.Path*) – Directory, to which generated files should be written.
- **mode** (*str*) – Specifies how writing to file should be handled. Should be one of characters: ‘a’ (append to existing file), ‘x’ (only write if file doesn’t exist yet), or ‘w’ (overwrite file if it already exists).

generic(*data*: List[Union[tesliper.glassware.arrays.DataArray, tesliper.glassware.arrays.IntegerArray, tesliper.glassware.arrays.FloatArray, tesliper.glassware.arrays.BooleanArray, tesliper.glassware.arrays.InfoArray]], *name_template*: Union[str, string.Template] = '{cat}.{det}.{ext}')

Writes generic data from multiple *DataArray*-like objects to a single file. Said objects should provide a single value for each conformer.

Parameters

- **data** – *DataArray* objects that are to be exported.
- **name_template** – Template that will be used to generate filenames. Refer to *make_name()* documentation for details on supported placeholders.

overview(*energies*: Sequence[tesliper.glassware.arrays.Energies], *frequencies*: Optional[tesliper.glassware.arrays.Bands] = None, *stoichiometry*: Optional[tesliper.glassware.arrays.InfoArray] = None, *name_template*: Union[str, string.Template] = '{cat}.{det}.{ext}')

Writes essential information from multiple Energies objects to single txt file.

Notes

All Energy objects given should contain information for the same set of files.

Parameters

- **energies** (*list of glassware.Energies*) – Energies objects that is to be exported
- **frequencies** (*glassware.DataArray, optional*) – DataArray object containing frequencies, needed for imaginary frequencies count
- **stoichiometry** (*glassware.InfoArray, optional*) – InfoArray object containing stoichiometry information
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to *make_name()* documentation for details on supported placeholders.

energies(*energies*: tesliper.glassware.arrays.Energies, *corrections*: Optional[tesliper.glassware.arrays.FloatArray] = None, *name_template*: Union[str, string.Template] = 'distribution-\${genre}.{det}.{ext}')

Writes Energies object to txt file.

Parameters

- **energies** (*glassware.Energies*) – Energies object that is to be serialized
- **corrections** (*glassware.DataArray, optional*) – DataArray object, containing energies corrections
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to *make_name()* documentation for details on supported placeholders.

single_spectrum(*spectrum*: tesliper.glassware.spectra.SingleSpectrum, *name_template*: Union[str, string.Template] = '{cat}.{genre}-{det}.{ext}')

Writes SingleSpectrum object to txt file.

Parameters

- **spectrum** (*glassware.SingleSpectrum*) – spectrum, that is to be serialized
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

spectral_activities(*band: tesliper.glassware.arrays.SpectralActivities, data: List[tesliper.glassware.arrays.SpectralActivities], name_template: Union[str, string.Template] = '\${conf}.\${cat}-\${det}.\${ext}'*)

Writes SpectralActivities objects to txt files (one for each conformer).

Parameters

- **band** (*glassware.SpectralActivities*) – object containing information about band at which transitions occur; it should be frequencies for vibrational data and wavelengths or excitation energies for electronic data
- **data** (*list of glassware.SpectralActivities*) – SpectralActivities objects that are to be serialized; all should contain information for the same conformers. Assumes that all *data*'s elements have the same *spectra_type*, which is passed to the *name_template* as “det”.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

Raises ValueError – if *data* is an empty sequence

spectral_data(*band: tesliper.glassware.arrays.SpectralActivities, data: List[tesliper.glassware.arrays.SpectralData], name_template: Union[str, string.Template] = '\${conf}.\${cat}-\${det}.\${ext}'*)

Writes SpectralData objects to txt files (one for each conformer).

Parameters

- **band** (*glassware.SpectralData*) – object containing information about band at which transitions occur; it should be frequencies for vibrational data and wavelengths or excitation energies for electronic data
- **data** (*list of glassware.SpectralData*) – SpectralData objects that are to be serialized; all should contain information for the same conformers. Assumes that all *data*'s elements have the same *spectra_type*, which is passed to the *name_template* as “det”.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

Raises ValueError – if *data* is an empty sequence

spectra(*spectra: tesliper.glassware.spectra.Spectra, name_template: Union[str, string.Template] = '\${conf}.\${genre}.\${ext}'*)

Writes Spectra object to text files (one for each conformer).

Parameters

- **spectra** (*glassware.Spectra*) – Spectra object, that is to be serialized
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.


```
transitions(transitions: tesliper.glassware.arrays.Transitions, wavelengths: tesliper.glassware.arrays.Bands,
             only_highest=True, name_template: Union[str, string.Template] =
             '${conf}.${cat}-${det}.${ext}')
```

Writes electronic transitions data to text files (one for each conformer).

Parameters

- **transitions** (*glassware.Transitions*) – Electronic transitions data that should be serialized.
- **wavelengths** (*glassware.ElectronicActivities*) – Object containing information about wavelength at which transitions occur.
- **only_highest** (*bool*) – Specifies if only transition of highest contribution to given band should be reported. If `False` all transition are saved to file. Defaults to `True`.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

```
_get_handle(template: Union[str, string.Template], template_params: dict, open_params: Optional[dict] =
             None) → Iterator[IO]
```

Helper method for creating files. Given additional kwargs will be passed to `Path.open()` method. Implemented as context manager for use with `with` statement.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames.
- **template_params** (*dict*) – Dictionary of {identifier: value} for `.make_name` method.
- **open_params** (*dict, optional*) – Arguments for `Path.open()` used to open file.

Yields

- *IO* – file handle, will be closed automatically after `with` statement exits
- *meta public*:

```
_iter_handles(filenames: Iterable[str], template: Union[str, string.Template], template_params: dict,
               open_params: Optional[dict] = None) → Iterator[IO]
```

Helper method for iteration over generated files. Given additional kwargs will be passed to `Path.open()` method.

Parameters

- **filenames** (*list of str*) – list of source filenames, used as value for `${conf}` placeholder in `name_template`
- **template_params** (*dict*) – Dictionary of {identifier: value} for `.make_name` method.
- **open_params** (*dict, optional*) – arguments for `Path.open()` used to open file.

Yields

- *TextIO* – file handle, will be closed automatically on next iteration
- *meta public*:

property destination: `pathlib.Path`

Directory, to which generated files should be written.

Raises `FileNotFoundError` – If given destination doesn't exist or is not a directory.

Type `pathlib.Path`

static `distribute_data`(*data*: `List`) → `Tuple[Dict[str, List], Dict[str, Any]]`

Sorts given data by genre category for use by specialized writing methods.

Returns

- **distr** (*dict*) – Dictionary with `DataArray`-like objects, sorted by their type. Each {key: value} pair is {name of the type in lowercase format: list of `DataArray` objects of this type}.
- **extras** (*dict*) – Spacial-case genres: extra information used by some writer methods when exporting data. Available {key: value} pairs (if given in *data*) are:

corrections: dict of {"energy genre": `FloatArray`},

frequencies: `Bands`,

wavelengths: `Bands`,

excitation: `Bands`,

stoichiometry: `InfoArray`,

charge: `IntegerArray`,

multiplicity: `IntegerArray`

geometry(*geometry*: `tesliper.glassware.arrays.Geometry`, *charge*:

`Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None`, *multiplicity*:

`Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None`,

name_template: `Union[str, string.Template] = ""`)

Interface for writing single object with geometry of each conformer. Evoked when handling `Geometry` objects.

Parameters

- **geometry** – Positions of atoms in each conformer. Mandatory in custom implementation.
- **charge** – Value of each structure's charge. Mandatory in custom implementation.
- **multiplicity** – Value of each structure's multiplicity. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

make_name(*template*: `Union[str, string.Template]`, *conf*: `str = "", num: Union[str, int] = "", genre: str = "", cat: str = "", det: str = "", ext: str = ""`) → `str`

Create filename using given template and given or global values for known identifiers. The identifier should be used in the template as "`{{identifier}}`" where "identifier" is the name of identifier. Available names and their meaning are:

`${ext}` - appropriate file extension
`${conf}` - name of the conformer
`${num}` - number of the file according to internal counter
`${genre}` - genre of exported data
`${cat}` - category of produced output
`${det}` - category-specific detail

The `${ext}` identifier is filled with the value of Writers `extension` attribute if not explicitly given as parameter to this method's call. Values for other identifiers should be provided by the caller.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames. It should contain only known identifiers, listed above.
- **conf** (*str*) – value for `${conf}` identifier, defaults to empty string.
- **num** (*str or int*) – value for `${str}` identifier, defaults to empty string.
- **genre** (*str*) – value for `${genre}` identifier, defaults to empty string.
- **cat** (*str*) – value for `${cat}` identifier, defaults to empty string.
- **det** (*str*) – value for `${det}` identifier, defaults to empty string.
- **ext** (*str*) – value for `${ext}` identifier, defaults to empty string.

Raises ValueError – If given template or string contains any unexpected identifiers.

Examples

Must be first subclassed and instantiated:

```
>>> class MyWriter(WriterBase):
>>>     extension = "foo"
>>> wrt = MyWriter("/path/to/some/directory/")
```

```
>>> wrt.make_name(template="somefile.${ext}")
"somefile.foo"
>>> wrt.make_name(template="${conf}.${ext}")
".foo" # conf is empty string by default
>>> wrt.make_name(template="${conf}.${ext}", conf="conformer")
"conformer.foo"
>>> wrt.make_name(template="Unknown_identifier_${bla}.${ext}")
Traceback (most recent call last):
ValueError: Unexpected identifiers given: bla.
```

property mode

Specifies how writing to file should be handled. Should be one of characters: "a", "x", or "w". "a" - append to existing file; "x" - only write if file doesn't exist yet; "w" - overwrite file if it already exists.

Raises ValueError – If given anything other than "a", "x", or "w".

write(*data: List*) → None

Writes `DataArray`-like objects to disk, decides how to write them based on the type of each object. If some types of given objects are not supported by this writer, data of this type is ignored and a warning is emitted.

Parameters `data` (*List*) – *DataArray*-like objects that should be written to disk.

tesliper.writing.writer_base

Interface for writing data to disk.

This module contains `writer()` factory function that enables to dynamically create a writer object that's responsible for saving data in a desired output format. `writer()` instantiates a subclass of `WriterBase`, an Abstract Base Class also defined here. `WriterBase` provides an interface for all serial data writers (objects that export conformers' data to multiple files) used by `tesliper`.

`WriterBase` expects its subclasses to provide an *extension* class attribute, which is used as an extension of files produced by this particular writer, and also as an identifier for the output format, used by the `writer()` factory function. `tesliper` is shipped with four such writers: `TxtWriter` for writing to .txt files, `CsvWriter` for writing in CSV format, `XlsxWriter` for creating Excel files, and `GjfWriter` for preparing Gaussian input files.

