cwepr – a Python package for analysing cw-EPR data focussing on reproducibility and simple usage

—Supporting Information—

Mirjam Schröder[a], Till Biskup[b,c,*]

[a] *Leibnitz-Institut für Katalyse e.V., Albert-Einstein-Straße 29a, 18059 Rostock, Germany*
[b] *Institut für Physikalische Chemie, Albert-Ludwigs-Universität Freiburg, Albertstraße 21, 79104 Freiburg, Germany*
[c] *Current Address: Physikalische Chemie, Universität des Saarlandes, Campus B2 2, 66123 Saarbrücken, Germany*

Keywords: reproducible research, electron paramagnetic resonance spectroscopy, software, data analysis

Contents

1. Reproducibility of scientific results

Reproducibility of scientific results is fundamental to science. Nevertheless, full reproducibility is still much less widespread than commonly anticipated. It involves much more than providing the raw data and a description of how data have been processed, and even this might not be regularly available. While some disciplines have realised this and have come up with workflows and data pipelines to ensure good scientific practice, others seem still not aware of the problem and its dimension. While in a student's exam, we value the approach higher than the result, in science, we too often blindly trust the final outcome.

Here, we briefly focus on two aspects of reproducibility relevant in context of the cwepr Python package and the underlying ASpecD framework. A more detailed account is given in [1].

1.1. The two meanings of reproducibility

Reproducibility has two meanings that are relevant to the scientific method. Other scientists should be able to independently repeat an analysis and ideally come to the same result, and they should be able to understand each individual step that has been performed. Repeating an analysis may be possible given the raw data, the software in the identical version to that originally used, and the computer codes employed for the actual analysis.[1] Except of the last point, none of these aspects is trivial. For a more detailed discussion, see *e.g.* [2]. Understanding each individual step of data processing and analysis requires additionally that the programs are available in source code and that each parameter used, implicit and explicit, can be determined. Eventually, usually at least some computer codes need to be read and understood. While reading one's own code can be challenging already, reading other people's code requires both, intimate knowledge of the programming language employed and the code to be as readable and obvious as possible [3, 4]. Basically, the whole of software engineering since its origins in the 1960s [5] is an attempt to develop and provide tools that enhance readability of code. In short: Scientists are not and will usually not become software engineers, but they are often forced to write complex computer codes. To succeed,

*Corresponding author
Email address: research@till-biskup.de (Till Biskup)

[1] Here, software means both, the programming language and all libraries used, and computer code refers to the script (or similar) using the software to perform the data analysis.

they hence require applying at least some of the tools and principles of software engineering. Most meaningful data analysis is too complex to be coded by hand, regardless of the programming language used.

1.2. Scripts vs. library of functions

Generally, two approaches of data processing and analysis can be distinguished: (i) one script per analysis using only the vocabulary provided by the programming language used, or (ii) a library of more general functions that can be reused for individual analyses and that abstract from the underlying code performing the individual steps.

Some people claim that the first approach, one script per analysis, solves the problem of reproducibility and is hence superior to a library of functions. A closer look reveals this claim to be unsubstantiated. Every higher programming language provides some level of abstraction from the instructions eventually carried out by the computer. Some programming languages are better suited for scientific data analysis than others, as they provide a rich vocabulary of relevant terms. Nevertheless, one of the most important and intellectually demanding tasks of every programmer is to develop a domain-specific language for the problem at hand, providing additional and powerful abstractions. Failing to create such domain-specific language results in manually coding the same steps over and over again. This is both, time consuming and rather error-prone (typos, copy and paste mentality, loss of oversight which erros have been fixed in what version of a script). Additionally, the readability of such code (often nick-named 'spaghetti code') is clearly insufficient, preventing others from understanding it with justifiable effort and hence impairing reproducibility.

The only viable way to overcome the problems of scripts mentioned above is to develop a library of more generally applicable functions that provide abstractions [6] for individual tasks and enhance the vocabulary available in the programming language, *i.e.*, a domain-specific language. Nowadays, nobody will seriously reimplement basic linear algebra algorithms, as highly optimised and proven libraries like LAPACK [7] and BLAS [8] are available. The same is true for much more high-level libraries, *e.g.* NumPy [9] and SciPy [10] of the Python Scientific Software Stack or EasySpin [11] for simulating of EPR spectra. All these libraries provide the programmer with essential abstractions allowing to implement algorithms for particular processing and analysis steps of data. And all these libraries have in common that they follow best practices of software engineering and provide unique version numbers for reference, the latter a necessary prerequisite for reproducibility.

Furthermore, when using a library with functions for data processing and analysis, all parameters of the individual steps need to be recorded automatically, together with the version of the library, to ensure reproducibility of the results. However, in context of scientific data analysis, this is nothing the individual researchers should need to care about, as the analysis as such requires already their full attention and intellectual capacity.

The Python programming language is designed with readability in mind, and this is one reason for its widespread adoption (not only) in the scientific community. However, regardless how readable a programming language as such tries to be: every non-trivial computer program will consist of at least hundreds of lines of code. Hence, modularisation is essential, providing the reader with different levels of abstraction and allowing to focus on one small problem at a time ('separation of concerns' [12]). This is the opposite of the 'one script per analysis' approach. Developing such a library requires employing at least some of the tools and concepts of software engineering. Most essential for reproducibility are a version control system and a scheme for unique version numbers, besides code readability [3, 4].

In summary, for computer-based scientific data processing and analysis to become truly reproducible, tools need to be available that take care of automatically recording all relevant information while providing a high-level interface allowing the scientist to focus on the actual task at hand rather than its implementation in computer code. Furthermore, taken together these tools need to span the entire data processing and analysis pipeline.

2. Metadata: Info file format

As mentioned in the main text, data are represented within the cwepr package as 'datasets', *i.e.* the unit of (numerical) data and accompanying metadata. While a lot of crucial parameters are usually recorded by the vendor-specific software and stored in the respective data formats, some essential information regularly remains unaccounted for. Examples are details regarding the sample, the purpose of the measurement, and probehead and cooling system used. Whether the sample temperature is recorded depends on the setup used, and benchtop spectrometers are usually more integrated and therefore tend to record more parameters. In any case, it is the responsibility of the scientist performing the measurements to record the missing information, at best in an electronic format that can be directly read by the analysis software.

A few essential criteria for developing a file format for storing this kind of metadata: (i) easily human-*writable* while still machine-readable, (ii) plain text, as this is the only truly platform-independent and long-term accessible format, (iii) structured, guiding the user, (iv) self-contained, *i.e.* understandable without external documentation.

One such file format, termed 'info file format' has been developed by one of the authors and successfully employed over more than ten years. It is a simple structured text format that resembles the INI file format, an example is given in Listing 1. The format starts with a line providing information on the file format. This is crucial, as (i) reading only the first line of the file is sufficient for file format identification, and (ii) metadata formats change

over time, though rather slowly, due to changing needs. The remainder of the file is structured into blocks each containing a list of key-value pairs. Due to its original use with MATLAB®-based analysis routines, the format does not support any special characters beyond the first 128 characters encoded in the ASCII table.

