ImpedanceFitter Documentation

Release 2.0.2

Julius Zimmermann, Leonard Thiele

Aug 31, 2021

CONTENTS:

CHAPTER

ONE

EXAMPLES

These are a few examples of how ImpedanceFitter could be used.

1.1 Generating and using a model

The general idea of ImpedanceFitter is to generate equivalent-circuit models from the basic elements. However, there are some more complex circuits and models that are pre-implemented. One example is the Randles circuit.

The Randles circuit can be formulated in ImpedanceFitter as:

```
model = 'R_s + parallel(R_ct + W, C)'
```

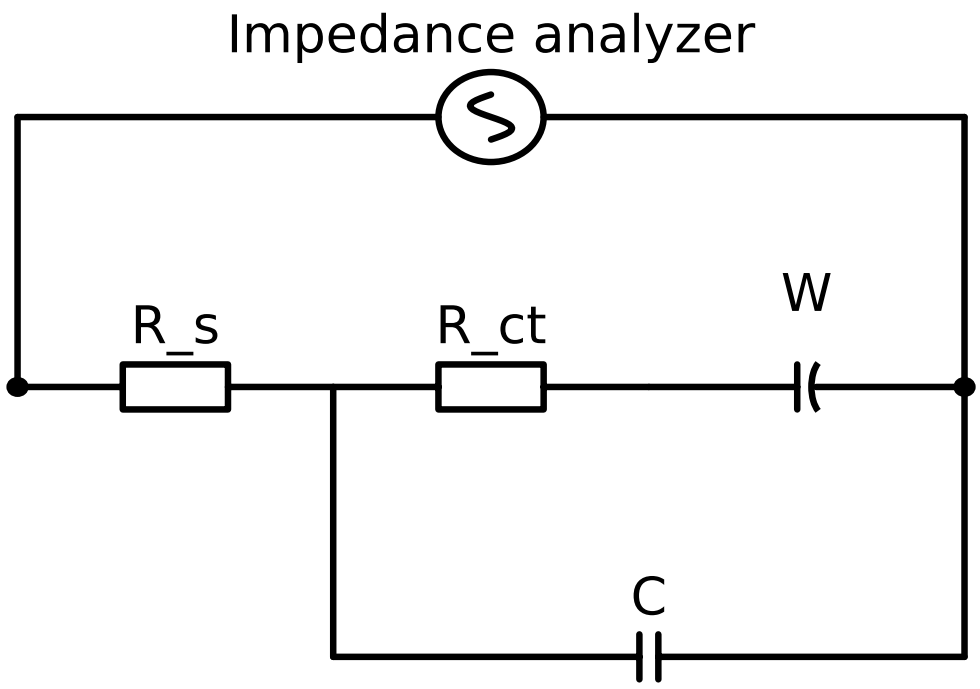
The model consists of the basic elements R, W, and C with respective suffix.

It can be drawn by

```
impedancefitter.draw_scheme(model)
```

Note that the interface to draw the equivalent circuit is quite simplistic and might not be correct for very complicated circuits!

For this circuit it yields



The impedance for a list of frequencies can the be computed by calling

```
frequencies = numpy.logspace(0, 8)
Rct = 100.
Rs = 20.
Aw = 300.
C0 = 25e-6

 model = 'R_s + parallel(R_ct + W, C)'
lmfit_model = impedancefitter.get_equivalent_circuit_model(model)
Z = lmfit_model.eval(omega=2. * numpy.pi * frequencies,
                     ct_R=Rct, s_R=Rs,
                     C=C0, Aw=Aw)
```

The same circuit has been pre-implemented and is available as

```
model = 'Randles'
lmfit_model = impedancefitter.get_equivalent_circuit_model(model)
Z = lmfit_model.eval(omega=2. * numpy.pi * frequencies,
                      Rct=Rct, Rs=Rs,
                      C0=C0, Aw=Aw)
```

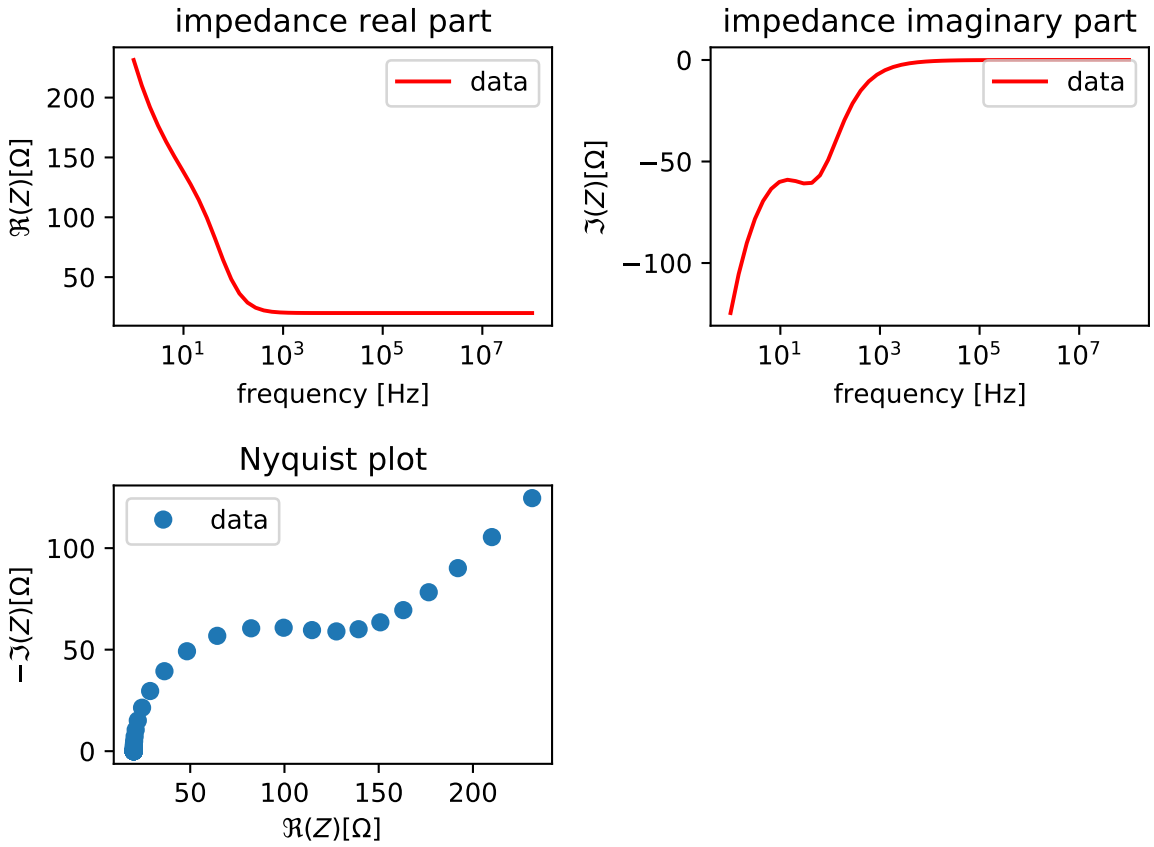
Note: LMFIT names parameters with a prefix. When writing equivalent circuits, it is usual to use suffixes. Hence, the models here are formulated using suffixes but the parameters need to be named with prefixes. Then, if the model is *R_ct*, the respective parameter is *ct_R*.