You may want to export your data to other file formats - in such case you will need to implement your own writer. To do this, subclass `WriterBase`, provide its *extension* as mentioned above, and implement writing methods for data you intend to support in your writer. The table below lists these methods, along with a brief description and *DataArray*-like object, for which the method will be called by writer's `write()` method.

Table 33: Methods used by default to write certain data

Writer's Method	Description	Associated array
<code>generic()</code>	Generic data: any genre that provides one value for each conformer.	<i>DataArray</i> , <i>IntegerArray</i> , <i>FloatArray</i> , <i>BooleanArray</i> , <i>InfoArray</i> .
<code>overview()</code>	General information about conformers: energies, imaginary frequencies, stoichiometry.	<i>Energies</i>
<code>energies()</code>	Detailed information about conformers' relative energy, including calculated populations	<i>Energies</i>
<code>single_spectrum()</code>	A spectrum - calculated for single conformer or averaged.	<i>SingleSpectrum</i>
<code>spectral_data()</code>	Data related to spectral activity, but not convertible to spectra.	<i>SpectralData</i>
<code>spectral_activities()</code>	Data that may be used to simulate conformers' spectra.	<i>SpectralActivities</i>
<code>spectra()</code>	Spectra for multiple conformers.	<i>Spectra</i>
<code>transitions()</code>	Electronic transitions from ground to excited state, contributing to each band.	<i>Transitions</i>
<code>geometry()</code>	Geometry (positions of atoms in space) of conformers.	<i>Geometry</i>

Note: These methods are not abstract methods, but will still raise a `NotImplementedError` if called. This is to let you omit implementation of methods you don't need or wouldn't make sense for the particular format and still provide an abstract interface. `tesliper` takes advantage of this in its implementation of `GjfWriter`, which only implements `geometry()` method, because export of, e.g. a calculated spectrum as a Gaussian input would be pointless.

Writer object decides which of these methods to call based on the type of each *DataArray*-like object passed to the `write()` method. For some of them, it also passes additional *DataArray*-like objects, referred to as *extras*, e.g. corresponding *Bands* for spectral data. See documentation for particular method to learn, which of its parameters are mandatory, which are optional, and which should expect `None` as a possible value of *extra*.

When implementing one of these methods in your writer, you should take care of opening and closing file files, formatting data you export, and writing to the file. For the first part you may use one of the helper methods that provide a ready-to-use file handles: `_iter_handles()` for writing to many files in batch or `_get_handle()` for writing to one file only. Both require a template that will be used to generate filename for produced files. To learn more about how these templates are handled by tesliper, see `make_name()` documentation.

As mentioned before, writer object uses type of the `DataArray`-like object (or, more precisely, a name of its class) to decide which method to use for writing to disk. If you introduce a new subclass of `DataArray` for handling some genres, you will need to tell the Writer class, how it should handle these new objects. This is done by implementing a custom handler method. It's name should begin with an underscore, followed by the name of your subclass in lower case, followed by `"_handler"`. Also, it should take two parameters: `data` and `extras`. First one is a list of instances of your subclass, second one is a dictionary of special-case genres, both retrieved from arguments given to `write()` method (for details on which genres as treated as special cases, see `distribute_data()`). Handler is responsible for calling appropriate writing method with arguments it needs.

Here is an example: let's assume you have implemented a custom `DataArray` subclass for "ldip" and "lrot" genres with some additional functionality, but you'd like tesliper to treat it as the original `ElectronicActivities` class for purposes of writing to disk.

```
class LengthActivities(ElectronicActivities):
    associated_genres = ("ldip", "lrot")
    ... # custom functionality implemented here

class UpdatedTxtWriter(TxtWriter):
    extension = "txt"

    def _lengthactivities_handler(self, data, extras):
        # written like ``ElectronicActivities``, so just delegate to its handler
        self._electronicactivities_handler(data, extras)
```

If you'd like to treat this new subclass differently, then you should provide a custom writing method for this kind of data:

```
class UpdatedTxtWriter(TxtWriter):
    extension = "txt"

    def length_activities(
        self,
        band: Bands,
        data: List[LengthActivities],
        name_template: Union[str, Template] = "${conf}.${cat}-${det}.${ext}",
    ):
        # we will use ``_iter_handles`` method for opening/closing files
        template_params = {"genre": band.genre, "cat": "activity", "det": "length"}
        handles = self._iter_handles(band.fileNames, name_template, template_params)
        # we will iterate conformer by conformer
        values = zip(*[arr.values for arr in data])
        for values, handle in zip(values, handles):
            ... # writting logic

    def _lengthactivities_handler(self, data, extras):
        self.length_activities(band=extras["wavelengths"], data=data)
```

In both cases `UpdatedTxtWriter` will be picked by the `writer()` instead of the original `TxtWriter`, thanks to the automatic registration done by the base class `WriterBase`.

Warning: If `extension = "txt"` line would be omitted in the `UpdatedTxtWriter` definition, it would be picked by the `writer()` for “txt” format anyway, because `extension`’s value would be inherited from `TxtWriter`. If you want to prevent this, you can provide a falsy value for the `extension` class attribute, i.e. an empty string or `None`. If your custom writer should still use the same extension as one of the default writers, provide `extension` also as an instance-level attribute:

```
class UpdatedTxtWriter(TxtWriter):
    extension = "" # do not register

    def __init__(self, destination, mode):
        super().__init__(destination, mode)
        self.extension = "txt" # use in generated filenames
```

Functions

<code>writer(fmt, destination[, mode])</code>	Factory function that returns concrete implementation of <code>WriterBase</code> subclass, most recently defined for export to <i>fmt</i> file format.
---	--

Classes

<code>WriterBase(destination[, mode])</code>	Base class for writers that handle export process based on genre of exported data.
--	--

`tesliper.writing.writer_base.writer(fmt: str, destination: Union[str, pathlib.Path], mode: str = 'x', **kwargs) → tesliper.writing.writer_base.WriterBase`

Factory function that returns concrete implementation of `WriterBase` subclass, most recently defined for export to *fmt* file format.

Parameters

- **fmt** (*str*) – File format, to which export will be done.
- **destination** (*Union[str, Path]*) – Path to file or directory, to which export will be done.
- **mode** (*str*) – Specifies how writing to file should be handled. Should be one of characters: “a” (append to existing file), “x” (only write if file doesn’t exist yet), or “w” (overwrite file if it already exists). Defaults to “x”.
- **kwargs** – Any additional keyword arguments will be passed as-is to the constructor of the retrieved `WriterBase` subclass.

Returns Initialized `WriterBase` subclass most recently defined for export to *fmt* file format.

Return type `WriterBase`

Raises `ValueError` – If `WriterBase` subclass for export to *fmt* file format was not defined.

class `tesliper.writing.writer_base.WriterBase(destination: Union[str, pathlib.Path], mode: str = 'x')`

Base class for writers that handle export process based on genre of exported data.

Subclasses should provide an `extension` class-level attribute and writing methods that subclass intend to support (see below). Value of `extension` will be used to register subclass as a default writer for export to files that

this value indicates (“txt”, “csv”, *etc.*). Not providing value for this attribute results in a `TypeError` exception. If subclass should not be registered, use an empty string as the attribute’s value.

`WriterBase` provides a `write()` method for writing arbitrary `DataArray`-like objects to disk. It dispatches those objects to appropriate writing methods, based on their type. Those writing methods are:

```
generic(),
overview(),
energies(),
single_spectrum(),
spectral_data(),
spectral_activities(),
spectra(),
transitions(),
geometry().
```

To learn more about implementing custom writers, see their documentation and `writer_base` documentation or extend section.

Parameters

- **destination** (*str* or *pathlib.Path*) – Directory, to which generated files should be written.
- **mode** (*str*) – Specifies how writing to file should be handled. Should be one of characters: ‘a’ (append to existing file), ‘x’ (only write if file doesn’t exist yet), or ‘w’ (overwrite file if it already exists).

`energies_order = ['zpe', 'ten', 'ent', 'gib', 'scf']`

Default order, in which energy-related data is written to files.

abstract property extension

Identifier of this writer, indicating the format of files generated, and a default extension of those files used by the `make_name()` method. A falsy value, i.e. an empty string or `None` prevents this writer from being registered and used by `writer()` factory function.

Returns Default extension of files generated by this writer and it’s identifier.

Return type `str`

property mode

Specifies how writing to file should be handled. Should be one of characters: “a”, “x”, or “w”. “a” - append to existing file; “x” - only write if file doesn’t exist yet; “w” - overwrite file if it already exists.

Raises **ValueError** – If given anything other than “a”, “x”, or “w”.

property destination: `pathlib.Path`

Directory, to which generated files should be written.

Raises **FileNotFoundError** – If given destination doesn’t exist or is not a directory.

Type `pathlib.Path`

static `distribute_data(data: List) → Tuple[Dict[str, List], Dict[str, Any]]`

Sorts given data by genre category for use by specialized writing methods.

Returns

- **distr** (*dict*) – Dictionary with `DataArray`-like objects, sorted by their type. Each {key: value} pair is {name of the type in lowercase format: list of `DataArray` objects of this type}.

- **extras** (*dict*) – Spacial-case genres: extra information used by some writer methods when exporting data. Available {key: value} pairs (if given in *data*) are:

corrections: dict of {"energy genre": *FloatArray*},
frequencies: *Bands*,
wavelengths: *Bands*,
excitation: *Bands*,
stoichiometry: *InfoArray*,
charge: *IntegerArray*,
multiplicity: *IntegerArray*

make_name(*template: Union[str, string.Template], conf: str = "", num: Union[str, int] = "", genre: str = "", cat: str = "", det: str = "", ext: str = ""*) → *str*

Create filename using given template and given or global values for known identifiers. The identifier should be used in the template as "\${identifier}" where "identifier" is the name of identifier. Available names and their meaning are:

\${ext} - appropriate file extension
\${conf} - name of the conformer
\${num} - number of the file according to internal counter
\${genre} - genre of exported data
\${cat} - category of produced output
\${det} - category-specific detail

The *\${ext}* identifier is filled with the value of Writers *extension* attribute if not explicitly given as parameter to this method's call. Values for other identifiers should be provided by the caller.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames. It should contain only known identifiers, listed above.
- **conf** (*str*) – value for *\${conf}* identifier, defaults to empty string.
- **num** (*str or int*) – value for *\${str}* identifier, defaults to empty string.
- **genre** (*str*) – value for *\${genre}* identifier, defaults to empty string.
- **cat** (*str*) – value for *\${cat}* identifier, defaults to empty string.
- **det** (*str*) – value for *\${det}* identifier, defaults to empty string.
- **ext** (*str*) – value for *\${ext}* identifier, defaults to empty string.

Raises ValueError – If given template or string contains any unexpected identifiers.