Listing 1: Example for an info file used to store metadata that have been obtained during data acquisition.

```
cwEPR Info file - v. 0.1.4 (2020-01-21)

GENERAL
Filename:                Sa42-01
Date start:              2021-03-13
Date end:                2021-03-14
Time start:              15:50:00
Time end:                09:00:00
Operator:                John Doe
Label:                   Sa42-01
Purpose:                 First overview

SAMPLE
Name:                    Random sample
ID:                      42
Description:             film on substrate
Solvent:                 N/A
Preparation:             drop-cast
Tube:                    3.9 x 3 x 250 mm

EXPERIMENT
Type:                    field-sweep
Runs:                    1
Variable parameter:      field
Increment:               0.02 mT

SPECTROMETER
Model:                   Bruker EMX
Software:                Xenon 1.3b.5

MAGNETIC FIELD
Field probe type:        Hall
Field probe model:       xxx
Start:                   341.4 mT
Stop:                    361.5 mT
Step:                    0.02 mT
Sequence:                up
Controller:              Bruker EMX Field
    ➥ Controller
Power supply:            Bruker ER 081 (90/30)

BRIDGE
Model:                   Bruker EMX PremiumX
Controller:              Bruker EMX
Attenuation:             20 dB
Power:                   2.00 mW
Detection:               diode
Frequency counter:       Bruker
MW frequency:            9.846977 GHz
Q value:                 8300

SIGNAL CHANNEL
Model:                   Bruker EMX Signal Channel
Modulation amplifier:    Bruker EMX Modulation
    ➥ Amplifier
Accumulations:           50
Modulation frequency:    100 kHz
Modulation amplitude:    0.1 mT
Receiver gain:           80 dB
Offset:                  0.0
Conversion time:         30.00 ms
Time constant:           20.48 ms
Phase:                   0 deg

DIGITAL FILTER
Mode:                    Manual
Number of Points:        0

GONIOMETER
Start:                   0 deg
Increment:               10 deg
Number of Points:        19
Fine Tuning per Slice:   off

PROBEHEAD
Type:                    HQ
Model:                   Bruker 4119HS-W1
Coupling:                critical

TEMPERATURE
Temperature:             298 K
Controller:              N/A
Cryostat:                N/A
Cryogen:                 N/A

COMMENT
Pretty weak signal, but at least some angular
    ➥ dependence visible on first inspection.
```

The info file as such can serve as a replacement for a lab notebook entry, as it contains basically all crucial information necessary to fully analyse the data. A few fields deserve special mentioning: Providing a purpose (in the first block, `GENERAL`) can be crucial for making use of data in hindsight, as during data acquisition, particularly when systematically varying parameters, the purpose of the individual measurement is obvious, but less so later on. In the same vein, the last block (`COMMENT`) should be present in any format used to store metadata, as it is a common requirement to note down further information that does not fit into any of the pre-defined fields. Information that happens to enter the comment block on a regular base furthermore provides valuable insight into how to further develop the format and what fields to add. A full specification of the info file format is available online[2].

The cwepr Python package has full support for this file format, and as the mappings between keys in the info file format and the dataset structure of the cwepr Python package are stored separately from the actual importer, support for new versions of the info file does not require any coding, only adding the mappings to the corresponding (YAML) file contained within the cwepr package.

As some information stored in the info file is recorded by most modern spectrometer control and data acquisiion software provided by the respective vendor, there is a certain chance that the information contained in an info file deviates from that recorded automatically. Usually, the automatically obtained data are more authoritative, and thus it may be convenient to correct the info file after-

[2] https://www.till-biskup.de/en/software/info/format

wards during data processing. To this end, a special reporter has been implemented in the cwepr Python package writing info files and including all information contained in a dataset's metadata. As during import of the metadata into the dataset, the precedence of the automatically recorded values has been taken care of, this leads to more authoritative content of an info file. Make sure, however, to use this reporter *prior* to any data processing altering the metadata, including microwave frequency correction. An example is given in Listing 2.

Listing 2: Updating an info file using the `InfofileReporter`. Note that this should be done directly after importing the dataset(s) and before any processing that may change the dataset's metadata

```
- kind: report
  type: InfofileReporter
  properties:
    filename: datafilename.info
  apply_to:
  - datafilename
```

To repeat the most important aspect of metadata recording and storage: Regardless of the actual file format used, it is important to record the missing information *during data acquisition* and to provide it in a machine-readable form for the analysis software.

3. Overview of the cwepr package

This section provides a general overview of the functionality implemented within the cwepr Python package. For a more detailed user manual, the interested reader is referred to the extensive user and developer documentation available online for both, the cwepr package [13] and the ASpecD framework [14] it is based upon. For details on how ASpecD and derived packages are implemented, see Ref. [1]. Here, we briefly describe the underlying concepts.

One of the particular strengths of the cwepr Python package is its simple user interface. As the package is based on the ASpecD framework [14], it supports 'recipe-driven data analysis': The user creates a simple, structured text file containing a list of datasets to load and a list of tasks to perform on these datasets. A first example of such a recipe is provided in Listing 3. Getting served the results of 'cooking' this recipe is as simple as issuing a single command in the terminal: `serve my-recipe.yaml` assuming you have saved the listing to a file named 'my-recipe.yaml' and installed the cwepr Python package and its dependencies locally. Detailed installation instructions can be found in the documentation available online [13].

The idea behind recipe-driven data analysis is to reduce complexity and to allow the user to focus on the actual science, namely data processing and analysis. Usually, we have an idea which tasks we want to perform on a dataset, and we will even have ideas which parameters we would need for the individual tasks. All this enters the recipe in a highly structured and obvious way. While the details of the individual tasks will be discussed below, the recipe presented in Listing 3 should be pretty self-explanatory (not only) for a spectroscopist used to dealing with cw-EPR data.

Listing 3: Example of a recipe used for recipe-driven data analysis within the cwepr Python package. Here, a list of datasets is followed by a list of tasks. The user needs no programming skills, but can fully focus on the tasks to be performed. 'Cooking' this recipe is a matter of issuing a single command on the terminal.

```
format:
  type: ASpecD recipe
  version: '0.2'

settings:
  default_package: cwepr

datasets:
  - /path/to/first/dataset
  - /path/to/second/dataset

tasks:
  - kind: processing
    type: FrequencyCorrection
    properties:
      parameters:
        frequency: 9.5
  - kind: processing
    type: BaselineCorrection
  - kind: singleplot
    type: SinglePlotter1D
    properties:
      filename:
        - first-dataset.pdf
        - second-dataset.pdf
```

Often, tasks should only be applied to a subset of the datasets loaded. Therefore, the list of datasets a task should operate on can be given explicitly. Otherwise, the task will operate on all datasets. For details, see the examples in section 6.

But what about reproducibility? Upon 'cooking' the recipe presented in Listing 3 and serving its results, a history will be written detailing each individual step. For a first impression, cf. Listing 4. As this is a valid recipe in itself, it serves a dual purpose: (i) it contains all information necessary to fully reproduce the analysis, including the list and version of all relevant Python packages and all explicit and implicit parameters, and (ii) it can be used to automatically rerun the analysis.

3.1. Data import and supported formats

Data are represented within the cwepr package as 'datasets', *i.e.* the unit of (numerical) data and accompanying metadata. As mentioned above, a lot of crucial parameters are usually recorded by the vendor-specific software and stored in the respective data formats. However, some

Listing 4: Excerpt of the history automatically written by serving the example recipe displayed in Listing 3. Notable are the automatically added blocks at the top containing information on the time of execution as well as the system used, including version numbers of all relevant Python packages. Furthermore, as the baseline correction results in different coefficients for each of the two datasets, those are separately presented for each individual dataset.

```
info:
  start: '2021-11-26T09:03:52'
  end: '2021-11-26T09:03:57'
system_info:
  python:
    version: "3.7.3 ..."
  packages:
    aspecd: 0.6.4
    jinja2: 3.0.2
    matplotlib: 3.4.3
    numpy: 1.21.3
    scipy: 1.7.1
    oyaml: '1.0'
    asdf: 2.8.1
    bibrecord: 0.1.0
    cwepr: 0.2.0
  platform: Linux-4.19.0-18-...
  user:
    login: johndoe
format:
  type: ASpecD recipe
  version: '0.2'
settings:
  default_package: cwepr
  # ...
datasets:
- /path/to/first/dataset
- /path/to/second/dataset
tasks:
- kind: processing
  type: BaselineCorrection
  properties:
    parameters:
      kind: polynomial
      order: 0
      coefficients:
      - -0.06901404916763308
      fit_area:
      - 10
      - 10
      axis: 0
  apply_to:
  - /path/to/first/dataset
- kind: processing
  type: BaselineCorrection
  properties:
    parameters:
      kind: polynomial
      order: 0
      coefficients:
      - -0.07042420227050784
      fit_area:
      # ... remainder same as above
  apply_to:
  - /path/to/second/dataset
# ...
```

essential information regularly remains unaccounted for, such as details regarding the sample, the purpose of the measurement, and probehead and cooling system used. Whether the sample temperature is recorded depends on the setup used, and benchtop spectrometers are typically more integrated and therefore tend to record more parameters. In any case, it is the responsibility of the scientist performing the measurements to record the missing information, at best in an electronic format that can be read directly by the analysis software. One such file format, termed 'info file format' has been developed by one of the authors and successfully employed over more than ten years. For details see section 2. Regardless of the actual file format used, it is important to record the missing information *during data acquisition* and to provide it in a machine-readable form for the analysis software.