Warning: Make sure to add suffixes to each element or circuit if there any elements with the same name! LMFIT does not (yet) support models, where you have for example *ct_R* and *R*. As shown here you need *s_R* AND (!) *ct_R*.

The computed impedance can also be visualized

```
impedancefitter.plot_impedance(2. * numpy.pi * frequencies, Z)
```

The real and imaginary part are shown together with the Nyquist plot



1.1.1 See Also

`examples/Randles/randles_model.py`.

1.2 Validating the data

1.2.1 LinKK test

Before fitting the experimental data to an equivalent circuit model, you can make sure that the data are valid. The data are valid if the real and imaginary part are related through the Kramers-Kronig (KK) relations. The LinKK test [Schoenleber2014] permits to validate the data by fitting the data to a KK compliant model. A less automated version of this test is also used in proprietary software (see, e.g. this application note).

Here, it is summarized how the test works in ImpedanceFitter.

Assume the model discussed in [Schoenleber2014]. It can be represented in ImpedanceFitter by

```
model = 'R_s1 + parallel(C_s3, R_s2)  + parallel(R_s4, W_s5)'
```

After having generated artifical data stored in *test.csv*, we can initialize the fitter.

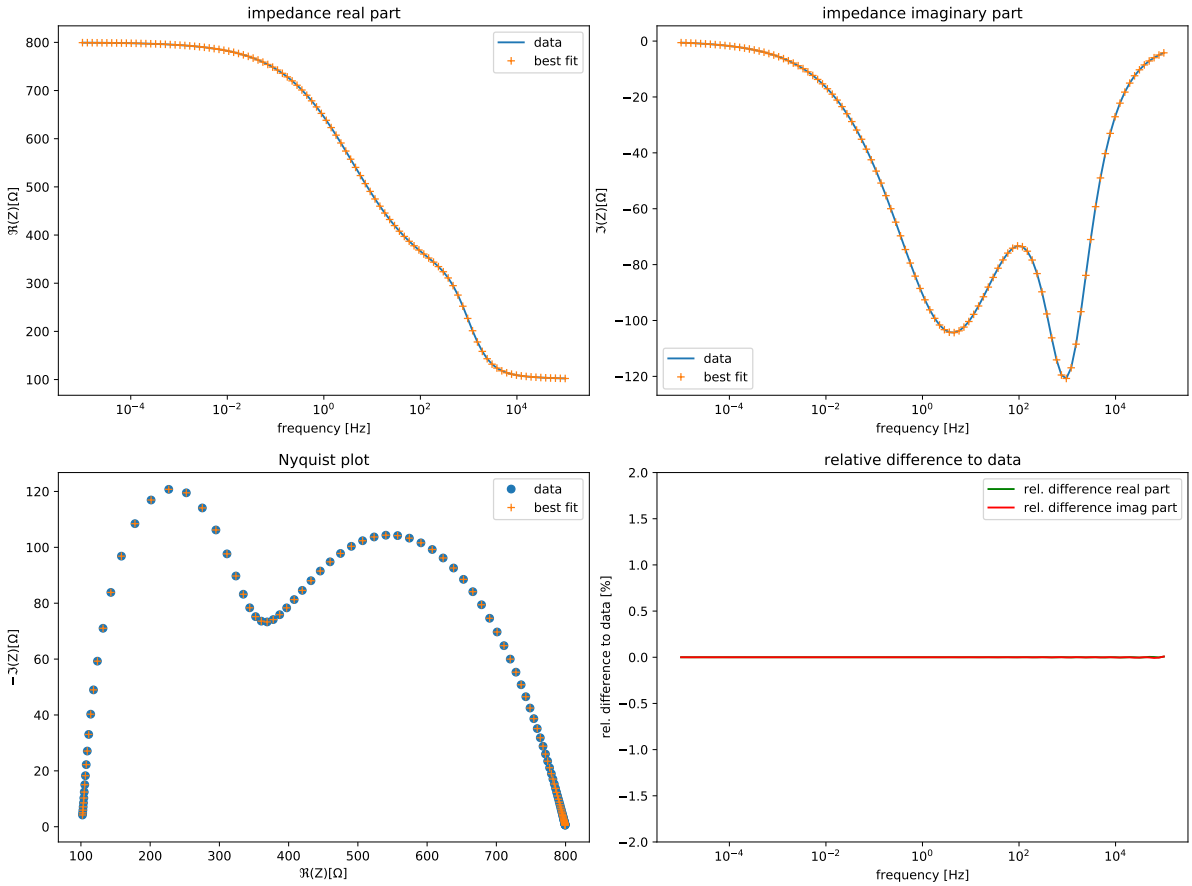
```
fitter = impedancefitter.Fitter('CSV')
```