Examples

Must be first subclassed and instantiated:

```
>>> class MyWriter(WriterBase):
>>>     extension = "foo"
>>> wrt = MyWriter("/path/to/some/directory/")

>>> wrt.make_name(template="somefile.${ext}")
"somefile.foo"
>>> wrt.make_name(template="${conf}.${ext}")
".foo" # conf is empty string by default
>>> wrt.make_name(template="${conf}.${ext}", conf="conformer")
"conformer.foo"
>>> wrt.make_name(template="Unknown_identifier_${bla}.${ext}")
Traceback (most recent call last):
ValueError: Unexpected identifiers given: bla.
```

_get_handle(*template: Union[str, string.Template], template_params: dict, open_params: Optional[dict] = None*) → Iterator[IO]

Helper method for creating files. Given additional kwargs will be passed to `Path.open()` method. Implemented as context manager for use with `with` statement.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames.
- **template_params** (*dict*) – Dictionary of {identifier: value} for `.make_name` method.
- **open_params** (*dict, optional*) – Arguments for `Path.open()` used to open file.

Yields

- *IO* – file handle, will be closed automatically after `with` statement exits
- *meta public:*

_iter_handles(*filenames: Iterable[str], template: Union[str, string.Template], template_params: dict, open_params: Optional[dict] = None*) → Iterator[IO]

Helper method for iteration over generated files. Given additional kwargs will be passed to `Path.open()` method.

Parameters

- **filenames** (*list of str*) – list of source filenames, used as value for `${conf}` placeholder in `name_template`
- **template_params** (*dict*) – Dictionary of {identifier: value} for `.make_name` method.
- **open_params** (*dict, optional*) – arguments for `Path.open()` used to open file.

Yields

- *TextIO* – file handle, will be closed automatically on next iteration
- *meta public:*

write(data: List) → None

Writes [DataArray](#)-like objects to disk, decides how to write them based on the type of each object. If some types of given objects are not supported by this writer, data of this type is ignored and a warning is emitted.

Parameters data (List) – [DataArray](#)-like objects that should be written to disk.

generic(data: List[Union[tesliper.glassware.arrays.DataArray, tesliper.glassware.arrays.IntegerArray, tesliper.glassware.arrays.FloatArray, tesliper.glassware.arrays.BooleanArray, tesliper.glassware.arrays.InfoArray]], name_template: Union[str, string.Template] = "")

Interface for writing generic data: any that provides one value for each conformer. Evoked when handling [DataArray](#), [IntegerArray](#), [FloatArray](#), [BooleanArray](#), or [InfoArray](#).

Parameters data – List of objects that provide one value for each conformer.

Raises **NotImplementedError** – Whenever called, this is an interface that should not be used directly.

overview(energies: Sequence[tesliper.glassware.arrays.Energies], frequencies: Optional[tesliper.glassware.arrays.Bands] = None, stoichiometry: Optional[tesliper.glassware.arrays.InfoArray] = None, name_template: Union[str, string.Template] = "")

Interface for generating an overview of known conformers: values of energies, number of imaginary frequencies, and stoichiometry for each conformer. Evoked when handling [Energies](#) objects.

Parameters

- **energies** – List of objects representing different energies genres for each conformer. Mandatory in custom implementation.
- **frequencies** – [Bands](#) of “freq” genre, with list of frequencies for each conformer. Mandatory in custom implementation. May be None when method evoked by handler.
- **stoichiometry** – Stoichiometry of each conformer. Mandatory in custom implementation. May be None when method evoked by handler.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises **NotImplementedError** – Whenever called, this is an interface that should not be used directly.

energies(energies: tesliper.glassware.arrays.Energies, corrections: Optional[tesliper.glassware.arrays.FloatArray] = None, name_template: Union[str, string.Template] = "")

Interface for writing energies values, and optionally their corrections. Evoked when handling [Energies](#) objects.

Parameters

- **energies** – Conformers’ energies. Mandatory in custom implementation.
- **corrections** – Correction of energies values. Mandatory in custom implementation. May be None when method evoked by handler.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises **NotImplementedError** – Whenever called, this is an interface that should not be used directly.

single_spectrum(*spectrum*: [tesliper.glassware.spectra.SingleSpectrum](#), *name_template*: *Union[str, string.Template]* = "")

Interface for writing a single spectrum to disk: calculated for one conformer or averaged. Evoked when handling [SingleSpectrum](#) objects.

Parameters

- **spectrum** – Single calculated spectrum. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises [NotImplementedError](#) – Whenever called, this is an interface that should not be used directly.

spectral_data(*band*: [tesliper.glassware.arrays.Bands](#), *data*: *List[tesliper.glassware.arrays.SpectralData]*, *name_template*: *Union[str, string.Template]* = "")

Interface for writing multiple objects with spectral data that is not a spectral activity (cannot be converted to signal intensity). Evoked when handling one of the: [VibrationalData](#), [ElectronicData](#), [ScatteringData](#) objects.

Parameters

- **band** – Band at which transitions occur for each conformer. Mandatory in custom implementation.
- **data** – List of objects representing different spectral data genres (but not spectral activities). Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises [NotImplementedError](#) – Whenever called, this is an interface that should not be used directly.

spectral_activities(*band*: [tesliper.glassware.arrays.Bands](#), *data*: *List[tesliper.glassware.arrays.SpectralActivities]*, *name_template*: *Union[str, string.Template]* = "")

Interface for writing multiple objects with spectral activities (data that may be converted to signal intensity). Evoked when handling one of the: [VibrationalActivities](#), [ElectronicActivities](#), [ScatteringActivities](#) objects.

Parameters

- **band** – Band at which transitions occur for each conformer. Mandatory in custom implementation.
- **data** – List of objects representing different spectral activities genres. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises [NotImplementedError](#) – Whenever called, this is an interface that should not be used directly.

spectra(*spectra*: [tesliper.glassware.spectra.Spectra](#), *name_template*: *Union[str, string.Template]* = "")

Interface for writing a set of spectra of one type calculated for many conformers. Evoked when handling [Spectra](#) objects.

Parameters

- **spectra** – Spectra of one type calculated for multiple conformers. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

transitions(*transitions*: `tesliper.glassware.arrays.Transitions`, *wavelengths*: `tesliper.glassware.arrays.Bands`, *only_highest*: `bool = True`, *name_template*: `Union[str, string.Template] = ""`)

Interface for writing single object with electronic transitions data. Evoked when handling *Transitions* objects.

Parameters

- **transitions** – List of objects representing different spectral data genres (but not *spectral_activities*). Mandatory in custom implementation.
- **wavelengths** – Wavelengths at which transitions occur for each conformer. Mandatory in custom implementation.
- **only_highest** – Boolean flag indicating if all transitions should be written to disk or only these transition that contributes the most for each wavelength/ May be omitted in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

geometry(*geometry*: `tesliper.glassware.arrays.Geometry`, *charge*: `Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None`, *multiplicity*: `Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None`, *name_template*: `Union[str, string.Template] = ""`)

Interface for writing single object with geometry of each conformer. Evoked when handling *Geometry* objects.

Parameters

- **geometry** – Positions of atoms in each conformer. Mandatory in custom implementation.
- **charge** – Value of each structure's charge. Mandatory in custom implementation.
- **multiplicity** – Value of each structure's multiplicity. Mandatory in custom implementation.
- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

tesliper.writing.xlsx_writer

Data export to excel files.

Classes

<code>XlsxWriter(destination[, mode, filename])</code>	Writes extracted data to .xlsx file.
--	--------------------------------------

```
class tesliper.writing.xlsx_writer.XlsxWriter(destination: Union[str, pathlib.Path], mode: str = 'x',
                                             filename: str = 'tesliper-output.${ext}')
```

Writes extracted data to .xlsx file.

Parameters

- **destination** (*str or pathlib.Path*) – Directory, to which generated files should be written.
- **mode** (*str*) – Specifies how writing to file should be handled. Should be one of characters: 'a' (append to existing file), 'x' (only write if file doesn't exist yet), or 'w' (overwrite file if it already exists).
- **filename** (*str or string.Template*) – Filename of created .xlsx file or a template for generation of the name using `make_name()` method.

write(*data: List*) → None

Writes `DataArray`-like objects to disk, decides how to write them based on the type of each object. If some types of given objects are not supported by this writer, data of this type is ignored and a warning is emitted.

Parameters *data* (*List*) – `DataArray`-like objects that should be written to disk.

```
generic(data: List[Union[tesliper.glassware.arrays.DataArray, tesliper.glassware.arrays.IntegerArray,
                        tesliper.glassware.arrays.FloatArray, tesliper.glassware.arrays.BooleanArray,
                        tesliper.glassware.arrays.InfoArray]], name_template: Union[str, string.Template] = '${cat}.${det}')
```

Writes generic data from multiple `DataArray`-like objects to a single sheet. Said objects should provide a single value for each conformer.

Parameters

- **data** – `DataArray` objects that are to be exported.
- **name_template** – Template that will be used to generate filenames. Refer to `make_name()` documentation for details on supported placeholders.

```
overview(energies: Sequence[tesliper.glassware.arrays.Energies], frequencies:
Optional[tesliper.glassware.arrays.DataArray] = None, stoichiometry:
Optional[tesliper.glassware.arrays.InfoArray] = None, name_template: Union[str, string.Template]
= '${cat}')
```

Writes summarized information from multiple `Energies` objects to xlsx file. Creates a worksheet with energy values and calculated populations for each energy object given, as well as number of imaginary frequencies and stoichiometry of conformers if *frequencies* and *stoichiometry* are provided, respectively.

Parameters

- **energies** (*list of glassware.Energies*) – `Energies` objects that are to be exported

- **frequencies** (*glassware.DataArray, optional*) – DataArray object containing frequencies
- **stoichiometry** (*glassware.InfoArray, optional*) – InfoArray object containing stoichiometry information
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames, defaults to “\${cat}”. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

energies (*energies: tesliper.glassware.arrays.Energies, corrections: Optional[tesliper.glassware.arrays.FloatArray] = None, name_template: Union[str, string.Template] = 'distribution-\${genre}'*)

Writes detailed information from multiple [Energies](#) objects to xlsx file. Creates one worksheet for each [Energies](#) object provided. The sheet contains energy values, energy difference to lowest-energy conformer, Boltzmann factor, population of each conformer and corrections, if those are provided.

Parameters

- **energies** (*list of glassware.Energies*) – Energies objects that are to be exported
- **corrections** (*list of glassware.DataArray*) – DataArray objects containing energies corrections
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames, defaults to “distribution-\${genre}”. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

spectral_data (*band: tesliper.glassware.arrays.SpectralActivities, data: Iterable[tesliper.glassware.arrays.SpectralData], name_template: Union[str, string.Template] = '\${conf}.\${cat}-\${det}'*)

Writes [SpectralData](#) objects to xlsx file (one sheet for each conformer).