In terms of vendor file formats, the cwepr Python package currently supports the different Bruker file formats for the old ESP and EMX spectrometer series as well as the newer BES3T format. Additionally, Magnettech XML files can be read, and as a last resort, bare text files (CSV and alike) can be imported. The latter, however, usually lack any metadata. Thanks to the highly modular architecture of the cwepr package, adding importers for additional file formats is simple and straight-forward. Details can be found in the package documentation available online [13]. What is much more relevant for the user of the package: File formats will be auto-detected and the respective importer chosen.[3] If you ever need to control in more detail which importer is used, and *e.g.* in case of CSV data, provide additional parameters, this is possible as well directly within the recipe.

All further steps operating on datasets, such as data processing and analysis, plotting, report generation, and export, are summarised under the term 'tasks' and defined in the `tasks:` block of a recipe, respectively. For each task, one can define which datasets it should be applied to and provide a wealth of additional necessary and optional parameters. A bit more details will be provided below for each kind of task.

3.2. Data processing

Data processing is a necessary prerequisite for data analysis, and therefore separate from the latter. The main effort in spectroscopy is usually not recording the raw data, but processing and analysing those data in order to answer the questions that triggered the measurements in the first place. The difference between processing and analysis in context of the cwepr Python package: Processing steps operate on datasets and always return (modified) datasets, while analysis steps operate on datasets and extract information, but may return everything from a scalar value to a full (calculated) dataset, depending on the type of analysis step.

[3] In software engineering terms, a factory pattern [15] is used here.

The processing currently available within the cwepr Python package can be categorised further: corrections, simple algebra, normalisation, handling two-dimensional datasets, and working with multiple datasets.

Corrections contain magnetic field calibration, microwave frequency correction, and baseline correction. Magnetic field calibration usually consists of measuring a reference sample with known g value, extracting the measured magnetic field value for the line and calculating a scalar offset that can afterwards be applied to the magnetic field axis of the dataset of the unknown sample. Technically speaking, magnetic field calibration is hence a series of an analyis step (field offset calculation from measuring the standard sample) and a processing step (applying the offset to the field axis of the actual data). Microwave frequency correction is a linear algebraic operation essentially shifting the magnetic field axis. It is a prerequisite for any meaningful comparison of different measurements, as any single measurement will be recorded at an at least slightly different microwave frequency to any other. Baseline correction is usually performed in terms of fitting a polynomial of n-th degree to (parts of) the data and afterwards subtracting the polynomial. A zeroth-order baseline correction (offset correction) can (and should) always safely be applied. First-order baselines (linear drifts) are frequently encountered, too. Higher-order polynomials should rarely be used. Often, it is more sensible to record a background spectrum and subtract this afterwards.

Simple algebra involves applying a scalar value by addition, subtraction, multiplication, or division to the intensity values of a recorded spectrum. Typical scenarios for multiplying the intensity values of a cw-EPR spectrum with a scalar factor are comparing spectra resulting from a single species to those known to originate from multiple species, or different (known) concentrations. Adding and subtracting scalars can of course be used conveniently to separate traces within a plot, but are probably less important in terms of actual analysis. Furthermore, dedicated plotters are available for the stacked display of spectra. Particulary with older spectrometers, multiple scans are usually recorded in an additive fashion, meaning that the resulting intensities are the sums of the intensities of the individual scans rather than their average. The same is true with the receiver gain setting. Therefore, in those cases, for meaningful (semi-)quantitative comparison of different measurements, data need to be corrected for the same number of scans and the same receiver gain setting by dividing the signal intensity by the appropriate scalar value.

Normalising data to some common characteristics is a prerequisite for comparing datasets among each other. A number of normalisations are common to nearly every kind of data: normalisation to maximum, minimum, amplitude, and area. Which kind of normalisation works best depends highly on the given situation and the intent. Comparing differences in line shapes, such as broadening or narrowing, usually requires scaling to maximum, minimum, or amplitude. The meaning of normalising to area is less straightforward for cw-EPR spectra, as usually, first-derivative line shapes are encountered. Given appropriate measurement conditions (no saturation, no line broadening due to overmodulation, proper phasing), the cw-EPR signal intensity should be proportional to the number of spins in the active volume of the probehead. Therefore, with all crucial experimental parameters directly affecting the signal strength being equal (microwave power, modulation amplitude), normalising to same area should be the most straight-forward way of comparing two spectra in a meaningful way. However, bare in mind that this is only valid for absorptive (zeroth-derivative or zeroth harmonic) spectra. While it is the user's responsibility to decide which type of normalisation is best, the cwepr Python package makes it pretty simple to apply. Even better: Normalisations can be applied to a given range, and the range provided in axis values (*e.g.*, mT) as well as indices (*e.g.*, indices 35–42), whatever fits better in the given context. An example involving real data will be given in the next section, showcasing the elegance and power of the user interface particularly when it comes to more complicated data processing.

Two-dimensional datasets are encountered in cw-EPR spectroscopy in different flavours. Depending on the spectrometer and measurement software used, microwave power and modulation amplitude variations as well as goniometer sweeps are either recorded as separate datasets (one dataset per trace) or as two-dimensional dataset. Furthermore, kinetics measurements that can be used as well to record independent scans and to prevent averaging within the software are other use-cases for two-dimensional datasets. In all these cases, typical operations on two-dimensional datasets are slice extraction, averaging, and projection along one axis. Slice extraction and averaging allow again to provide the slice or average in terms of indices or axis values. In case of the latter, values are interpolated to the nearest neighbour on the grid spanned by the axis values.

The last category of processing steps mentioned here involves operating on multiple datasets in terms of 'dataset algebra', *i.e.* adding, subtracting, and averaging multiple datasets. As mentioned already, subtracting a separately recorded background signal from the signal of a sample is a typical use-case. Due to microwave frequency correction, each dataset has its individual magnetic field axis. Therefore, to add, subtract, or average multiple datasets, first, a common axis range needs to be extracted and the data points interpolated to a common grid. Here, metadata accompanying the numerical data and stored in the datasets come in quite handy, as they allow for automatically deciding whether two axes are compatible. As sometimes, axis labels will differ in their names but still be compatible, you are free to explicitly override this check. In any case, dataset algebra is always a two-step process: (i) extracting a common range for a series of datasets, and (ii) performing the actual dataset algebra, such as adding, subtracting or averaging the data.

3.3. Data analysis

In nearly all cases, data analysis needs to be preceded with data processing. While processing steps can often be automated to a large extend and are rather generally applicable, data analysis is usually much more focussed on individual types of measurements and the actual questions at hand. As a reminder: In context of the cwepr Python package, processing and analysis steps differ technically in their results. While both operate on datasets, processing steps always return an (altered) dataset, while analysis steps may return everything from a scalar to a complete (calculated) dataset that can be operated on in the same way as any other dataset, including graphical representations and further processing and analysis.

Most important, the analysis steps provided by the cwepr package are generally meant as basic building blocks to be used in arbitrarily complex overall analyses, consisting of large lists of processing and analysis steps, usually interspersed with plotting steps for graphical feedback. Eventually, the possibilities are only limited by the user's creativity and imagination. This is the focus and power of the cwepr package: freeing the users from dealing with the implementation details of each individual processing and analysis step and allowing them to creatively combine the different tasks in a fully transparent manner. To make it even better: Everything is fully reproducible, and repeatedly replaying and systematically modifying a complicated analysis is as simple as modifying a highly descriptive text file. For more complicated and time-consuming tasks, the analysis can even be run fully unattended and in parallel.