Then, the LinKK test can be performed for all data sets by running

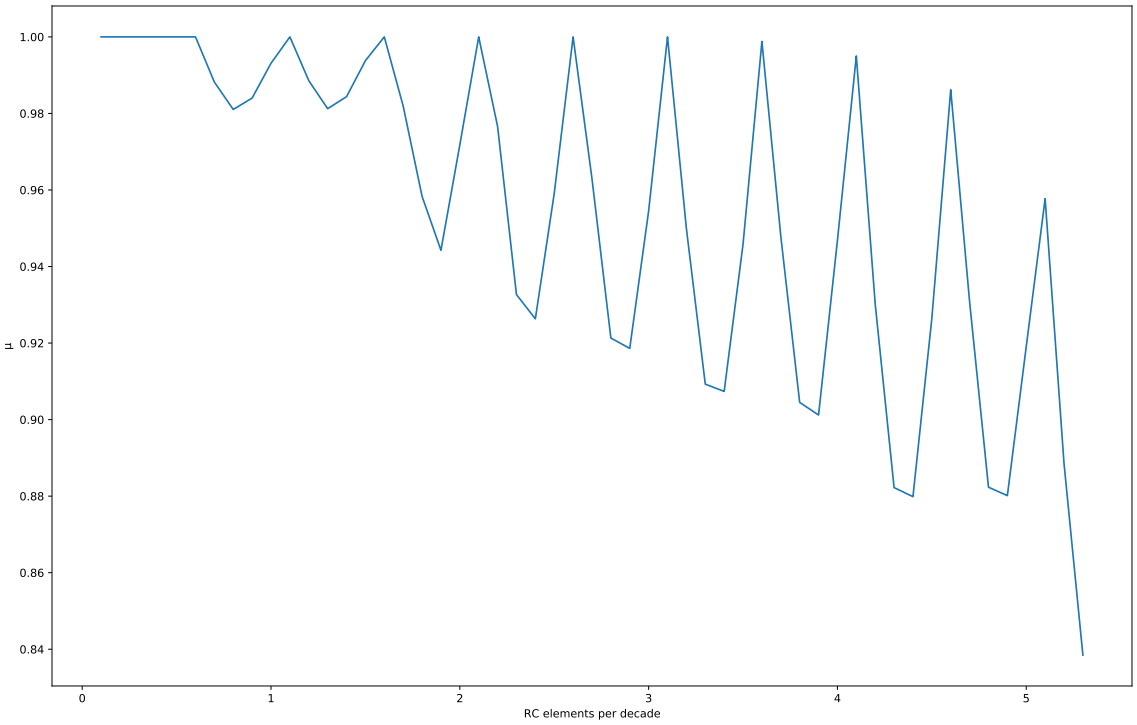
```
results, mus, residuals = fitter.linkk_test()
```

results contains the fit results as a dictionary. *mus* is a dictionary with all μ values for an increasing number of RC-elements used in the LinKK-test. *residuals* is a dictionary containing all residuals during the least-squares fit.

The result of the fit looks like this:



The parameter μ decays with an increased number of RC elements as described in [Schoenleber2014]. It is used to detect overfitting. The threshold for overfitting is set to *0.85* but can be manually adjusted.



In this example, all residuals are very small (as expected for artifical data). If the relative difference exceeds 1% or if there is a drift in the residuals, concerns about the validity of the experimental data could be raised. If you observe sinusoidal oscillations in your residuals, increase the number of RC-elements either manually or by decreasing the threshold to values below *0.85*.

If there is an inductive or capacitive element present, it can be benefitial to add an extra capacitance or inductance to the circuit. This can be done by