Parameters

- **band** (*glassware.SpectralActivities*) – object containing information about band at which transitions occur; it should be frequencies for vibrational data and wavelengths or excitation energies for electronic data
- **data** (*iterable of glassware.SpectralData*) – SpectralData objects that are to be serialized; all should contain information for the same conformers. Assumes that all *data*’s elements have the same *spectra_type*, which is passed to the *name_template* as “det”.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames, defaults to “\${conf}.\${cat}-\${det}”. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

Raises ValueError – if *data* is an empty sequence

spectral_activities (*band: tesliper.glassware.arrays.SpectralActivities, data: Iterable[tesliper.glassware.arrays.SpectralActivities], name_template: Union[str, string.Template] = '\${conf}.\${cat}-\${det}'*)

Writes [SpectralActivities](#) objects to xlsx file (one sheet for each conformer).

Parameters

- **band** (*glassware.SpectralActivities*) – object containing information about band at which transitions occur; it should be frequencies for vibrational data and wavelengths or excitation energies for electronic data

- **data** (*iterable of glassware.SpectralActivities*) – SpectralActivities objects that are to be serialized; all should contain information for the same conformers. Assumes that all *data*’s elements have the same *spectra_type*, which is passed to the *name_template* as “det”.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames, defaults to “\${conf}.\${cat}-\${det}”. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

Raises ValueError – if *data* is an empty sequence

spectra(*spectra: tesliper.glassware.spectra.Spectra, name_template: Union[str, string.Template] = '\${genre}'*)

Writes given spectra collectively to one sheet of xlsx workbook.

Parameters

- **spectra** (*glassware.Spectra*) – Spectra object, that is to be serialized
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames, defaults to “\${genre}”. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

single_spectrum(*spectrum: tesliper.glassware.spectra.SingleSpectrum, name_template: Union[str, string.Template] = '\${cat}.\${genre}-\${det}'*)

Writes SingleSpectrum object to new sheet of xlsx workbook.

Parameters

- **spectrum** (*glassware.SingleSpectrum*) – spectrum, that is to be serialized
- **name_template** (*str or string.Template*) – Template that will be used to generate sheet names, defaults to “\${cat}.\${genre}-\${det}”. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

_get_handle(*template: Union[str, string.Template], template_params: dict, open_params: Optional[dict] = None*) → *Iterator[IO]*

Helper method for creating files. Given additional kwargs will be passed to `Path.open()` method. Implemented as context manager for use with `with` statement.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames.
- **template_params** (*dict*) – Dictionary of {identifier: value} for *.make_name* method.
- **open_params** (*dict, optional*) – Arguments for `Path.open()` used to open file.

Yields

- *IO* – file handle, will be closed automatically after `with` statement exits
- *meta public:*

_iter_handles(*filenames: Iterable[str], template: Union[str, string.Template], template_params: dict, open_params: Optional[dict] = None*) → *Iterator[IO]*

Helper method for iteration over generated files. Given additional kwargs will be passed to `Path.open()` method.

Parameters

- **filenames** (*list of str*) – list of source filenames, used as value for `#{conf}` placeholder in `name_template`
- **template_params** (*dict*) – Dictionary of {identifier: value} for `.make_name` method.
- **open_params** (*dict, optional*) – arguments for `Path.open()` used to open file.

Yields

- *TextIO* – file handle, will be closed automatically on next iteration
- *meta public*:

property destination: `pathlib.Path`

Directory, to which generated files should be written.

Raises `FileNotFoundError` – If given destination doesn't exist or is not a directory.

Type `pathlib.Path`

static distribute_data(*data: List*) → `Tuple[Dict[str, List], Dict[str, Any]]`

Sorts given data by genre category for use by specialized writing methods.

Returns

- **distr** (*dict*) – Dictionary with *DataArray*-like objects, sorted by their type. Each {key: value} pair is {name of the type in lowercase format: list of *DataArray* objects of this type}.
- **extras** (*dict*) – Spacial-case genres: extra information used by some writer methods when exporting data. Available {key: value} pairs (if given in *data*) are:

corrections: dict of {"energy genre": *FloatArray*},
frequencies: *Bands*,
wavelengths: *Bands*,
excitation: *Bands*,
stoichiometry: *InfoArray*,
charge: *IntegerArray*,
multiplicity: *IntegerArray*

geometry(*geometry: tesliper.glassware.arrays.Geometry*, *charge:*

Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None, *multiplicity:*

Optional[Union[tesliper.glassware.arrays.IntegerArray, Sequence[int], int]] = None,

name_template: Union[str, string.Template] = "")

Interface for writing single object with geometry of each conformer. Evoked when handling *Geometry* objects.

Parameters

- **geometry** – Positions of atoms in each conformer. Mandatory in custom implementation.
- **charge** – Value of each structure's charge. Mandatory in custom implementation.
- **multiplicity** – Value of each structure's multiplicity. Mandatory in custom implementation.

- **name_template** – Template that defines naming scheme for files generated by this method. May be omitted in custom implementation.

Raises `NotImplementedError` – Whenever called, this is an interface that should not be used directly.

make_name(*template: Union[str, string.Template], conf: str = "", num: Union[str, int] = "", genre: str = "", cat: str = "", det: str = "", ext: str = ""*) → str

Create filename using given template and given or global values for known identifiers. The identifier should be used in the template as "\${identifier}" where “identifier” is the name of identifier. Available names and their meaning are:

\${ext} - appropriate file extension
 \${conf} - name of the conformer
 \${num} - number of the file according to internal counter
 \${genre} - genre of exported data
 \${cat} - category of produced output
 \${det} - category-specific detail

The \${ext} identifier is filled with the value of Writers *extension* attribute if not explicitly given as parameter to this method’s call. Values for other identifiers should be provided by the caller.

Parameters

- **template** (*str or string.Template*) – Template that will be used to generate filenames. It should contain only known identifiers, listed above.
- **conf** (*str*) – value for \${conf} identifier, defaults to empty string.
- **num** (*str or int*) – value for \${str} identifier, defaults to empty string.
- **genre** (*str*) – value for \${genre} identifier, defaults to empty string.
- **cat** (*str*) – value for \${cat} identifier, defaults to empty string.
- **det** (*str*) – value for \${det} identifier, defaults to empty string.
- **ext** (*str*) – value for \${ext} identifier, defaults to empty string.

Raises `ValueError` – If given template or string contains any unexpected identifiers.

Examples

Must be first subclassed and instantiated:

```
>>> class MyWriter(WriterBase):
>>>     extension = "foo"
>>> wrt = MyWriter("/path/to/some/directory/")
```

```
>>> wrt.make_name(template="somefile.${ext}")
"somefile.foo"
>>> wrt.make_name(template="${conf}.${ext}")
".foo" # conf is empty string by default
>>> wrt.make_name(template="${conf}.${ext}", conf="conformer")
"conformer.foo"
```

(continues on next page)

(continued from previous page)

```
>>> wrt.make_name(template="Unknown_identifier_{$bla}.{$ext}")
Traceback (most recent call last):
  ValueError: Unexpected identifiers given: bla.
```

property mode

Specifies how writing to file should be handled. Should be one of characters: “a”, “x”, or “w”. “a” - append to existing file; “x” - only write if file doesn’t exist yet; “w” - overwrite file if it already exists.

Raises `ValueError` – If given anything other than “a”, “x”, or “w”.

transitions(*transitions*: [tesliper.glassware.arrays.Transitions](#), *wavelengths*: [tesliper.glassware.arrays.Bands](#), *only_highest*=*True*, *name_template*: *Union[str, string.Template]* = *'\${conf}.\${cat}-\${det}'*)

Writes electronic transitions data to xlsx file (one sheet for each conformer).

Parameters

- **transitions** (*glassware.Transitions*) – Electronic transitions data that should be serialized.
- **wavelengths** (*glassware.ElectronicActivities*) – Object containing information about wavelength at which transitions occur.
- **only_highest** (*bool*) – Specifies if only transition of highest contribution to given band should be reported. If *False* all transition are saved to file. Defaults to *True*.
- **name_template** (*str or string.Template*) – Template that will be used to generate filenames, defaults to “*\${conf}.\${cat}-\${det}*”. Refer to [make_name\(\)](#) documentation for details on supported placeholders.

3.9 Change Log

3.9.1 v. 0.9.3

GUI:

- Added button for recursive extraction.

Other Changes:

- Now warning will be issued after reading abnormally terminated files.
- Minor corrections in the documentation.

3.9.2 v. 0.9.2

Bug Fixes:

- Fixed `__version__` and other metadata attributes broken in 0.9.1.

3.9.3 v. 0.9.1

Bug Fixes:

- Fixed `ImportError` occurring in Python 3.10.
- Corrected creation of "filanemes" pseudo-genre.
- Corrected `len()` behavior with `Spectra` instances.

New Features:

- Added "top-level" temperature setting in both, API and GUI.
- Allowed ignoring of unexpected keyword arguments in `Conformers.arrayed()`.

Other Changes:

- Moved requirements to `setup.py` file.
- Added `tesliper-gui` entry point.
- `Tesliper.get_averaged_spectrum()` now tries to calculate missing spectra.
- Minor supplementation to documentation and README.

3.9.4 v. 0.9.0

Created online documentation! Available at <https://tesliper.readthedocs.io/>

Bug Fixes:

- Fixed error on parsing radical molecules.
- Corrected `ArrayProperty` ignoring its `.fill_value`.
- Fixed infinite recursion error on `SpectralData.wavelen` access.
- Prevented creation of empty files on export of empty data arrays.
- Prevented intermediate `.xlsx` file saving when exporting multiple data genres.
- Corrected trimming abnormally terminated conformers in GUI.

New Features:

- `rmsd_sieve` and `Conformers.trim_rmsd` now allow for arbitrary windows.
- Added `datawork.geometry.pyramid_windows` window strategy function.
- Extended `Soxhlet` to allow use of arbitrary registered parsers.
- Allowed for automatic instantiation of data arrays for genres that depend on a different genre.
- Introduced *optimized_geom* genre
- Added export of generic data arrays.
- Added parametrization of `GjfWriter.link0` commands.

Other Changes:

- Reviewed and corrected calculation of intensities.
- Improved automatic scaling of spectra.
- Renamed `Parser` to `ParserBase` for consistency with other base classes.

- Unified base classes' registering mechanism of their subclasses.
- Cleaned up `extraction.gaussian_parser`. Changed all data sequences to lists.
- Supplemented type hints.
- Renamed *geometry* genre to *last_read_geom*.
- Supplemented `Conformers` to fully implement `OrderedDict` interface.
- Added storage and serialization of experimental spectra.

GUI:

- Unified terminology used with the one in code and documentation.