Concrete analysis steps include extracting basic characteristics (minimum, maximum, amplitude, and area) and statistics (mean, median, standard deviation, and variance), peak finding, signal-to-noise estimation, polynomial fits and linear regression with fixed intercept. Besides these rather general tasks, a series of analysis steps specific for cw-EPR spectroscopy is available as well, such as field calibration (obtaining a field offset value given the measurement of a standard sample with known g value), linewidth detection (peak-to-peak, fwhm), and analysis of systematically varied modulation amplitude and microwave power. Results of analysis steps returning either scalar values or lists of values for an individual dataset can be aggregated into a (calculated) dataset. Thus, one could *e.g.* integrate a series of datasets, calculate the area under the respective curve (under ideal conditions proportional to the concentration of paramagnetic species) and plot the results as a function of the datasets (or any value characteristic for the series of datasets) or return a table of values. Both would make for a decent (semi-)quantitative analysis of cw-EPR data.

Eventually, fitting spectral simulations to the recorded data can be regarded as analysis steps as well. Usually, they are the only way to extract parameters such as g or hyperfine splitting values from cw-EPR data with sufficient accuracy. As mentioned already, these tasks are not and will not be part of the cwepr Python package, but analysis steps provide the interface to packages dedicated to this purpose that are currently being developed [16, 17].

3.4. Data representation: plotting

Graphically representing the results of processing (and analysis) steps is of paramount importance in data analysis, not only as means of finally presenting the results, but for ensuring that the individual tasks performed on the data give sensible results. Furthermore, given that datasets contain both, (numerical) data as well as accompanying metadata, correct axes labels can (and will) be created fully automatically. By using the Matplotlib library [18] of the scientific Python stack, publication-quality figures are readily available.

Three general types of plotters are available: plotters for single datasets and for multiple datasets, and composite plotters consisting of other plotters arranged in a grid within one figure. For individual datasets, 1D and 2D plotters are available, for multiple datasets, only 1D plots are possible. Specifically for cw-EPR spectroscopy, plotters for representing angular-dependent measurements (*i.e.*, goniometer sweeps) and the results of power saturation analysis have been implemented.

3.5. Report generation: accessing information

While plotters are an excellent way to obtain publication-quality figures without hassle, and the recipe history automatically created contains all information necessary to fully reproduce and replay the tasks, there is a lot more of information contained in the datasets and potentially the recipes as well. The latter is even more true in light of recipes supporting adding comments to each individual task, as well as figure captions to plotters. Hence, being able to automatically create well-formatted reports using pre-defined templates opens an entirely new dimension in terms of comparing different datasets and workflows, besides presenting the results of the research.

Key to report generation is using a template engine thus separating the file format of the final report from the logic used for filling in the contents in the template from the data source (usually a dataset). Template engines have flourished due to the needs of modern web development, and are here used to create well-formatted reports containing all information available from within a dataset. Currently, the most mature templates supported by the cwepr Python package, by means of the underlying ASpecD framework, are LaTeX templates presenting the contents of single datasets, including details for each task and a full list of figures. Even more, given a LaTeX installation, one can compile the report into a PDF file straight from within a recipe. An example of such a report is provided in section 6.

While plots are already quite complex due to the amount of parameters one can tweak, reports provide yet another level of complexity. Automatically generating not only the

figures for a manuscript or thesis, but as well the captions, and having them included in the main text, is only one possible application. Besides that, generating reports of (complex) routine processing and analysis workflows for individual datasets provides means to easily compare the results. Never underestimate the power of well-formatted and uniform reports allowing to focus on the differences rather than having to find the parameters to compare in different places. This again shows the design philosophy as well as the potential of the cwepr Python package for analysing data, as it allows to focus on the important aspects while fully transparently automating all the routine work and ensuring full reproducibility.

3.6. Data export and supported formats

As most data processing and analysis tasks are not too time-consuming and can always be repeated starting from the raw data, saving the resulting processed datasets may not be an immediate need. Nevertheless, for more complex tasks this may change. Therefore, two particular formats for datasets are supported by the cwepr package by means of the underlying ASpecD framework: the 'Advanced Scientific Data Format' (ASDF) [19] and a format particularly developed for the ASpecD framework and termed 'ASpecD Dataset Format' (ADF). Both are fully self-contained, *i.e.*, come with their own specification, and are thus platform-independent, relying on well-developed standards. Furthermore, for a maximum of interoperability, data can be exported to plain text. Note, however, that in this case usually all metadata accompanying the data will be lost, rendering this a choice of last resort. As with data import, writing own exporters is both, straight-forward and simple. Details can be found in the documentation available online [13]. Some general aspects of extending the cwepr package are provided in the next section.

4. Extending the cwepr package

The cwepr Python package is based on the ASpecD framework providing all functionality necessary for full reproducibility as well as for recipe-driven data analysis. Due to the modular nature of the ASpecD framework, it is comparably easy and straight-forward to extend the cwepr package. Basically, the cwepr package can be thought of as an extension of the ASpecD framework focussing on the particular needs of cw-EPR spectroscopy.

Due to using the paradigma of object-oriented programming [20], developers can focus on implementing the actual functionality by simply inheriting from the correct class of the ASpecD framework, rather than caring about the mechanisms ensuring reproducibility. An example for a class of the cwepr package is given in Listing 5. While this is an analysis step, implementing a processing step would be similar. The actual functionality is implemented in the method `_perform_task`. Here, additionally the applicability of the analysis step is checked for using the method `applicable`. Additionally, if users can provide parameters, these parameters need to be checked. For this purpose, the method `_sanitise_parameters` can be implemented and will be called automatically. The parameter `description` set in the constructor provides a short description of the analysis step that can, *e.g.*, be included in a report detailing all the individual processing and analysis steps performed on a dataset.

In a similar vein, processing steps, plotters, importers, exporters, and alike can be implemented. Depending on the algorithms used, the implementations may be more involved than that shown in Listing 5, probably spanning several (private) methods. Of course, as with every framework, extending the cwepr package requires to be familiar with the concepts of the underlying ASpecD framework. For further details, see the documentation of the cwepr package [13] and the ASpecD framework [14] available online as well as the source code of both packages.

When implementing extensions to the cwepr package, ideally, unit tests should be provided as well. The ASpecD framework is developed fully test-driven (*i.e.*, test-first), and we aim at providing a good test coverage for the cwepr package as well.

Listing 5: Stripped-down example of the peak-to-peak linewidth analysis step implementation contained in the cwepr package. The docstrings have been shortened to focus on the important aspects. Thanks to inheriting from the ASpecD framework, developers can focus on implementing the actual functionality in the method `_perform_task`. Here, additionally the applicability of the analysis step is checked for using the method `applicable`. Additionally, if user-provided parameters need to be checked, the method `_sanitise_parameters` can be implemented and gets called automatically. For further details, see the documentation of the cwepr package [13] and the ASpecD framework [14] available online.

```
class LinewidthPeakToPeak(aspecd.analysis.SingleAnalysisStep):
    """Peak to peak linewidth in derivative spectrum."""

    def __init__(self):
        super().__init__()
        self.description = "Determine peak-to-peak linewidth"

    @staticmethod
    def applicable(dataset):
        """
        Check whether analysis step is applicable to the given dataset.

        Line width detection can only be applied to 1D datasets.
        """
        return dataset.data.data.ndim == 1

    def _perform_task(self):
        index_max = np.argmax(self.dataset.data.data)
        index_min = np.argmin(self.dataset.data.data)
        linewidth = abs(self.dataset.data.axes[0].values[index_min] -
                        self.dataset.data.axes[0].values[index_max])
        self.result = linewidth
```

5. Recipe-driven data analysis vs. 'plain' Python

Basically, recipe-driven data analysis can be thought of a special type of user interface to the cwepr Python package (and the underlying ASpecD framework). The normal user of such package has a clear idea how to process and analyse data, but is not necessarily interested in (or capable of) actually programming a lot. Furthermore, reproducible science requires the history of each and every processing and analysis step to be recorded and stored in a way that can be used and understood long after the steps have been carried out (decades rather than weeks or months).

From the user's perspective, all that is required is a human-writable file format and a list of datasets followed by a list of tasks to be performed on these datasets. For each task, the user will want to provide all necessary parameters. Eventually, the user is providing the metadata of the data analysis, such as Listing 6.