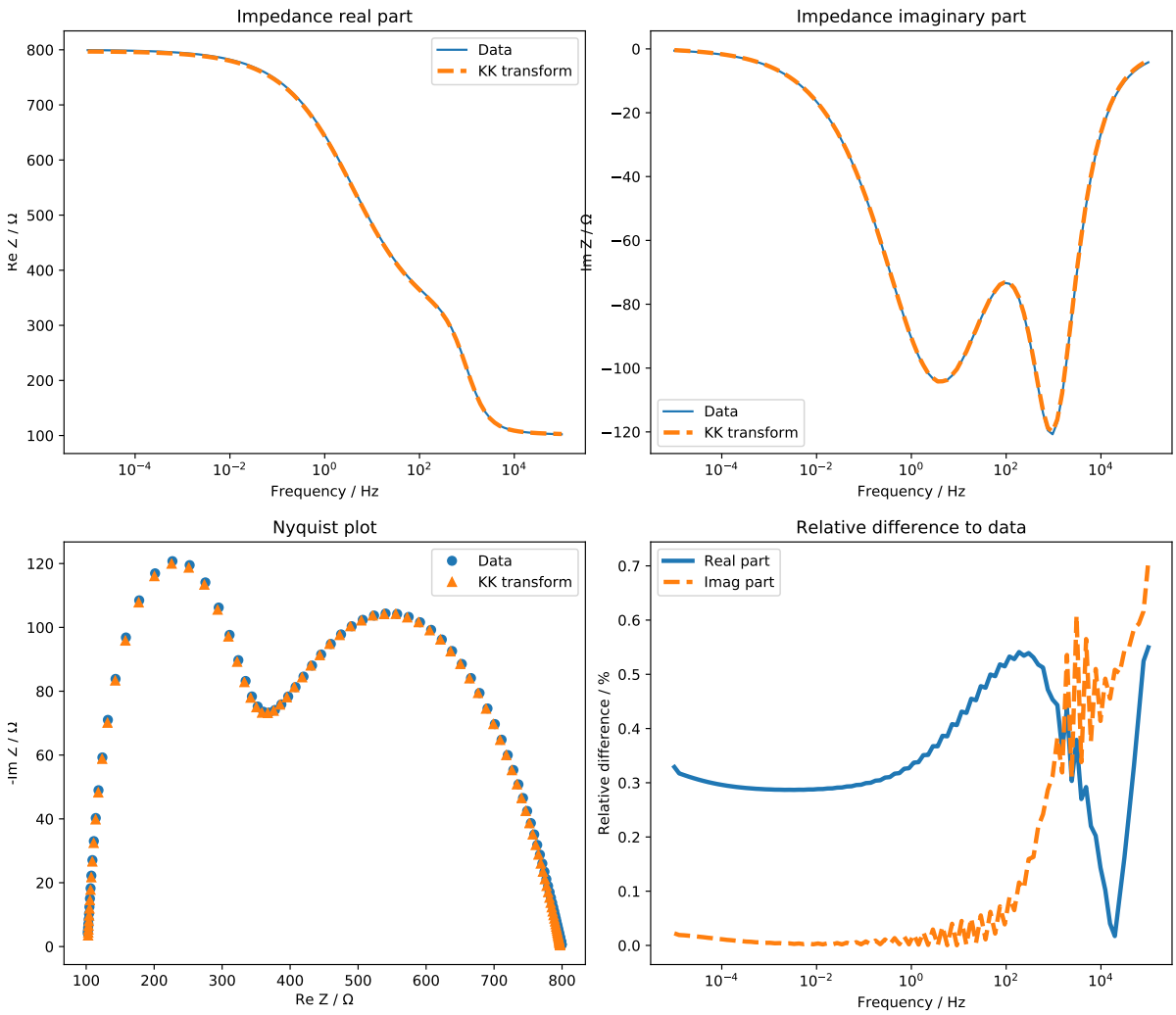
```
results, mus = fitter.linkk_test(capacitance=True)
results, mus = fitter.linkk_test(inductance=True)
results, mus = fitter.linkk_test(capacitance=True, inductance=True)
```

Especially if you observe large residuals at high frequencies, an inductive element should be added.

1.2.2 Numerical Integration

The Kramers-Kronig relations are integral transforms. These integrals can be evaluated numerically [Urquidi1990]. This functionality is available through the function *KK_integral_transform*. For the abovementioned model it can be done by

```
ZKK = impedancefitter.KK_integral_transform(2. * numpy.pi * frequencies, Z)
# add the high frequency impedance to the real part
ZKK += Z[-1].real
# plot the impedance and show the residual
impedancefitter.plot_impedance(2. * numpy.pi * frequencies, Z, Z_fit=ZKK,
                               residual="absolute", labels=["Data", "KK transform", "
↪"])
```



The result indicates that the data fulfil the KK relations. However, the error is not as small as with the LinKK test (mostly due to numerical accuracy of the integration scheme).

1.2.3 See Also

`examples/LinKK/linkk.py`. `examples/LinKK/linkk_cap.py`. `examples/LinKK/linkk_ind.py`. `examples/LinKK/linkk_ind_cap.py`. `examples/KK/kk.py`.

impedancefitter.fitter.Fitter.linkk_test()

1.3 Fitting experimental data

Impedance spectroscopy data can be processed using *impedancefitter.fitter.Fitter*.

Currently, a few fileformats can be used. They are summarized in *impedancefitter.utils.available_file_format()*.

In this example, artificial data will be generated and fitted.

We want to fit a user-defined circuit, which reads

```
model = 'R_s + parallel(R_ct + W, C)'
```

First the data is generated using the following parameters

```
frequencies = numpy.logspace(0, 8)
Rct = 100.
Rs = 20.
Aw = 300.
C0 = 25e-6
```

Then the model is defined and data generated and exported

```
lmfit_model = impedancefitter.get_equivalent_circuit_model(model)
Z = lmfit_model.eval(omega=2. * numpy.pi * frequencies,
                     ct_R=Rct, s_R=Rs,
                     C=C0, Aw=Aw)
data = {'freq': frequencies, 'real': Z.real,
        'imag': Z.imag}
# write data to csv file
df = pandas.DataFrame(data=data)
df.to_csv('test.csv', index=False)
```

The fitter is initialized with verbose output. Also, the fit results will be plotted immediately.

```
fitter = impedancefitter.Fitter('CSV', LogLevel='DEBUG', show=True)
os.remove('test.csv')
```

We use the Randles circuit that corresponds to the custom circuit model. The initial guess is passed in a dictionary with minimal information.

```
model = 'Randles'
parameters = {'Rct': {'value': 3. * Rct},
              'Rs': {'value': 0.5 * Rs},
              'C0': {'value': 0.1 * C0},
              'Aw': {'value': 1.2 * Aw}}
```

Then the fit is simply run by

```
fitter.run(model, parameters=parameters)
```

```
circuit: Randles


Created composite model <lmfit.Model: Model(Z_randles)>


Using provided parameter dictionary.
```

```
Setting values for parameters ['Rct', 'Rs', 'Aw', 'C0']

Parameters: {'Rct': {'value': 300.0}, 'Rs': {'value': 10.0}, 'C0': {'value':␣
↪2.5e-06}, 'Aw': {'value': 360.0}}

Rct needs to be positive. Changed your min value to 0.

Rs needs to be positive. Changed your min value to 0.

Aw needs to be positive. Changed your min value to 0.

C0 needs to be positive. Changed your min value to 0.

Number of data sets:1

Going to fit

################################

fit data to Z_randles model

################################

#########
Fitting started
#########

[[Model]]
    Model(Z_randles)
[[Fit Statistics]]
    # fitting method   = least_squares
    # function evals   = 9
    # data points      = 100
    # variables        = 4
    chi-square         = 3.4118e-21
    reduced chi-square = 3.5540e-23
    Akaike info crit   = -5165.22119
    Bayesian info crit = -5154.80051
[[Variables]]
    Rct:  100.000000 +/- 3.0601e-12 (0.00%) (init = 300)
    Rs:   20.0000000 +/- 9.7854e-13 (0.00%) (init = 10)
    Aw:   300.000000 +/- 8.7380e-12 (0.00%) (init = 360)
    C0:   2.5000e-05 +/- 1.3551e-18 (0.00%) (init = 2.5e-06)
[[Correlations]] (unreported correlations are < 0.100)
    C(Rct, Aw) = -0.656
    C(Rct, C0) =  0.354
    C(Rs, C0)  =  0.348
    C(Aw, C0)  = -0.341
```