3.9.5 v. 0.8.2

API:

- Corrected data export when `Tesliper`'s default genres used.
- Corrected error when `Tesliper.calculate_spectra` called with default values.
- Corrected default filenames generated for spectral data and activities.
- Supplemented genres' full names and other metadata.

3.9.6 v. 0.8.1

API:

- Corrected handling of invalid start, stop, step parameters combination when calculating spectra.

GUI:

- Fixed incorrect floats' rounding in numeric entries.
- Added reaction (trim conformers/redraw spectra) to "Enter" key press, when editing a numeric entry.
- Fixed an error occurring when "show activities" is checked but there are no activities in a plotting range.
- Added auto-update of energies-related values after trimming.

3.9.7 v. 0.8.0

API:

- added RMSD-based trimming of conformers with similar geometry
- added auto scaling and shifting spectra to match reference
- added support for handling and exporting electronic transitions
- added export to .gjf files
- added serialization of `Tesliper` class
- renamed `Molecules` class to `Conformers`
- significant changes to `...Writer` classes
- significant changes to `DataArray` subclasses

- major code refactoring
- many smaller changes and improvements

GUI:

- new application layout
- added scroll response to numeric fields
- changed available and default colour schemes
- supplemented data export options

3.9.8 v. 0.7.4**API:**

- Tesliper's method 'average_spectra' returns reference to dict of averaged spectra

GUI:

- fixed files export (broken in v. 0.7.3)

3.9.9 v. 0.7.3**API:**

- introduced exceptions.py submodule
- glassware module turned into package
- improved mechanism for dealing with inconsistent data sizes
- added mechanism for trimming conformers with inconsistent data sizes
- fixed Molecules' trim_incomplete function
- enhanced Molecules' trim_non_matching_stoichiometry function
- introduced dict_view classes for iteration through trimmed Molecules
- improved Molecules indexing mechanism to return in O(1)
- removed 'cpu_time' from data extracted by gaussian_parser
- fixed error on parsing ECD calculations from g.09B

GUI:

- fixed problem with stacked spectra drawing
- added spectra reversing on demand
- fixed stacked spectra coloring
- corrected bars drawing for uv and ecd spectra
- added option for filtering conformers with inconsistent data sizes
- split un/check into separate buttons
- fixed checking/unchecking incomplete entries
- added checking/unchecking inconsistent sizes
- other minor changes and fixes

3.9.10 v. 0.7.2

- added support for string 'genres' parameter in `Tesliper.calculate_spectra` method
- added support for .xy spectra files
- gui: fixed problem with averaged and stacked spectra drawing
- gui: set "user_home_dir/tesliper/" as default location for `tslr_err_log.exe`
- other minor fixes and enhancements

3.9.11 v. 0.7.1

- fixed crash on spectra drawing when Matplotlib 3 used
- fixed problem with loading spectra from some txt files
- added support for loading spectra from csv files
- other minor fixes

3.9.12 v. 0.7.0

- graphical user interface redesigned
- significant changes in code architecture
- many fixes

3.9.13 v. 0.6.4

- calculated spectra precision in txt files changed to e-4
- spectra lines width changed
- data trimming features corrected
- spectra plot erasing on session clearing implemented
- inverting x axis for uv and ecd spectra added

3.9.14 v. 0.6.3

- fixed export error when not chosen, but all data were exported
- fixed export error when export occurred after closing popup window
- fixed export error when energies were not exported to separate txt files
- entry validation improved

3.9.15 v. 0.6.2

- solved some problems with corrupted files extraction
- added warning when files from mixed gaussian runs found
- fixed RuntimeError on overlapping actions
- fixed export popup error
- errors description moved to tslr_err_log.txt
- fixed ValueError on empty settings in gui_main.current_settings
- corrected session instantiation from files (unwanted files problem)
- changed energies precision to .6
- added Min. Boltzmann factor in GUI

3.9.16 v. 0.6.1

First beta release

3.9.17 v. 0.6.0 and earlier

Early development stages

3.10 Index

PYTHON MODULE INDEX

t

- [tesliper](#), 59
- [tesliper.datawork](#), 60
 - [tesliper.datawork.atoms](#), 60
 - [tesliper.datawork.energies](#), 61
 - [tesliper.datawork.geometry](#), 63
 - [tesliper.datawork.intensities](#), 68
 - [tesliper.datawork.spectra](#), 70
- [tesliper.exceptions](#), 74
- [tesliper.extraction](#), 74
 - [tesliper.extraction.gaussian_parser](#), 75
 - [tesliper.extraction.parameters_parser](#), 79
 - [tesliper.extraction.parser_base](#), 81
 - [tesliper.extraction.soxhlet](#), 84
 - [tesliper.extraction.spectra_parser](#), 86
- [tesliper.glassware](#), 89
 - [tesliper.glassware.array_base](#), 89
 - [tesliper.glassware.arrays](#), 100
 - [tesliper.glassware.conformers](#), 116
 - [tesliper.glassware.spectra](#), 126
- [tesliper.tesliper](#), 129
- [tesliper.writing](#), 140
 - [tesliper.writing.csv_writer](#), 140
 - [tesliper.writing.gjf_writer](#), 146
 - [tesliper.writing.serializer](#), 153
 - [tesliper.writing.txt_writer](#), 154
 - [tesliper.writing.writer_base](#), 160
 - [tesliper.writing.xlsx_writer](#), 169

Symbols

`_get_handle()` (*tesliper.writing.csv_writer.CsvWriter* method), 141

`_get_handle()` (*tesliper.writing.gif_writer.GifWriter* method), 147

`_get_handle()` (*tesliper.writing.txt_writer.TxtWriter* method), 157

`_get_handle()` (*tesliper.writing.writer_base.WriterBase* method), 165

`_get_handle()` (*tesliper.writing.xlsx_writer.XlsxWriter* method), 171

`_iter_handles()` (*tesliper.writing.csv_writer.CsvWriter* method), 141

`_iter_handles()` (*tesliper.writing.gif_writer.GifWriter* method), 147

`_iter_handles()` (*tesliper.writing.txt_writer.TxtWriter* method), 157

`_iter_handles()` (*tesliper.writing.writer_base.WriterBase* method), 165

`_iter_handles()` (*tesliper.writing.xlsx_writer.XlsxWriter* method), 171

A

`activities` (*tesliper.tesliper.Tesliper* property), 133

`add_state()` (*tesliper.extraction.gaussian_parser.GaussianParser* method), 78

`add_state()` (*tesliper.extraction.parser_base.ParserBase* method), 82

`add_state()` (*tesliper.extraction.spectra_parser.SpectraParser* method), 88

`all_files` (*tesliper.extraction.soxhlet.Soxhlet* property), 84

`all_have_genres()` (*tesliper.glassware.conformers.Conformers* method), 120

`ArchiveLoader` (class in *tesliper.writing.serializer*), 154

`ArchiveWriter` (class in *tesliper.writing.serializer*), 153

`ArrayBase` (class in *tesliper.glassware.array_base*), 98

`arrayed`, 8

`arrayed()` (*tesliper.glassware.conformers.Conformers* method), 119

`ArrayProperty` (class in *tesliper.glassware.array_base*), 94

`as_kcal_per_mol` (*tesliper.glassware.arrays.Energies* property), 103

`associated_genres` (*tesliper.glassware.array_base.ArrayBase* property), 99

`Atom` (class in *tesliper.datawork.atoms*), 60

`atomic_number()` (in module *tesliper.datawork.atoms*), 61

`Averagable` (class in *tesliper.glassware.arrays*), 104

`average()` (*tesliper.glassware.spectra.Spectra* method), 128

`average_conformers()` (*tesliper.glassware.arrays.Averagable* method), 104

`average_conformers()` (*tesliper.glassware.arrays.ElectronicActivities* method), 113

`average_conformers()` (*tesliper.glassware.arrays.ScatteringActivities* method), 111

`average_conformers()` (*tesliper.glassware.arrays.SpectralActivities* method), 109

`average_conformers()` (*tesliper.glassware.arrays.VibrationalActivities* method), 110

`average_spectra()` (*tesliper.tesliper.Tesliper* method), 137

`averaged` (*tesliper.tesliper.Tesliper* attribute), 131

B

`Bands` (class in *tesliper.glassware.arrays*), 104

`BOLTZMANN` (in module *tesliper.datawork.energies*), 62

`BooleanArray` (class in *tesliper.glassware.arrays*), 103

`by_index()` (*tesliper.glassware.conformers.Conformers* method), 119

C

`calc_rmsd()` (in module `tesliper.datawork.geometry`), 65
`calculate_average()` (in module `tesliper.datawork.spectra`), 71
`calculate_deltas()` (in module `tesliper.datawork.energies`), 62
`calculate_min_factors()` (in module `tesliper.datawork.energies`), 62
`calculate_populations()` (in module `tesliper.datawork.energies`), 62
`calculate_populations()` (`tesliper.glassware.arrays.Energies` method), 104
`calculate_single_spectrum()` (`tesliper.tesliper.Tesliper` method), 136
`calculate_spectra()` (in module `tesliper.datawork.spectra`), 71
`calculate_spectra()` (`tesliper.glassware.arrays.ElectronicActivities` method), 113
`calculate_spectra()` (`tesliper.glassware.arrays.ScatteringActivities` method), 112
`calculate_spectra()` (`tesliper.glassware.arrays.VibrationalActivities` method), 110
`calculate_spectra()` (`tesliper.tesliper.Tesliper` method), 136
`center()` (in module `tesliper.datawork.geometry`), 65
`check_input()` (`tesliper.glassware.array_base.ArrayProperty` method), 95
`check_input()` (`tesliper.glassware.array_base.CollapsibleArrayProperty` method), 98
`check_input()` (`tesliper.glassware.array_base.JaggedArrayProperty` method), 96
`check_shape()` (`tesliper.glassware.array_base.ArrayProperty` method), 95
`check_shape()` (`tesliper.glassware.array_base.CollapsibleArrayProperty` method), 98
`check_shape()` (`tesliper.glassware.array_base.JaggedArrayProperty` method), 97
`clear()` (`tesliper.glassware.conformers.Conformers` method), 117
`clear()` (`tesliper.tesliper.Tesliper` method), 132
`coefficients` (`tesliper.glassware.arrays.Transitions` property), 115
`CollapsibleArrayProperty` (class in `tesliper.glassware.array_base`), 97
`Conformers` (class in `tesliper.glassware.conformers`), 116
`conformers` (`tesliper.tesliper.Tesliper` attribute), 131
`contribution` (`tesliper.glassware.arrays.Transitions` property), 115
`convert_band()` (in module `tesliper.datawork.spectra`),

73

`copy()` (`tesliper.glassware.conformers.Conformers` method), 117
`count_imaginary()` (in module `tesliper.datawork.spectra`), 70
`CsvWriter` (class in `tesliper.writing.csv_writer`), 140