Of course, everything that can be done using a recipe and recipe-driven data analysis can be coded in 'plain' Python as well. Actually, the cwepr package is a regular Python package and provides a well-documented API [13]. Therefore, it can be easy integrated into other downstream pipelines such as simulation and fitting. To make the point, Listing 7 provides the implementation of the recipe presented in Listing 6 in Python using all the high-level functionality (*i.e.*, the domain-specific language for

Listing 6: Example of a simple recipe for recipe-driven data analysis within the cwepr Python package. Here, two basic processing steps are applied to two datasets and the results afterwards plotted individually for each dataset. For an implementation of the same functionality in 'plain' Python, cf. Listing 7

```
format:
  type: ASpecD recipe
  version: '0.2'

settings:
  default_package: cwepr

datasets:
  - /path/to/first/dataset
  - /path/to/second/dataset

tasks:
  - kind: processing
    type: FrequencyCorrection
    properties:
      parameters:
        frequency: 9.5
  - kind: processing
    type: BaselineCorrection
  - kind: singleplot
    type: SinglePlotter1D
    properties:
      filename:
        - first-dataset.pdf
        - second-dataset.pdf
```

Listing 7: Python code performing the same tasks as the recipe in Listing 6. Only in this rather simplistic case, a single loop could be used. Furthermore, despite the intrinsic readability of Python code, the recipe is much easier to understand and does not require any programming skills.

```
import aspecd
import cwepr

dataset_filenames = ['/path/to/first/dataset',
                     '/path/to/second/dataset']
figure_filenames = ['first-dataset.pdf',
                    'second-dataset.pdf']

importer_factory = cwepr.io.ImporterFactory()
frequency_correction =
    cwepr.processing.FrequencyCorrection()
frequency_correction.parameters =
    {"frequency": 9.5}
baseline_subtraction =
    aspecd.processing.BaselineCorrection()
plotter = aspecd.plotting.SinglePlotter1D()

for idx, source in enumerate(dataset_filenames):
    dataset = cwepr.dataset.ExperimentalDataset()
    importer =
        importer_factory.get_importer(source)
    dataset.import_from(importer)
    dataset.process(frequency_correction)
    dataset.process(baseline_subtraction)
    plot = dataset.plot(plotter)
    saver = aspecd.plotting.Saver()
    saver.filename = figure_filenames[idx]
    plot.save(saver)
```

analysing cw-EPR data) provided by the cwepr package and the ASpecD framework it is based upon.

It is not the lines of code (both are pretty much the same) but the readability that is different for a recipe and the Python code. This is not to say that Python code is hard to read, the opposite is true. Nevertheless, there are differences in readability between any programming language and a descriptive structural language such as YAML. Furthermore, using a single loop in the Python code as in the given example was only possible in this simple case. Applying some tasks to a selection of datasets – while easy to implement in a recipe – is much more involved in actual code.

Furthermore, the Python implementation presented in Listing 7 does not provide a history similar to the recipe history shown in the main text. Still, on a per-dataset level, a history is written and accessible using, *e.g.*, the reporting functionality of the cwepr package. Nevertheless, any tasks depending and operating on multiple datasets at once will not show up in the history of the individual datasets. Therefore, recipes and their automatically written history provide means for fully reproducible analysis of multiple datasets not easily available otherwise.

6. Recipes for the examples

The examples given in the main text necessarily show only essential parts of the recipes. Here, we document the entire recipes that have been used to create the figures shown in the manuscript.

6.1. Compare a series of recorded spectra

Listing 8: Recipe used to compare cw-EPR data of a series of samples recorded under nearly identical conditions. For a meaningful comparison of such a series of data, you need to perform a few processing steps that can be applied to nearly every situation. This is a baseline correction (polynomial, zeroth order) and a frequency correction. In this particular case, a normalisation (amplitude) has been carried out for easier comparing the differences in line shape. In a second step, dat a have been smoothed using a Savitzky-Golay filter, and original and smoothed data displayed in a common figure using a `CompositePlotter`.

```
format:
  type: ASpecD recipe
  version: '0.2'
settings:
  default_package: cwepr
  autosave_plots: false
datasets:
- Sa732-01
- Sa734-01
- Sa736-01
- Sa738-01
tasks:
- kind: processing
  type: BaselineCorrection
- kind: processing
  type: FrequencyCorrection
  properties:
    parameters:
      frequency: 9.48
- kind: processing
  type: Normalisation
  properties:
    parameters:
      kind: amplitude
- kind: multiplot
  type: MultiPlotter1D
  properties:
    parameters:
      show_legend: false
      show_zero_lines: true
      tight_layout: true
      g-axis: false
    properties:
      axes:
        xlim: [336, 340]
        yticks: []
  apply_to:
  - Sa732-01
  - Sa734-01
  - Sa736-01
  - Sa738-01
  result: unfiltered
- kind: processing
  type: Filtering
  properties:
    parameters:
      type: savgol
      window_length: 501
      order: 5
```

```
  apply_to:
  - Sa732-01
  - Sa734-01
  - Sa736-01
  - Sa738-01
  result:
  - Sa732-01-filtered
  - Sa734-01-filtered
  - Sa736-01-filtered
  - Sa738-01-filtered
- kind: multiplot
  type: MultiPlotter1D
  properties:
    parameters:
      show_legend: false
      show_zero_lines: true
      tight_layout: true
      g-axis: false
    properties:
      axes:
        xlim: [336, 340]
        yticks: []
  apply_to:
  - Sa732-01-filtered
  - Sa734-01-filtered
  - Sa736-01-filtered
  - Sa738-01-filtered
  result: filtered
- kind: compositeplot
  type: CompositePlotter
  properties:
    plotter:
    - unfiltered
    - filtered
    parameters:
      show_legend: false
      show_zero_lines: true
      tight_layout: false
    properties:
      figure:
        size:
        - 6.0
        - 6.0
        dpi: 100.0
    filename: comparison-unfiltered-filtered.pdf
    grid_dimensions: [2,1]
    subplot_locations:
    - [0, 0, 1, 1]
    - [1, 0, 1, 1]
```

6.2. Power saturation analysis

Listing 9: Recipe used to perform a complete power saturation analysis. In a first step, the cw-EPR signal amplitude and square root of the microwave power are returned as calculated dataset, afterwards a linear regression performed over the first few points. The results of both are graphically represented together, using a special plotter adding a second axis with the actual microwave power values on top.

```
format:
  type: ASpecD recipe
  version: '0.2'
settings:
  default_package: cwepr
  autosave_plots: false
datasets:
- BDPA-2DFieldPower
tasks:
- kind: processing
  type: BaselineCorrection
- kind: singleanalysis
  type: AmplitudeVsPower
  apply_to:
  - BDPA-2DFieldPower
  result: power_sweep_analysis
- kind: singleanalysis
  type: PolynomialFitOnData
  properties:
    parameters:
      order: 1
      points: 5
      return_type: dataset
  apply_to:
  - power_sweep_analysis
  result: fit
  comment: Linear fit covering the first five
      ➥ data points.
- kind: multiplot
  type: PowerSweepAnalysisPlotter
  properties:
    properties:
      drawings:
        - marker: '*'
        - color: red
      grid:
        show: true
        axis: both
      axes:
        title: Overview
        ylabel: '$EPR\ amplitude$'
        yticklabels: []
  apply_to:
  - power_sweep_analysis
  - fit
  result: overview
- kind: multiplot
  type: PowerSweepAnalysisPlotter
  properties:
    properties:
      drawings:
        - marker: '*'
        - color: red
      grid:
        show: true
        axis: both
      axes:
        title: Detailed view
        xlim: [0, 1.65]
        ylim: [0, 70]
        ylabel: '$EPR\ amplitude$'
        yticklabels: []
  apply_to:
  - power_sweep_analysis
  - fit
  result: details
- kind: compositeplot
  type: CompositePlotter
  properties:
    plotter:
    - overview
    - details
    filename: power_sweep_analysis.pdf
    caption:
      title: Power saturation analysis.
      text: >
        The left panel provides an overview of
            ➥ the entire measurement,
        while the right panel provides a detailed
            ➥  view of the first points,
```

```
        showing that 1 mW of microwave power
            ➥ already starts to saturate the
        cw-EPR signal.
    grid_dimensions: [1, 2]
    subplot_locations:
    - [0, 0, 1, 1]
    - [0, 1, 1, 1]
```