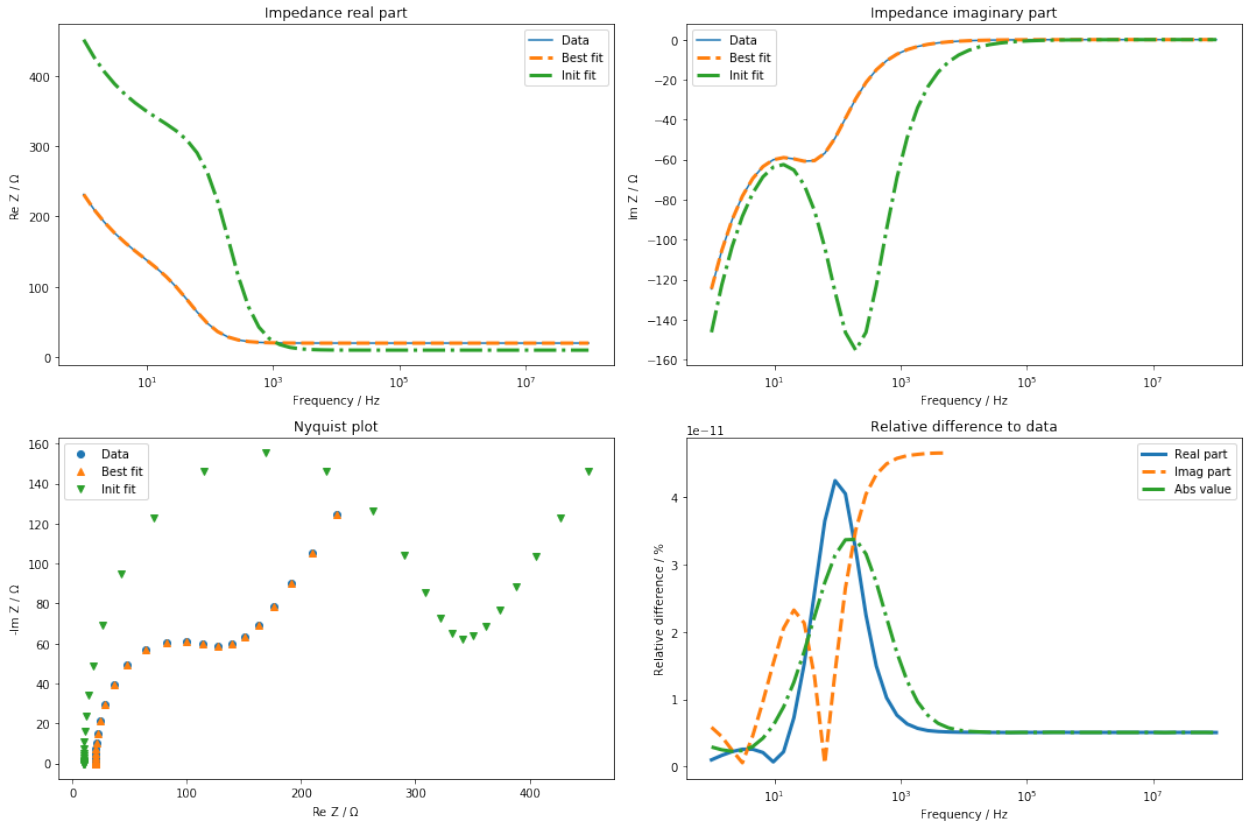
```
    C(Rct, Rs) = -0.183

Solver message: `gtol` termination condition is satisfied.

Fit successful
```

Initial and best fit can be plotted together to improve on the initial parameter guess

```
fitter.plot_initial_best_fit()
```



Here, the initial guess was passed in a dictionary with minimal information. One could also specify bounds or fix a parameter.

For example, if *Rct* was restricted to be between 50 and 500 and *C0* was known and thus fixed, the parameters would read

```
parameters = {'Rct': {'value': 3. * Rct,
                      'min': 50,
                      'max': 500},
              'Rs': {'value': 0.5 * Rs},
              'C0': {'value': C0, 'vary': False},
              'Aw': {'value': 1.2 * Aw}}
```

1.3.1 See Also

`examples/Randles/randles_data.py`.