D

`data` (`tesliper.extraction.gaussian_parser.GaussianParser` attribute), 77
`data array`, 8
`data inconsistency`, 9
`DataArray` (class in `tesliper.glassware.arrays`), 100
`deleter()` (`tesliper.glassware.array_base.ArrayProperty` method), 95
`deleter()` (`tesliper.glassware.array_base.CollapsibleArrayProperty` method), 98
`deleter()` (`tesliper.glassware.array_base.JaggedArrayProperty` method), 97
`deltas` (`tesliper.glassware.arrays.Energies` property), 103
`DependentParameter` (class in `tesliper.glassware.array_base`), 99
`destination` (`tesliper.writing.csv_writer.CsvWriter` property), 142
`destination` (`tesliper.writing.gif_writer.GifWriter` property), 148
`destination` (`tesliper.writing.serializer.ArchiveWriter` property), 154
`destination` (`tesliper.writing.txt_writer.TxtWriter` property), 157
`destination` (`tesliper.writing.writer_base.WriterBase` property), 163
`destination` (`tesliper.writing.xlsx_writer.XlsxWriter` property), 172
`dialect` (`tesliper.writing.csv_writer.CsvWriter` property), 142
`display()` (in module `tesliper.datawork.intensities`), 68
`dipole_moment()` (in module `tesliper.datawork.intensities`), 69
`distribute_data()` (`tesliper.writing.csv_writer.CsvWriter` static method), 143
`distribute_data()` (`tesliper.writing.gif_writer.GifWriter` static method), 148
`distribute_data()` (`tesliper.writing.txt_writer.TxtWriter` static method), 158
`distribute_data()` (`tesliper.writing.writer_base.WriterBase` static method), 163

`distribute_data()` (*tesliper.writing.xlsx_writer.XlsxWriter* static method), 172

`drop_atoms()` (in module *tesliper.datawork.geometry*), 64

E

`ElectronicActivities` (class in *tesliper.glassware.arrays*), 112

`ElectronicData` (class in *tesliper.glassware.arrays*), 108

`empty` (*tesliper.glassware.array_base.DependentParameter* attribute), 99

`Energies` (class in *tesliper.glassware.arrays*), 103

`energies` (*tesliper.tesliper.Tesliper* property), 133

`energies()` (*tesliper.writing.csv_writer.CsvWriter* method), 142

`energies()` (*tesliper.writing.gif_writer.GifWriter* method), 148

`energies()` (*tesliper.writing.txt_writer.TxtWriter* method), 155

`energies()` (*tesliper.writing.writer_base.WriterBase* method), 166

`energies()` (*tesliper.writing.xlsx_writer.XlsxWriter* method), 170

`energies_order` (*tesliper.writing.writer_base.WriterBase* attribute), 163

`ex_en` (*tesliper.glassware.arrays.Bands* property), 105

`excitation_energy` (*tesliper.glassware.arrays.Bands* property), 105

`excited` (*tesliper.glassware.arrays.Transitions* attribute), 114

`excited()` (*tesliper.extraction.gaussian_parser.GaussianParser* method), 78

`experimental` (*tesliper.tesliper.Tesliper* attribute), 132

`export_activities()` (*tesliper.tesliper.Tesliper* method), 138

`export_averaged()` (*tesliper.tesliper.Tesliper* method), 139

`export_data()` (*tesliper.tesliper.Tesliper* method), 137

`export_energies()` (*tesliper.tesliper.Tesliper* method), 138

`export_job_file()` (*tesliper.tesliper.Tesliper* method), 139

`export_spectra()` (*tesliper.tesliper.Tesliper* method), 138

`export_spectral_data()` (*tesliper.tesliper.Tesliper* method), 138

`extension` (*tesliper.writing.writer_base.WriterBase* property), 163

`extensions` (*tesliper.extraction.parser_base.ParserBase* property), 81

`extract()` (*tesliper.extraction.soxhlet.Soxhlet* method), 85

`extract()` (*tesliper.tesliper.Tesliper* method), 135

`extract_iter()` (*tesliper.extraction.soxhlet.Soxhlet* method), 85

`extract_iterate()` (*tesliper.tesliper.Tesliper* method), 134

F

`FileNamesArray` (class in *tesliper.glassware.arrays*), 102

`files` (*tesliper.extraction.soxhlet.Soxhlet* property), 84

`filter_files()` (*tesliper.extraction.soxhlet.Soxhlet* method), 85

`find_atoms()` (in module *tesliper.datawork.geometry*), 63

`find_best_shape()` (in module *tesliper.glassware.array_base*), 92

`find_imaginary()` (in module *tesliper.datawork.spectra*), 70

`find_imaginary()` (*tesliper.glassware.arrays.Bands* method), 105

`find_offset()` (in module *tesliper.datawork.spectra*), 73

`find_scaling()` (in module *tesliper.datawork.spectra*), 73

`fitting()` (in module *tesliper.extraction.parameters_parser*), 80

`fixed_windows()` (in module *tesliper.datawork.geometry*), 66

`flatten()` (in module *tesliper.glassware.array_base*), 92

`FloatArray` (class in *tesliper.glassware.arrays*), 101

`fmtparams` (*tesliper.writing.csv_writer.CsvWriter* property), 143

`freq` (*tesliper.glassware.arrays.Bands* property), 105

`freq` (*tesliper.glassware.arrays.ElectronicActivities* property), 113

`freq` (*tesliper.glassware.arrays.ElectronicData* property), 108

`freq` (*tesliper.glassware.arrays.SpectralActivities* property), 109

`freq` (*tesliper.glassware.arrays.SpectralData* property), 106

`frequencies` (*tesliper.glassware.arrays.Bands* property), 105

`frequencies` (*tesliper.glassware.arrays.ElectronicActivities* property), 113

`frequencies` (*tesliper.glassware.arrays.ElectronicData* property), 108

`frequencies` (*tesliper.glassware.arrays.ScatteringActivities* property), 112

`frequencies` (*tesliper.glassware.arrays.ScatteringData* property), 107

`frequencies` (*tesliper.glassware.arrays.SpectralActivities* property), 109

frequencies (tesliper.glassware.arrays.SpectralData property), 106	get_init_params() (tesliper.glassware.arrays.BooleanArray class method), 103
frequencies (tesliper.glassware.arrays.VibrationalActivities property), 110	get_init_params() (tesliper.glassware.arrays.DataArray class method), 100
frequencies (tesliper.glassware.arrays.VibrationalData property), 107	get_init_params() (tesliper.glassware.arrays.ElectronicActivities class method), 113
frequencies() (tesliper.extraction.gaussian_parser.GaussianParser class method), 78	get_init_params() (tesliper.glassware.arrays.ElectronicData class method), 108
from_parameter() (tesliper.glassware.array_base.DependentParameter class method), 99	get_init_params() (tesliper.glassware.arrays.Energies class method), 104
fromkeys() (tesliper.glassware.conformers.Conformers class method), 124	get_init_params() (tesliper.glassware.arrays.FilenamesArray class method), 102
G	get_init_params() (tesliper.glassware.arrays.FloatArray class method), 101
gaussian() (in module tesliper.datawork.spectra), 70	get_init_params() (tesliper.glassware.arrays.Geometry class method), 116
GaussianParser (class in tesliper.extraction.gaussian_parser), 75	get_init_params() (tesliper.glassware.arrays.InfoArray class method), 102
generic() (tesliper.writing.csv_writer.CsvWriter class method), 141	get_init_params() (tesliper.glassware.arrays.IntegerArray class method), 101
generic() (tesliper.writing.gif_writer.GifWriter class method), 148	get_init_params() (tesliper.glassware.arrays.ScatteringActivities class method), 112
generic() (tesliper.writing.txt_writer.TxtWriter class method), 155	get_init_params() (tesliper.glassware.arrays.ScatteringData class method), 107
generic() (tesliper.writing.writer_base.WriterBase class method), 166	get_init_params() (tesliper.glassware.arrays.SpectralActivities class method), 109
generic() (tesliper.writing.xlsx_writer.XlsxWriter class method), 169	get_init_params() (tesliper.glassware.arrays.SpectralData class method), 106
genre, 8	get_init_params() (tesliper.glassware.arrays.Transitions class method), 115
genre_getter (tesliper.glassware.array_base.DependentParameter property), 99	get_init_params() (tesliper.glassware.arrays.VibrationalActivities class method), 110
Geometry (class in tesliper.glassware.arrays), 115	get_init_params() (tesliper.glassware.arrays.VibrationalData class method), 107
geometry() (tesliper.extraction.gaussian_parser.GaussianParser class method), 77	get_repr_args() (tesliper.glassware.array_base.ArrayBase class method), 99
geometry() (tesliper.writing.csv_writer.CsvWriter class method), 143	
geometry() (tesliper.writing.gif_writer.GifWriter class method), 147	
geometry() (tesliper.writing.txt_writer.TxtWriter class method), 158	
geometry() (tesliper.writing.writer_base.WriterBase class method), 168	
geometry() (tesliper.writing.xlsx_writer.XlsxWriter class method), 172	
get() (tesliper.glassware.conformers.Conformers class method), 124	
get_averaged_spectrum() (tesliper.tesliper.Tesliper class method), 137	
get_init_params() (tesliper.glassware.array_base.ArrayBase class method), 99	
get_init_params() (tesliper.glassware.arrays.Bands class method), 105	