6.3. Subtracting a recorded background signal

Listing 10: Recipe used to subtract a recorded background signal from cw-EPR spectra of a series of actual samples. To subtract the background signal, the spectra need to be scaled in a meaningful way. Here, a particular field range is used for scaling all spectra to the same amplitude. The axis range is conveniently provided in axis values (here: mT).

```
format:
  type: ASpecD recipe
  version: '0.2'
settings:
  default_package: cwepr
  autosave_plots: true
directories:
  datasets_source: data_raw/
datasets:
  - source: complex1
    id: compound1
    label: Complex 1

  - source: complex2
    id: compound2
    label: Complex 2

  - source: complex3
    id: compound3
    label: Complex 3

  - source: tube2
    id: background
    label: 'Background'

tasks:
- kind: processing
  type: FrequencyCorrection
  properties:
    parameters:
      frequency: 9.84
  comment: Frequency correction

- kind: processing
  type: BaselineCorrection
  properties:
    parameters:
      order: 0
      percentage: [10,10]
  comment: Baseline correction

- kind: multiplot
  type: MultiPlotter1D
  properties:
    filename: baseline_freq_corr.png
    properties:
      figure:
        dpi: 600
        title: Baseline and frequency corrected
      axes:
        xlim: [250,400]
        yticks: []
        yticklabels: []
        ylabel: '$Intensity$ / a.u.'
      legend:
        loc: 'lower left'
    parameters:
      show_legend: True
      tight_layout: True
      g-axis: True
  comment: Plot after baseline and frequency
      ➥ correction

- kind: processing
  type: Normalisation
  properties:
    parameters:
      kind: amplitude
      range: [357, 375]  # in mT
      range_unit: axis
  apply_to:
  comment: Normalisation to range of background
      ➥ signal

- kind: multiplot
  type: MultiPlotter1D
  properties:
    filename: range_norm.png
    properties:
      figure:
        dpi: 300
        title: Normalised to background peak
      axes:
        xlim: [250,400]
        ylim: [-3, 1]
        yticks: []
        yticklabels: []
        ylabel: '$Intensity$ / a.u.'
      legend:
        loc: 'lower left'
    parameters:
      show_legend: True
      tight_layout: True
      g-axis: True
  comment: Plot after normalisation to background
      ➥  signal

- kind: multiprocessing
  type: CommonRangeExtraction
  comment: Range extraction in preparation for
      ➥ subtracting background in DatasetAlgebra

- kind: processing
  type: DatasetAlgebra
  properties:
    parameters:
      kind: minus
      dataset: background
  apply_to:
    - compound3
    - compound1
    - compound2
  comment: Subtract background spectrum

- kind: multiplot
  type: MultiPlotter1D
  properties:
    filename: background_subtracted.png
    properties:
      figure:
        dpi: 300
        title: Subtracted background
```

```
      axes:
        xlim: [250,400]
        ylim: [-3, 1]
        yticks: []
        yticklabels: []
        ylabel: '$Intensity$ / a.u.'
      legend:
        loc: 'lower left'
    parameters:
      show_legend: True
      tight_layout: True
      g-axis: True
  apply_to:
    - compound1
    - compound2
    - compound3
  comment: Figrue of background subtracted
      ➥ spectra.

- kind: processing
  type: Normalisation
  properties:
    parameters:
      kind: amplitude
  comment: Normalisation on complete spectrum

- kind: singleplot
  type: SinglePlotter1D
  properties:
    properties:
      figure:
        dpi: 300
      axes:
        ylabel: '$Intensity$ / a.u.'
      drawing:
        color: dodgerblue  # all Matplotlib
            ➥ colors possible, also in Hex-Code
    caption:
      title: >
        Background corrected spectrum of complex
            ➥ 1.
      text: >
        Resulting spectrum after the background
            ➥ correction.
    parameters:
      show_legend: True
      tight: x
      tight_layout: True
      g-axis: True
  apply_to:
    - compound1
  comment: Singleplotter for background spectrum
      ➥ with generic filename.

- kind: multiplot
  type: MultiPlotter1D
  properties:
    filename: final.png
    properties:
      figure:
        dpi: 300
        title: Final figure
      axes:
        xlim: [250,400]
        yticks: []
        yticklabels: []
        ylabel: '$Intensity$ / a.u.'
      legend:
        loc: 'lower left'
      caption:
        title: >
          Background corrected spectra of all
              ➥ compounds.
        text: >
          After background subtraction, all
              ➥ spectra were again normalised to
              ➥ the
          full amplitude.
    parameters:
      show_legend: True
      tight_layout: True
      g-axis: True
  apply_to:
    - compound1
    - compound2
    - compound3
  comment: Background corrected spectra, complete
      ➥  view

- kind: export
  type: TxtExporter
  properties:
    target:
      - data/compound1.txt
  apply_to:
    - compound1
  comment:  >
    Data exported to a txt file with magnetic
        ➥ field values in the first and
    intensity values in the second column. The
        ➥ file is stored in the
    directory "data" which first has to be
        ➥ created.

- kind: report
  type: LaTeXReporter
  properties:
    template: dataset.tex
    filename:
      - report_compound1.tex
  compile: true
  apply_to:
    - compound1
  comment: >
    Aggregation of all steps on the background
        ➥ dataset to give it a readable
    form.
```

The PDF output of the report task follows on the next pages. Besides covering details of all the tasks performed on the dataset and including the created plots as figures, the dataset report contains all the metadata stored within the dataset (and read from an info file upon import). Furthermore, the last section of the report lists all packages used, together with their versions.

Dataset report

— *mirjam*, 2021-12-10 19:27:18

1 Overview

Source: `complex1`
Label: Complex 1

This is an experimental dataset with 7 processing steps, 0 analyses, 1 annotation, 1 representation, and 8 total tasks. For details, see below. Information on how this report has been generated and how to cite the underlying software are given at the end.

2 Processing steps

In total, 7 processing steps have been carried out:

1. Correct magnetic field axis for given frequency
2. Correct baseline of dataset
3. Normalise data
4. Interpolate data of dataset
5. Extract common data range of several datasets
6. Perform algebra using two datasets
7. Normalise data

For details of the individual processing steps, see below.

2.1 Correct magnetic field axis for given frequency

This processing step is not undoable.

Parameters	
frequency	9.84

Comment: Frequency correction

2.2 Correct baseline of dataset

This processing step is undoable.

Parameters	
kind	polynomial
order	0
coefficients	[-1.20234879]
fit_area	[10, 10]
axis	0
percentage	[10, 10]

Comment: Baseline correction

2.3 Normalise data

This processing step is undoable.

Parameters	
kind	amplitude
range	[357, 375]
range_unit	axis
noise_range	None
noise_range_unit	percentage

Comment: Normalisation to range of background signal

2.4 Interpolate data of dataset

This processing step is undoable.

Parameters	
range	[[241.48692686 422.15552723]]
npoints	[8166]
unit	axis

2.5 Extract common data range of several datasets

This processing step is undoable.

Parameters	
ignore_units	False
common_range	[[241.48692685786594, 422.15552723018635]]
npoints	[8166]

2.6 Perform algebra using two datasets

This processing step is not undoable.

Parameters	
dataset	background
kind	minus

Comment: Subtract background spectrum

2.7 Normalise data

This processing step is undoable.