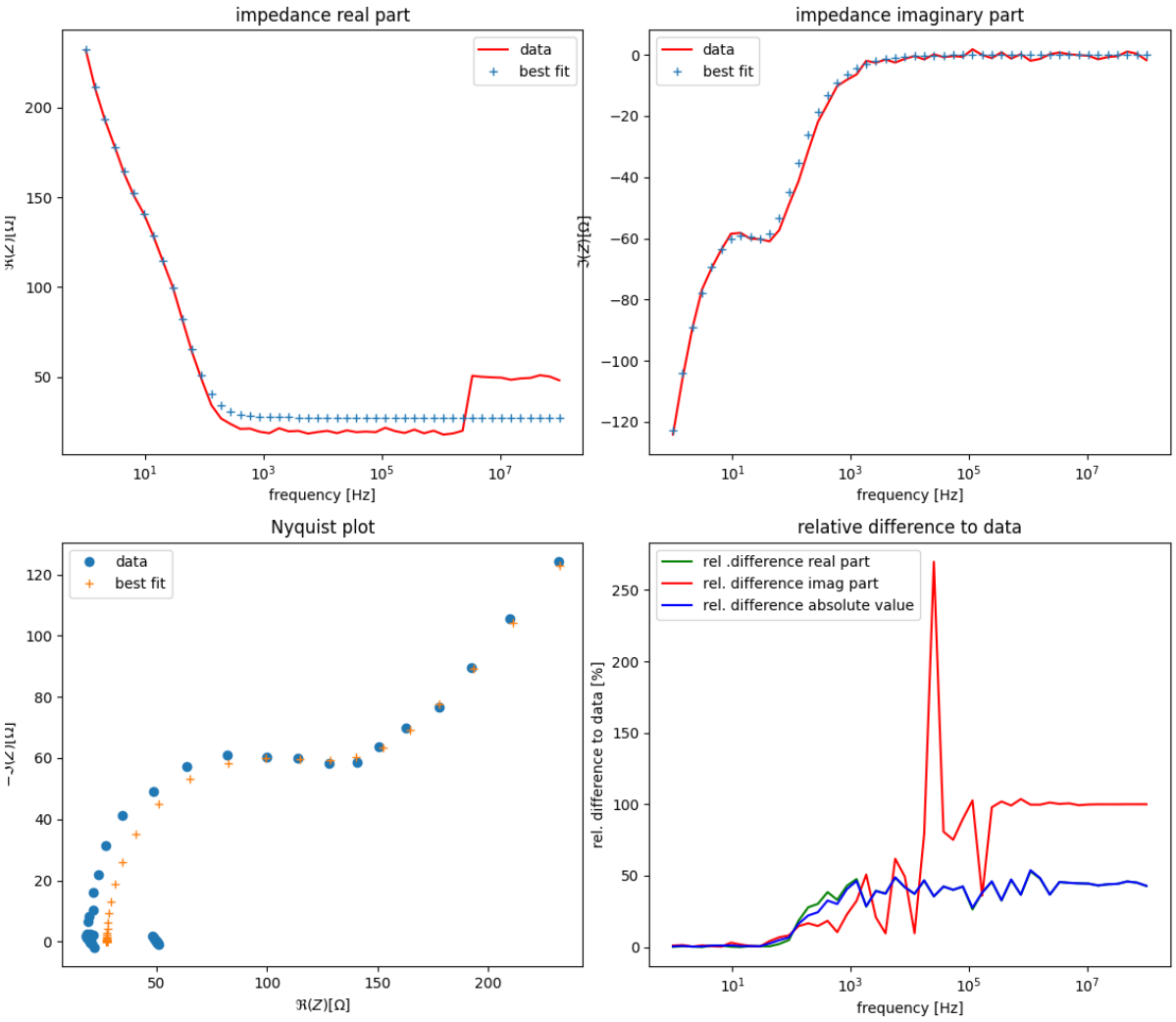
1.4 Customizing the solver

The previous example *Fitting experimental data* is chosen again.

This time, noise and outliers are added to the data

```
# parameters
frequencies = numpy.logspace(0, 8, num=50)
Rct = 100.
Rs = 20.
Aw = 300.
C0 = 25e-6
```