- `get_repr_args()` (*tesliper.glassware.arrays.Bands* method), 105
`get_repr_args()` (*tesliper.glassware.arrays.BooleanArray* method), 103
`get_repr_args()` (*tesliper.glassware.arrays.DataArray* method), 100
`get_repr_args()` (*tesliper.glassware.arrays.ElectronicActivities* method), 114
`get_repr_args()` (*tesliper.glassware.arrays.ElectronicData* method), 108
`get_repr_args()` (*tesliper.glassware.arrays.Energies* method), 104
`get_repr_args()` (*tesliper.glassware.arrays.FilenamesArray* method), 103
`get_repr_args()` (*tesliper.glassware.arrays.FloatArray* method), 101
`get_repr_args()` (*tesliper.glassware.arrays.Geometry* method), 116
`get_repr_args()` (*tesliper.glassware.arrays.InfoArray* method), 102
`get_repr_args()` (*tesliper.glassware.arrays.IntegerArray* method), 101
`get_repr_args()` (*tesliper.glassware.arrays.ScatteringActivities* method), 112
`get_repr_args()` (*tesliper.glassware.arrays.ScatteringData* method), 107
`get_repr_args()` (*tesliper.glassware.arrays.SpectralActivities* method), 109
`get_repr_args()` (*tesliper.glassware.arrays.SpectralData* method), 106
`get_repr_args()` (*tesliper.glassware.arrays.Transitions* method), 115
`get_repr_args()` (*tesliper.glassware.arrays.VibrationalActivities* method), 111
`get_repr_args()` (*tesliper.glassware.arrays.VibrationalData* method), 107
`get_triangular()` (in module *tesliper.datawork.geometry*), 65
`get_triangular_base()` (in module *tesliper.datawork.geometry*), 65
`getter()` (*tesliper.glassware.array_base.ArrayProperty* method), 95
`getter()` (*tesliper.glassware.array_base.CollapsibleArrayProperty* method), 98
`getter()` (*tesliper.glassware.array_base.JaggedArrayProperty* method), 97
GjfWriter (class in *tesliper.writing.gjf_writer*), 146
`ground` (*tesliper.glassware.arrays.Transitions* attribute), 114
`guess_extension()` (*tesliper.extraction.soxhlet.Soxhlet* method), 85
- ## H
- `HARTREE_TO_KCAL_PER_MOL` (in module *tesliper.datawork.energies*), 62
`has_any_genre()` (*tesliper.glassware.conformers.Conformers* method), 120
`has_genre()` (*tesliper.glassware.conformers.Conformers* method), 119
`highest_contribution` (*tesliper.glassware.arrays.Transitions* property), 115
- ## I
- `idx_offset()` (in module *tesliper.datawork.spectra*), 72
`imaginary` (*tesliper.glassware.arrays.Bands* property), 105
`inconsistency_allowed` (*tesliper.glassware.conformers.Conformers* property), 125
`InconsistentDataError`, 74
`index_of()` (*tesliper.glassware.conformers.Conformers* method), 119
`indices_highest` (*tesliper.glassware.arrays.Transitions* property), 115
`InfoArray` (class in *tesliper.glassware.arrays*), 101
`initial()` (*tesliper.extraction.gaussian_parser.GaussianParser* method), 77
`initial()` (*tesliper.extraction.parser_base.ParserBase* method), 82
`initial()` (*tesliper.extraction.spectra_parser.SpectraParser* method), 86
`input_dir` (*tesliper.tesliper.Tesliper* property), 134
`IntegerArray` (class in *tesliper.glassware.arrays*), 101
`intensities` (*tesliper.glassware.arrays.ElectronicActivities* property), 113
`intensities` (*tesliper.glassware.arrays.ScatteringActivities* property), 111
`intensities` (*tesliper.glassware.arrays.SpectralActivities* property), 109
`intensities` (*tesliper.glassware.arrays.VibrationalActivities* property), 111
`InvalidElementError`, 74

`InvalidStateError`, 74

`is_triangular()` (in module `tesliper.datawork.geometry`), 64

`items()` (`tesliper.glassware.conformers.Conformers` method), 124

J

`JaggedArrayProperty` (class in `tesliper.glassware.array_base`), 95

`jsondecode()` (`tesliper.writing.serializer.ArchiveLoader` method), 154

`jsonencode()` (`tesliper.writing.serializer.ArchiveWriter` method), 154

K

`kabsch_rotate()` (in module `tesliper.datawork.geometry`), 65

`kept`, 8

`kept` (`tesliper.glassware.conformers.Conformers` property), 117

`kept_items()` (`tesliper.glassware.conformers.Conformers` method), 124

`kept_keys()` (`tesliper.glassware.conformers.Conformers` method), 123

`kept_values()` (`tesliper.glassware.conformers.Conformers` method), 124

`key_of()` (`tesliper.glassware.conformers.Conformers` method), 119

`keys()` (`tesliper.glassware.conformers.Conformers` method), 124

L

`link0` (`tesliper.writing.gjf_writer.GjfWriter` property), 152

`load()` (`tesliper.tesliper.Tesliper` class method), 140

`load_experimental()` (`tesliper.tesliper.Tesliper` method), 136

`load_parameters()` (`tesliper.tesliper.Tesliper` method), 135

`longest_subsequences()` (in module `tesliper.glassware.array_base`), 91

`lorentzian()` (in module `tesliper.datawork.spectra`), 71

M

`make_name()` (`tesliper.writing.csv_writer.CsvWriter` method), 143

`make_name()` (`tesliper.writing.gjf_writer.GjfWriter` method), 149

`make_name()` (`tesliper.writing.txt_writer.TxtWriter` method), 158

`make_name()` (`tesliper.writing.writer_base.WriterBase` method), 164

`make_name()` (`tesliper.writing.xlsx_writer.XlsxWriter` method), 173

`mask()` (in module `tesliper.glassware.array_base`), 92

`min_factors` (`tesliper.glassware.arrays.Energies` property), 103

`mode` (`tesliper.writing.csv_writer.CsvWriter` property), 144

`mode` (`tesliper.writing.gjf_writer.GjfWriter` property), 150

`mode` (`tesliper.writing.serializer.ArchiveWriter` property), 153

`mode` (`tesliper.writing.txt_writer.TxtWriter` property), 159

`mode` (`tesliper.writing.writer_base.WriterBase` property), 163

`mode` (`tesliper.writing.xlsx_writer.XlsxWriter` property), 174

module

`tesliper`, 59

`tesliper.datawork`, 60

`tesliper.datawork.atoms`, 60

`tesliper.datawork.energies`, 61

`tesliper.datawork.geometry`, 63

`tesliper.datawork.intensities`, 68

`tesliper.datawork.spectra`, 70

`tesliper.exceptions`, 74

`tesliper.extraction`, 74

`tesliper.extraction.gaussian_parser`, 75

`tesliper.extraction.parameters_parser`, 79

`tesliper.extraction.parser_base`, 81

`tesliper.extraction.soxhlet`, 84

`tesliper.extraction.spectra_parser`, 86

`tesliper.glassware`, 89

`tesliper.glassware.array_base`, 89

`tesliper.glassware.arrays`, 100

`tesliper.glassware.conformers`, 116

`tesliper.glassware.spectra`, 126

`tesliper.tesliper`, 129

`tesliper.writing`, 140

`tesliper.writing.csv_writer`, 140

`tesliper.writing.gjf_writer`, 146

`tesliper.writing.serializer`, 153

`tesliper.writing.txt_writer`, 154

`tesliper.writing.writer_base`, 160

`tesliper.writing.xlsx_writer`, 169

`move_to_end()` (`tesliper.glassware.conformers.Conformers` method), 117

O

`offset` (`tesliper.glassware.spectra.SingleSpectrum` property), 127

`offset` (`tesliper.glassware.spectra.Spectra` property), 128

`optimization()` (`tesliper.extraction.gaussian_parser.GaussianParser` method), 78

`optionxform()` (`tesliper.extraction.parameters_parser.ParametersParser` method), 80

`osc_to_uv()` (in module `tesliper.datawork.intensities`), 69

`output_dir` (`tesliper.tesliper.Tesliper` property), 134

`output_files` (`tesliper.extraction.soxhlet.Soxhlet` property), 85

`overview()` (`tesliper.writing.csv_writer.CsvWriter` method), 144

`overview()` (`tesliper.writing.gif_writer.GifWriter` method), 150

`overview()` (`tesliper.writing.txt_writer.TxtWriter` method), 155

`overview()` (`tesliper.writing.writer_base.WriterBase` method), 166

`overview()` (`tesliper.writing.xlsx_writer.XlsxWriter` method), 169

P

`parameters` (`tesliper.extraction.parameters_parser.ParametersParser` property), 80

`parameters` (`tesliper.tesliper.Tesliper` attribute), 132

`ParametersParser` (class in `tesliper.extraction.parameters_parser`), 80

`parse()` (`tesliper.extraction.gaussian_parser.GaussianParser` method), 77

`parse()` (`tesliper.extraction.parameters_parser.ParametersParser` method), 80

`parse()` (`tesliper.extraction.parser_base.ParserBase` method), 83

`parse()` (`tesliper.extraction.spectra_parser.SpectraParser` method), 86

`parse_csv()` (`tesliper.extraction.spectra_parser.SpectraParser` method), 87

`parse_one()` (`tesliper.extraction.soxhlet.Soxhlet` method), 86

`parse_spc()` (`tesliper.extraction.spectra_parser.SpectraParser` method), 87

`parse_txt()` (`tesliper.extraction.spectra_parser.SpectraParser` method), 87

`ParserBase` (class in `tesliper.extraction.parser_base`), 81

`pop()` (`tesliper.glassware.conformers.Conformers` method), 125

`popitem()` (`tesliper.glassware.conformers.Conformers` method), 117

`populations` (`tesliper.glassware.arrays.Energies` property), 104

`primary_genres` (`tesliper.glassware.conformers.Conformers` attribute), 116

`purpose` (`tesliper.extraction.parser_base.ParserBase` property), 82

`pyramid_windows()` (in module `tesliper.datawork.geometry`), 67

Q

`quantity()` (in module `tesliper.extraction.parameters_parser`), 79

`quantum_software` (`tesliper.tesliper.Tesliper` attribute), 132

R

`reject_all()` (`tesliper.glassware.conformers.Conformers` method), 123

`remove_state()` (`tesliper.extraction.gaussian_parser.GaussianParser` method), 78

`remove_state()` (`tesliper.extraction.parser_base.ParserBase` method), 82

`remove_state()` (`tesliper.extraction.spectra_parser.SpectraParser` method), 88

`replace()` (`tesliper.glassware.array_base.DependentParameter` method), 99

`rot_to_sieve()` (in module `tesliper.datawork.geometry`), 67

`rot_to_ecd()` (in module `tesliper.datawork.intensities`), 69

`rot_to_vcd()` (in module `tesliper.datawork.intensities`), 69

`route` (`tesliper.writing.gif_writer.GifWriter` property), 152

S

`sanitizer()` (`tesliper.glassware.array_base.ArrayProperty` method), 95

`sanitizer()` (`tesliper.glassware.array_base.CollapsibleArrayProperty` method), 98