Parameters	
kind	amplitude
range	None
range_unit	index
noise_range	None
noise_range_unit	percentage

Comment: Normalisation on complete spectrum

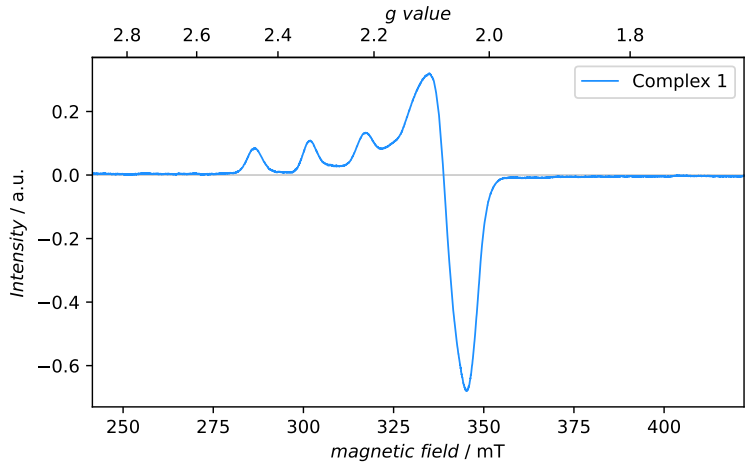

Figure 1: **Background corrected spectrum of complex 1.** Resulting spectrum after the background correction.

3 Annotations

In total, 1 annotation has been created:

1. Comment

3.1 Comment

Temperature shifted slightly to higher values during measurement.

4 Representations

In total, 1 representation has been created:

1. 1D plotting step for single dataset (Fig. 1)

5 Metadata

Please note: Due to better compatibility with LaTeX, the parameter names listed below have been changed from snake case (using the underscore "_" as word separator) to camel case (medial capitals) with respect to their names in Python.

5.1 Measurement

label Complex 1

start 2021-09-26 16:21:00

end 2021-09-26 16:41:00

purpose Signal at low temperature

operator Jim Knopf

5.2 Sample

description Complex in solvent, frozen

solvent buffer

preparation John Doe

tube glass capillary sealed with clay

name Complex 1

id 10

5.3 Temperature

cryostat Cryogenic CF VTC

cryogen He

temperature 130.0 K

controller Lake Shore 350

5.4 Experiment

type field seewp

runs 1

variableParameter Field

increment 0.22

harmonic 1.0

5.5 Spectrometer

model Bruker ELEXSYS

software Bruker XEPR

5.6 Magnetic field

start 239.95 mT

stop 420.02801782009027 mT

sweepWidth 180.1 mT

stepWidth 0.0 G

points 8192.0

fieldProbeType Hall

fieldProbeModel xxx

sequence Up

controller Bruker EMX

powerSupply Bruker ER083 (200/60)

5.7 Microwave Bridge

model Bruker EMX premiumX

controller Bruker EMX

attenuation 20.0 dB

power 0.001997 mW

detection diode

frequencyCounter Bruker

mwFrequency 9.84 GHz

qValue 8900

5.8 Signal channel

model Bruker ELEXSYS

modulationAmplifier Bruker ELEXSYS

accumulations 2.0

modulationFrequency 100.0 kHz

modulationAmplitude 0.4 mT

receiverGain 49.0 dB

conversionTime 0.08192 ms

timeConstant 0.02048 ms

phase 0.0 deg

5.9 Probehead

model Bruker ER4118X-MD5

type dielectric

coupling critical

Colophon

This report has been generated using the cwepr package that is based on the ASpecD framework. If you use it in your research, please cite cwepr (doi:10.5281/zenodo.4896687, details: `https://docs.cwepr.de/`), ASpecD (doi:10.5281/zenodo.4717937, details: `https://docs.aspecd.de/`) and the other packages accordingly.

Packages: aspecd (0.6.4), numpy (1.21.4), matplotlib (3.4.3), bibrecord (0.1.0), jinja2 (3.0.3), oyaml (1.0), asdf (2.8.1), scipy (1.7.2), cwepr (0.2.0).

Python version: 3.9.8 (main, Nov 12 2021, 14:53:26) [Clang 10.0.1 (clang-1001.0.46.4)]

Platform: macOS-10.14.6 [...]

6.4. Represent angular-dependent measurements

Listing 11: Recipe used to import and plot an angular-dependent measurement with the `GoniometerSweepPlotter`. This plotter automatically extracts the two slices at 0° and 180° as a sanity check as those spectra should be the same.

```
format:
  type: ASpecD recipe
  version: '0.2'
settings:
  default_package: cwepr
directories:
  datasets_source: data_raw/
datasets:
  - example1/RotationPattern-01
tasks:
- kind: processing
  type: BaselineCorrection
- kind: singleplot
  type: GoniometerSweepPlotter
  properties:
    properties:
      figure:
        dpi: 300
      axes:
        xlim: [349, 353]
    filename: output1.png
```

6.5. Comparison of data recorded at X and Q band

Listing 12: Recipe used to compare cw-EPR data of the same sample recorded at X and Q band. For a meaningful comparison, you need to convert the magnetic field axis to a g axis. In this case, the magnetic field was not calibrated, but for both measurements, an external standard (Li:LiF) had been recorded. Therefore, first a frequency and field correction needs to be carried out for both sets of measurements.

```
format:
  type: ASpecD recipe
  version: '0.2'
settings:
  default_package: cwepr
datasets:
- source: Sample-X
  label: X-band
- LiLiF-X
- source: Sample-Q
  label: Q-band
- LiLiF-Q
tasks:
- kind: processing
  type: BaselineCorrection
- kind: processing
  type: FrequencyCorrection
  properties:
    parameters:
      frequency: 9.5
  apply_to:
  - Sample-X
  - LiLiF-X
  comment: Frequency correction for X band
- kind: processing
  type: FrequencyCorrection
  properties:
    parameters:
      frequency: 34.0
  apply_to:
  - Sample-Q
  - LiLiF-Q
  comment: Frequency correction for Q band
- kind: singleanalysis
  type: FieldCalibration
  properties:
    parameters:
      standard: LiLiF
  apply_to:
  - LiLiF-X
  result: field-offset-X
  comment: Field offset for X band
- kind: singleanalysis
  type: FieldCalibration
  properties:
    parameters:
      standard: LiLiF
  apply_to:
  - LiLiF-Q
  result: field-offset-Q
  comment: Field offset for Q band
- kind: processing
  type: FieldCorrection
  properties:
    parameters:
      offset: field-offset-X
  apply_to:
  - Sample-X
  comment: Field correction for X band
- kind: processing
  type: FieldCorrection
  properties:
    parameters:
      offset: field-offset-Q
  apply_to:
  - Sample-Q
  - LiLiF-Q
  comment: Field correction for Q band
- kind: processing
  type: GAxisCreation
  apply_to:
  - Sample-X
  - Sample-Q
  result:
  - Sample-X-gaxis
  - Sample-Q-gaxis
  comment: Convert magnetic field axis to g axis
- kind: processing
  type: Normalisation
  properties:
    parameters:
      kind: amplitude
  apply_to:
  - Sample-X-gaxis
  - Sample-Q-gaxis
  comment: Normalise to amplitude for easier
    ➥ comparison
- kind: multiplot
  type: MultiPlotter1DStacked
  properties:
    parameters:
      show_legend: true
      show_zero_lines: false
      tight_layout: true
      g-axis: false
      offset: null
    properties:
      axes:
        xlim: [2.007, 1.999]
```

```
    grid:
      show: true
      axis: x
  filename: x-q-comparison.pdf
  caption:
    title: >
      Comparison of the cw-EPR spectra of the
          ➥ same substance recorded at
      X and Q band.
    text: >
      The splitting observed at Q band and not
          ➥ visible at X band
      can be attributed to g anisotropy.
          ➥ Furthermore, only conversion
      of the magnetic field axis to a g axis
          ➥ allows for directly
      comparing the spectra obtained at
          ➥ different fields and frequencies.
apply_to:
- Sample-Q-gaxis
- Sample-X-gaxis
```

7. Graphical frontend for recipes

Although the YAML file format is remarkably simple to write by hand, the mere number of options that can be set for certain tasks (in particular plotting tasks) can be daunting. Furthermore, a tool helping with automatically creating a recipe from its building blocks dramatically reduces the chance of problems due to wrong formatting, especially with respect to indentation. As this is a problem not special for the cwepr package, but common to all packages derived from the ASpecD framework, we are currently developing a graphical frontend in form of a WebUI for this purpose, based on the Python Flask framework. A preview of the already working prototype is shown in Fig. 1. This prototype has been used in part to create the recipes showcased in the main text and documented above.