When running the fit with the standard least-squares solver, we obtain

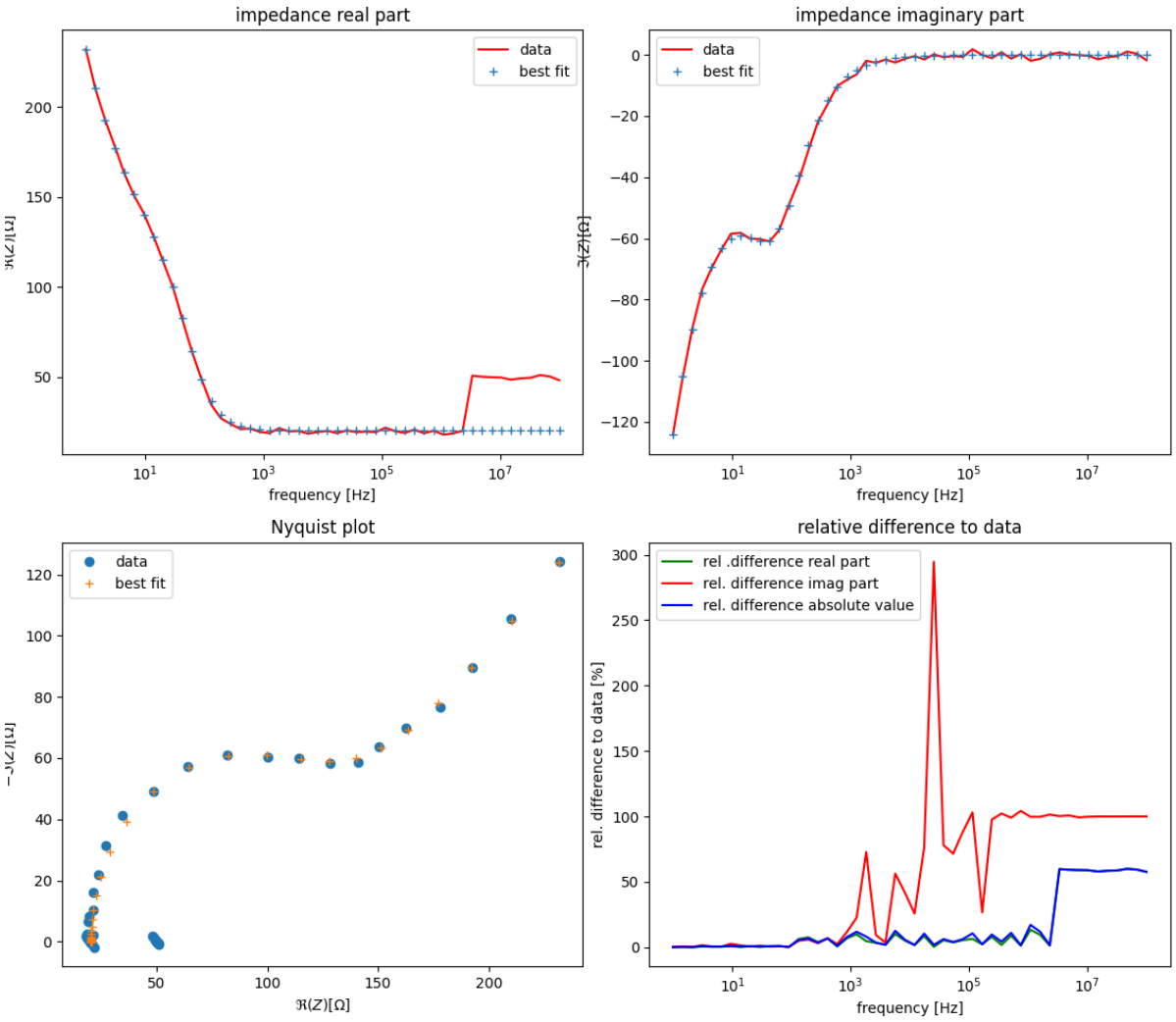


Instead, we can customize the solver:

```
df = pandas.DataFrame(data=data)
df.to_csv('test.csv', index=False)

# initialise fitter with verbose output
fitter = impedancefitter.Fitter('CSV', LogLevel='DEBUG')
```

and obtain



1.4.1 See Also

`examples/Randles/randles_solver.py`.

1.5 Analysing a larger data set

When performing multiple measurements on different samples, it is convenient to analyse the statistics of the data set.

Here, we show how to fit many samples and subsequently generate histograms and find the best probability distribution to describe the data.

```
#    The ImpedanceFitter is a package to fit impedance spectra to
#    equivalent-circuit models using open-source software.
#
#    Copyright (C) 2018, 2019 Leonard Thiele, leonard.thiele[AT]uni-rostock.de
#    Copyright (C) 2018, 2019, 2020 Julius Zimmermann, julius.zimmermann[AT]uni-
↪rostock.de
#
```

(continues on next page)

(continued from previous page)

```
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <https://www.gnu.org/licenses/>.

import numpy as np
import os
import pandas as pd
from collections import OrderedDict
from impedancefitter import get_equivalent_circuit_model, PostProcess, Fitter
from matplotlib import rcParams

rcParams['figure.figsize'] = [20, 10]

# parameters
f = np.logspace(1, 8)
omega = 2. * np.pi * f
R = 1000.
C = 1e-6

data = OrderedDict()
data['f'] = f

samples = 1000

model = "parallel(R, C)"
m = get_equivalent_circuit_model(model)
# generate random samples
for i in range(samples):
    Ri = 0.05 * R * np.random.randn() + R
    Ci = 0.05 * C * np.random.randn() + C

    Z = m.eval(omega=omega, R=Ri, C=Ci)
    # add some noise
    Z += np.random.randn(Z.size)

    data['real' + str(i)] = Z.real
    data['imag' + str(i)] = Z.imag

# save data to file
pd.DataFrame(data=data).to_csv('test.csv', index=False)

# initialize fitter
# LogLevel should be WARNING; otherwise there
# will be a lot of output


fitter = Fitter('CSV', LogLevel='WARNING')
os.remove('test.csv')
```

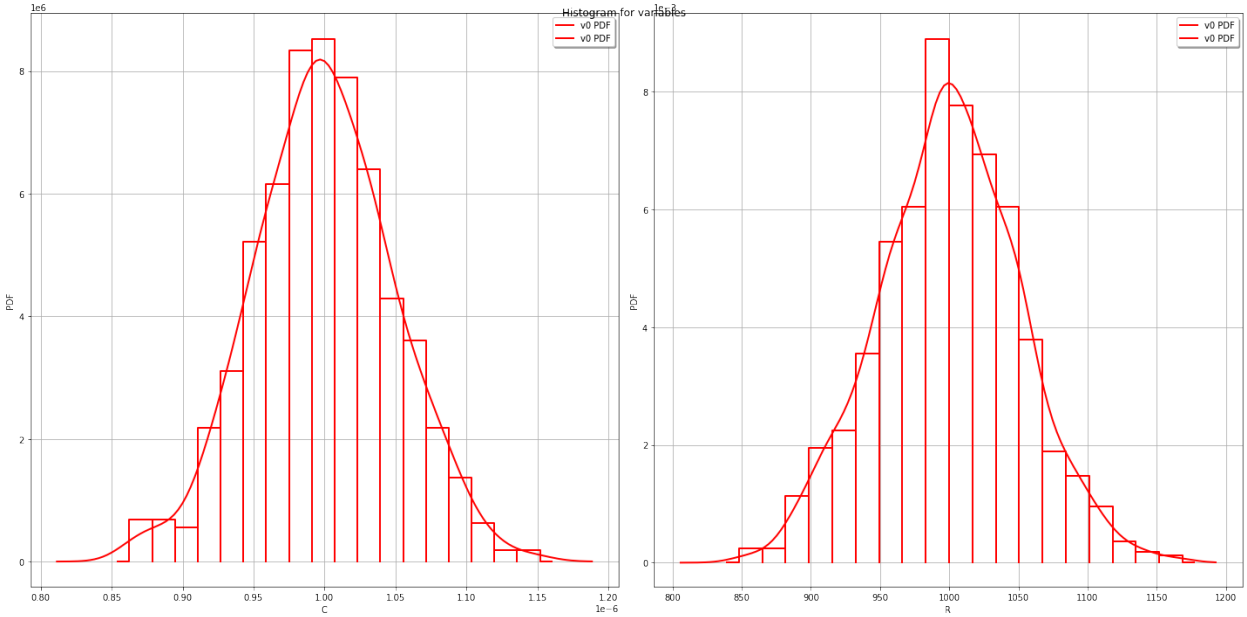
(continues on next page)