`sanitizer()` (`tesliper.glassware.array_base.JaggedArrayProperty` method), 97

`scale_to()` (`tesliper.glassware.spectra.SingleSpectrum` method), 127

`scale_to()` (`tesliper.glassware.spectra.Spectra` method), 129

`scaling` (`tesliper.glassware.spectra.SingleSpectrum` property), 127

`scaling` (`tesliper.glassware.spectra.Spectra` property), 128

`ScatteringActivities` (class in `tesliper.glassware.arrays`), 111

`ScatteringData` (class in `tesliper.glassware.arrays`), 107

`select_all()` (`tesliper.glassware.conformers.Conformers` method), 123

`select_atoms()` (in module `tesliper.datawork.geometry`), 63

`serialize()` (`tesliper.tesliper.Tesliper` method), 139

`setdefault()` (`tesliper.glassware.conformers.Conformers` method), 125

`setter()` (`tesliper.glassware.array_base.ArrayProperty` method), 95

setter() (tesliper.glassware.array_base.CollapsibleArray property), 98
 setter() (tesliper.glassware.array_base.JaggedArrayProperty property), 97
 shift_to() (tesliper.glassware.spectra.SingleSpectrum method), 127
 shift_to() (tesliper.glassware.spectra.Spectra method), 129
 single_spectrum() (tesliper.writing.csv_writer.CsvWriter method), 142
 single_spectrum() (tesliper.writing.gif_writer.GifWriter method), 150
 single_spectrum() (tesliper.writing.txt_writer.TxtWriter method), 155
 single_spectrum() (tesliper.writing.writer_base.WriterBase method), 166
 single_spectrum() (tesliper.writing.xlsx_writer.XlsxWriter method), 171
 SingleSpectrum (class in tesliper.glassware.spectra), 126
 source (tesliper.writing.serializer.ArchiveLoader property), 154
 Soxhlet (class in tesliper.extraction.soxhlet), 84
 Spectra (class in tesliper.glassware.spectra), 127
 spectra (tesliper.tesliper.Tesliper attribute), 131
 spectra() (tesliper.writing.csv_writer.CsvWriter method), 145
 spectra() (tesliper.writing.gif_writer.GifWriter method), 150
 spectra() (tesliper.writing.txt_writer.TxtWriter method), 156
 spectra() (tesliper.writing.writer_base.WriterBase method), 167
 spectra() (tesliper.writing.xlsx_writer.XlsxWriter method), 171
 spectra_type (tesliper.glassware.arrays.ElectronicActivities property), 114
 spectra_type (tesliper.glassware.arrays.ElectronicData property), 108
 spectra_type (tesliper.glassware.arrays.ScatteringActivities property), 112
 spectra_type (tesliper.glassware.arrays.ScatteringData property), 107
 spectra_type (tesliper.glassware.arrays.SpectralActivities property), 109
 spectra_type (tesliper.glassware.arrays.SpectralData property), 106
 spectra_type (tesliper.glassware.arrays.VibrationalActivities property), 111
 Spectra_type (tesliper.glassware.arrays.VibrationalData property), 106
 spectra_type (tesliper.glassware.spectra.SingleSpectrum property), 127
 spectra_type (tesliper.glassware.spectra.Spectra property), 128
 spectral_activities() (tesliper.writing.csv_writer.CsvWriter method), 142
 spectral_activities() (tesliper.writing.gif_writer.GifWriter method), 151
 spectral_activities() (tesliper.writing.txt_writer.TxtWriter method), 156
 spectral_activities() (tesliper.writing.writer_base.WriterBase method), 167
 spectral_activities() (tesliper.writing.xlsx_writer.XlsxWriter method), 170
 spectral_data() (tesliper.writing.csv_writer.CsvWriter method), 145
 spectral_data() (tesliper.writing.gif_writer.GifWriter method), 151
 spectral_data() (tesliper.writing.txt_writer.TxtWriter method), 156
 spectral_data() (tesliper.writing.writer_base.WriterBase method), 167
 spectral_data() (tesliper.writing.xlsx_writer.XlsxWriter method), 170
 SpectralActivities (class in tesliper.glassware.arrays), 108
 SpectralData (class in tesliper.glassware.arrays), 105
 SpectraParser (class in tesliper.extraction.spectra_parser), 86
 standard_parameters (tesliper.tesliper.Tesliper property), 134
 state() (tesliper.extraction.gaussian_parser.GaussianParser static method), 78
 state() (tesliper.extraction.parser_base.ParserBase static method), 83
 state() (tesliper.extraction.spectra_parser.SpectraParser static method), 88
 states (tesliper.extraction.parser_base.ParserBase attribute), 81
 stretching_windows() (in module tesliper.datawork.geometry), 66
 symbol_of_element() (in module tesliper.datawork.atoms), 60

T

take_atoms() (in module *tesliper.datawork.geometry*), 64
 temperature (*tesliper.glassware.conformers.Conformers* property), 117
 temperature (*tesliper.tesliper.Tesliper* property), 132
 tesliper
 module, 59
 Tesliper (class in *tesliper.tesliper*), 130
 tesliper.datawork
 module, 60
 tesliper.datawork.atoms
 module, 60
 tesliper.datawork.energies
 module, 61
 tesliper.datawork.geometry
 module, 63
 tesliper.datawork.intensities
 module, 68
 tesliper.datawork.spectra
 module, 70
 tesliper.exceptions
 module, 74
 tesliper.extraction
 module, 74
 tesliper.extraction.gaussian_parser
 module, 75
 tesliper.extraction.parameters_parser
 module, 79
 tesliper.extraction.parser_base
 module, 81
 tesliper.extraction.soxhlet
 module, 84
 tesliper.extraction.spectra_parser
 module, 86
 tesliper.glassware
 module, 89
 tesliper.glassware.array_base
 module, 89
 tesliper.glassware.arrays
 module, 100
 tesliper.glassware.conformers
 module, 116
 tesliper.glassware.spectra
 module, 126
 tesliper.tesliper
 module, 129
 tesliper.writing
 module, 140
 tesliper.writing.csv_writer
 module, 140
 tesliper.writing.gjf_writer
 module, 146
 tesliper.writing.serializer
 module, 153
 tesliper.writing.txt_writer
 module, 154
 tesliper.writing.writer_base
 module, 160
 tesliper.writing.xlsx_writer
 module, 169
 TesliperError, 74
 to_masked() (in module *tesliper.glassware.array_base*), 93
 Transitions (class in *tesliper.glassware.arrays*), 114
 transitions() (*tesliper.writing.csv_writer.CsvWriter* method), 146
 transitions() (*tesliper.writing.gjf_writer.GjfWriter* method), 151
 transitions() (*tesliper.writing.txt_writer.TxtWriter* method), 156
 transitions() (*tesliper.writing.writer_base.WriterBase* method), 168
 transitions() (*tesliper.writing.xlsx_writer.XlsxWriter* method), 174
 triggers (*tesliper.extraction.parser_base.ParserBase* attribute), 81
 trim_imaginary_frequencies() (*tesliper.glassware.conformers.Conformers* method), 121
 trim_incomplete() (*tesliper.glassware.conformers.Conformers* method), 120
 trim_inconsistent_sizes() (*tesliper.glassware.conformers.Conformers* method), 121
 trim_non_matching_stoichiometry() (*tesliper.glassware.conformers.Conformers* method), 121
 trim_non_normal_termination() (*tesliper.glassware.conformers.Conformers* method), 121
 trim_not_optimized() (*tesliper.glassware.conformers.Conformers* method), 121
 trim_rmsd() (*tesliper.glassware.conformers.Conformers* method), 122
 trim_to_range() (*tesliper.glassware.conformers.Conformers* method), 122
 trimmed_to() (*tesliper.glassware.conformers.Conformers* method), 125
 trimming, 8
 TxtWriter (class in *tesliper.writing.txt_writer*), 154

U

unify_abscissa() (in module *tesliper.datawork.spectra*), 72

- units (*tesliper.glassware.spectra.SingleSpectrum* property), 127
- units (*tesliper.glassware.spectra.Spectra* property), 128
- unpack_values() (*tesliper.glassware.arrays.Transitions* static method), 114
- untrimmed (*tesliper.glassware.conformers.Conformers* property), 125
- update() (*tesliper.glassware.conformers.Conformers* method), 119
- update() (*tesliper.tesliper.Tesliper* method), 134
- ## V
- validate_atoms() (in module *tesliper.datawork.atoms*), 61
- values (*tesliper.glassware.arrays.FilenamesArray* property), 102
- values (*tesliper.glassware.arrays.Transitions* attribute), 114
- values() (*tesliper.glassware.conformers.Conformers* method), 125
- VibrationalActivities (class in *tesliper.glassware.arrays*), 109
- VibrationalData (class in *tesliper.glassware.arrays*), 106
- ## W
- wait() (*tesliper.extraction.gaussian_parser.GaussianParser* method), 77
- wanted_files (*tesliper.extraction.soxhlet.Soxhlet* property), 84
- wanted_files (*tesliper.tesliper.Tesliper* property), 134
- wavelen (*tesliper.glassware.arrays.Bands* property), 105
- wavelen (*tesliper.glassware.arrays.ScatteringActivities* property), 112
- wavelen (*tesliper.glassware.arrays.ScatteringData* property), 107
- wavelen (*tesliper.glassware.arrays.SpectralActivities* property), 109
- wavelen (*tesliper.glassware.arrays.SpectralData* property), 106
- wavelen (*tesliper.glassware.arrays.VibrationalActivities* property), 111
- wavelen (*tesliper.glassware.arrays.VibrationalData* property), 107
- wavelengths (*tesliper.glassware.arrays.Bands* property), 105
- wavelengths (*tesliper.glassware.arrays.ElectronicActivities* property), 114
- wavelengths (*tesliper.glassware.arrays.ElectronicData* property), 108
- wavelengths (*tesliper.glassware.arrays.ScatteringActivities* property), 112
- wavelengths (*tesliper.glassware.arrays.ScatteringData* property), 107
- wavelengths (*tesliper.glassware.arrays.SpectralActivities* property), 109
- wavelengths (*tesliper.glassware.arrays.SpectralData* property), 106
- wavelengths (*tesliper.glassware.arrays.VibrationalActivities* property), 111
- wavelengths (*tesliper.glassware.arrays.VibrationalData* property), 107
- with_traceback() (*tesliper.exceptions.InconsistentDataError* method), 74
- with_traceback() (*tesliper.exceptions.InvalidElementError* method), 74
- with_traceback() (*tesliper.exceptions.InvalidStateError* method), 74
- with_traceback() (*tesliper.exceptions.TesliperError* method), 74
- workhorse (*tesliper.extraction.gaussian_parser.GaussianParser* property), 79
- workhorse (*tesliper.extraction.parser_base.ParserBase* property), 82
- workhorse (*tesliper.extraction.spectra_parser.SpectraParser* property), 89
- write() (*tesliper.writing.csv_writer.CsvWriter* method), 145
- write() (*tesliper.writing.gif_writer.GifWriter* method), 152
- write() (*tesliper.writing.txt_writer.TxtWriter* method), 159
- write() (*tesliper.writing.writer_base.WriterBase* method), 165
- write() (*tesliper.writing.xlsx_writer.XlsxWriter* method), 169
- writer() (in module *tesliper.writing.writer_base*), 162
- WriterBase (class in *tesliper.writing.writer_base*), 162
- ## X
- x (*tesliper.glassware.spectra.SingleSpectrum* property), 127
- x (*tesliper.glassware.spectra.Spectra* property), 128
- XlsxWriter (class in *tesliper.writing.xlsx_writer*), 169
- ## Y
- y (*tesliper.glassware.spectra.SingleSpectrum* property), 127
- y (*tesliper.glassware.spectra.Spectra* property), 128