The graphical frontend allows to load datasets, add and remove them from the recipe, preview, edit, and add tasks to the recipe, and finally cook the recipe and serve the results. Furthermore, the messages usually printed to the command line when cooking a recipe are displayed as well. Taken together, we anticipate the WebUI to make working with the cwepr package even simpler, particularly for non-spectroscopists. In the future, the WebUI will probably even be able to directly display the results of graphical representations, making recipe-driven data analysis more interactive and helping with exploratory analyses.

Using web technologies makes the graphical frontend intrinsically platform-independent, and working with a web framework such as Flask is much simpler than implementing a GUI with one of the usual GUI frameworks (GTK, Qt). Furthermore, GUI programming is a rather complex topic on its own, particularly when focussing on modular code following best practices of software engineering.

The WebUI will be made available open-source and free of charge as a Python package under comparable conditions as the ASpecD framework and the cwepr package. The frontend is designed to run on your local computer, not on a web server accessible from the outside. Although the latter would be technically possible, this immediately raises security concerns that are currently not being dealt with, such as access control and data protection. However, we will provide detailed documentation on how to locally install and deploy the WebUI and even run it inside a Docker container for better isolation.

References

[1] J. Popp, T. Biskup, ASpecD: A modular framework for the analysis of spectroscopic data focussing on reproducibility and good scientific practice, ChemRxiv (2021). doi:10.26434/chemrxiv-2021-6jt1l.

[2] V. Stodden, F. Leisch, R. D. Peng (Eds.), Implementing Reproducible Research, CRC press, Boca Raton, Florida, 2014.

[3] B. W. Kernighan, P. J. Plauger, The Elements of Programming Style, second ed., McGraw-Hill, New York, 1978.

[4] R. C. Martin, Clean Code. A Handbook of Agile Software Craftmanship, Prentice Hall, Upper Saddle River, New Jersey, 2008.

[5] P. Naur, B. Randell (Eds.), Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, NATO, Scientific Affairs Division, Brussels, 1969.

[6] E. W. Dijkstra, The humble programmer, Commun. ACM 15 (1972) 859–865. doi:10.1145/355604.361591.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, third ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[8] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, An updated set of basic linear algebra subprograms (BLAS), ACM Trans. Math. Software 28 (2002) 135–151. doi:10.1145/567806.567807.

[9] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (2020) 357–362. doi:10.1038/s41586-020-2649-2.

[10] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 Contributors, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nat. Methods 17 (2020) 261–272. doi:10.1038/s41592-019-0686-2.

[11] S. Stoll, A. Schweiger, EasySpin, a comprehensive software package for spectral simulation and analysis in EPR, J. Magn. Reson. 178 (2006) 42–55. doi:10.1016/j.jmr.2005.08.013.

[12] E. W. Dijkstra, EWD447: On the role of scientific thought, Springer-Verlag, New York, 1982, pp. 60–66.

[13] M. Schröder, T. Biskup, cwepr Python package, 2021. URL: https://docs.cwepr.de/. doi:10.5281/zenodo.4896687.

[14] T. Biskup, ASpecD framework, 2021. URL: https://docs.aspecd.de/. doi:10.5281/zenodo.4717937.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston, 1995.

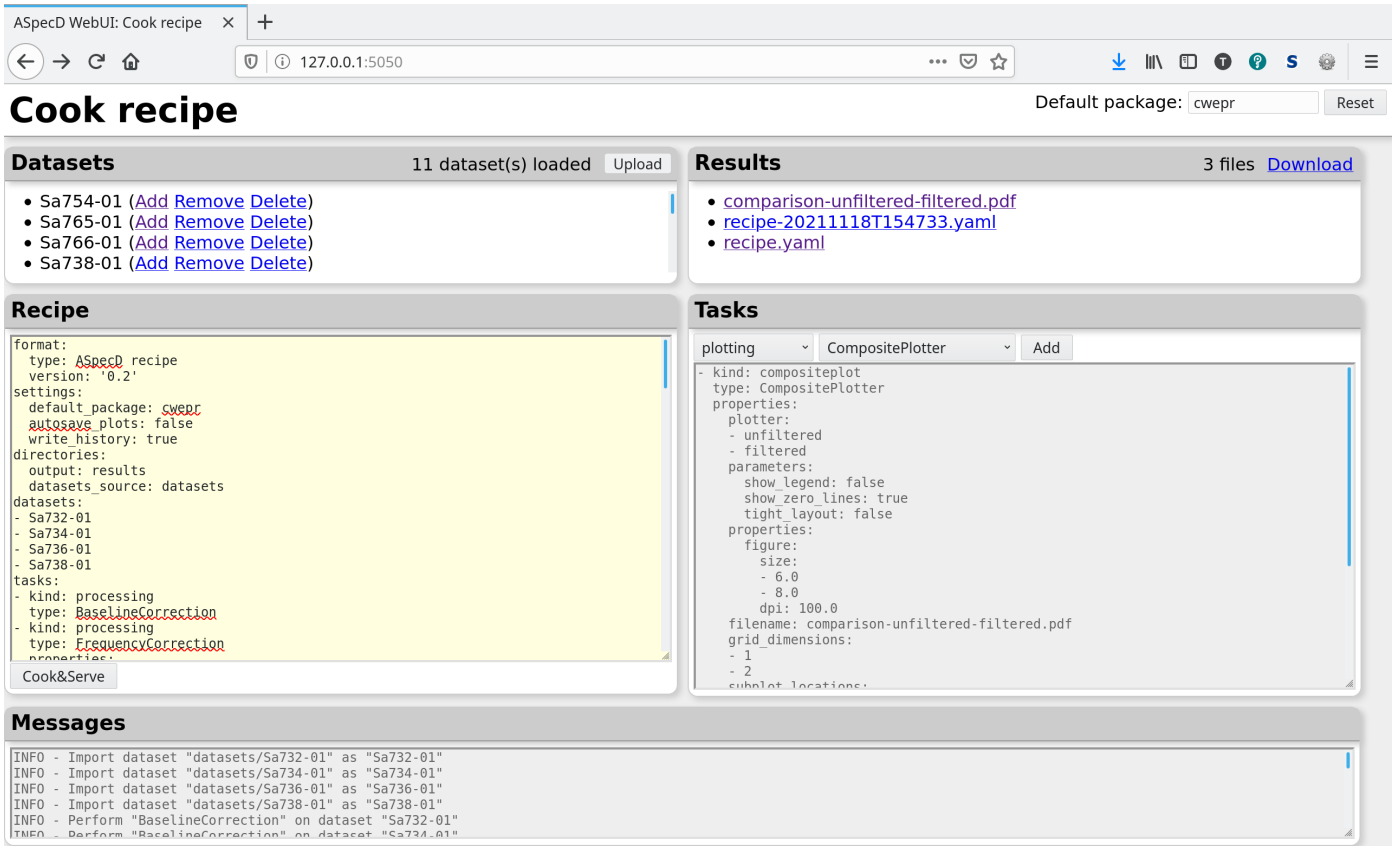

Figure 1: First impression of a WebUI for comfortably working with recipes. The program runs locally and allows for conveniently editing recipes, previewing the structure for the different tasks, and 'cooking' the recipe. The tasks preview in particular liberates the user from having to copy&paste the YAML structure from the documentation and provides a quick overview of the available kinds of tasks and the parameters that can be set.

[16] T. Biskup, SpinPy Python package, 2021. URL: `https://docs.spinpy.de/`.

[17] T. Biskup, FitPy Python package, 2021. URL: `https://docs.fitpy.de/`.

[18] J. D. Hunter, Matplotlib: a 2D graphics environment, Comput. Sci. Eng. 9 (2007) 90–95. doi:`10.1109/MCSE.2007.55`.

[19] P. Greenfield, M. Droettboom, E. Bray, ASDF: a new data format for astronomy, Astron. Comput. 12 (2015) 240–251. doi:`10.1016/j.ascom.2015.06.004`.

[20] B. Meyer, Object-Oriented Software Construction, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.