(continued from previous page)

```
parameters = {'R': {'value': R},
              'C': {'value': C}}
fitter.run(model, parameters=parameters)

postp = PostProcess(fitter.fit_data)
# show histograms
postp.plot_histograms()

# compare different algorithms to find matching model
print("BIC best model for R:",
      postp.best_model_bic('R', ['Normal', 'Beta', 'Gamma'])[0])
print("Chisquared best model for R:",
      postp.best_model_chisquared('R', ['Normal', 'Beta', 'Gamma'])[0])
print("Kolmogorov best model for R:",
      postp.best_model_kolmogorov('R', ['Normal', 'Beta', 'Gamma'])[0])
print("Expected result:\nNormal(mu = {}, sigma = {})".format(R, 0.05 * R))
```




```
BIC best model for R: Normal(mu = 999.898, sigma = 50.9305)
Chisquared best model for R: Normal(mu = 999.898, sigma = 50.9305)
Kolmogorov best model for R: Normal(mu = 0, sigma = 1)
Expected result:
Normal(mu = 1000.0, sigma = 50.0)
```

1.5.1 See Also

`examples/Statistics/statistics.py`.

1.6 Available file formats

We have analysed impedance data measured with different devices. Here, we show how the different file formats, in which the data are stored, can be read in. Please let us know if you have used ImpedanceFitter with other file formats.

1.6.1 Gamry Reference 600+

The data are saved in the DTA format. They can be read in using

```
fitter = impedancefitter.Fitter("DTA")
```

1.6.2 Sciospec ISX-3

The data are stored in *.spec* files, which are plain text files and can be read in using

```
fitter = impedancefitter.Fitter("TXT", ending=".spec", skiprows_txt=6, delimiter=",")
```

1.6.3 Keysight E4980AL

The data were stored through the USB interface and stored in a CSV file that is structured like: *frequency, real part of impedance, imaginary part of impedance, voltage, current*. For these files, we wrote a special function to read it in. It can be accessed through

```
fitter = impedancefitter.Fitter("CSV_E4980AL")
```

1.6.4 Various instruments with data stored in CSV or Excel files

The structure of CSV and Excel files has to be: *frequency, real part of impedance, imaginary part of impedance, real part, imaginary part, ...*. This means that there can be multiple impedance spectra in one file. However, all spectra have to be measured at the same frequency points, which must be stored in the first column of the data set. Then the data can be read in using:

```
fitter = impedancefitter.Fitter("CSV")
fitter = impedancefitter.Fitter("XSLX")
```

CHAPTER

TWO

FITTING

The script will cycle through all files in a selected directory (unless certain files are excluded or explicitly listed) and will store the experimental data. The experimental data can then be fitted to user-defined model.

2.1 Formulate the model

The ImpedanceFitter parser understands circuits that follow a simple pattern:

- Elements in series are connected by a +.
- Elements in parallel are connected by *parallel(A, B)*.

An example of a circuit could be:

```
parallel(R, C) + CPE
```

This stands for a resistor in parallel with a capacitor that are in series with a constant phase element (CPE).

Also nested parallels are possible:

```
parallel(parallel(L, C), R)
```

Find all available elements in Section *Circuit elements* and all available circuits in Section *Circuits*.

You can also use prefixes. This is needed if you want to combine multiple elements or circuits of the same type. Otherwise, the parameters cannot be distinguished by LMFIT.

For example:

```
parallel(R_f1, C_f1) + parallel(R_f2, C_f2)
```

2.2 Execute the fit

Using *impedancefitter.fitter.Fitter.run()*, those files can be fitted to an equivalent circuit model. If there are two models involved that shall be fitted sequentially for each file, refer to *impedancefitter.fitter.Fitter.sequential_run()*. This method allows one to communicate inferered parameters to the second model. In [Sabuncu2012], an example of such a sequential procedure has been presented.

2.3 Add a custom model

If you want to add a custom model that cannot be built by the existing models, you need to follow these steps:

1. Create a new function for this like

```
def example_function(omega, parameterA, parameterB):
    impedance = ...
    return impedance
```

   The first argument of the function needs to be named *omega* and is the angular frequency! The same holds true for elements. LMFIT generates the model based on the function arguments and always takes the first argument as the independent variable. The other parameters are then accessible by their names.

2. Give this model a reference name that does not contain numbers or underscores. Link it in `impedancefitter.utils._model_function()` to the function you defined in the previous step. Add the model to *impedancefitter.utils.available_models()*.
3. Add the new parameter names and their corresponding LaTex representation to *impedancefitter.utils.get_labels()*.
4. Write a unit test for the model.
5. Use the model.

2.4 API Reference

class impedancefitter.fitter.**Fitter**(*inputformat*, *directory=None*, ***kwargs*)

The main fitting object class. All files in the data directory with matching file ending are imported to be fitted.

Parameters

- **inputformat** (*string*) – The inputformat of the data files. Must be one of the formats specified in *impedancefitter.utils.available_file_format()*. Moreover, the ending of the file must match the *inputformat*.
- **directory** (*string, optional*) – Path to data directory. Provide the data directory if the data directory is not the current working directory.
- **LogLevel** (*{'DEBUG', 'INFO', 'WARNING'}, optional*) – choose level for logger. Case DEBUG: the script will output plots after each fit, case INFO: the script will output results from each fit to the console.
- **excludeEnding** (*string, optional*) – For file ending that should be ignored (if there are files with the same ending as the chosen inputformat). Useful for instance, if there are files like **_data.csv* and **_result.csv* around and only the first should be fitted.
- **ending** (*string, optional*) – When *inputformat* is *TXT*, an alternative file ending is possible. The default is *.TXT*.
- **minimumFrequency** (*float, optional*) – If you want to use another frequency than the minimum frequency in the dataset.
- **maximumFrequency** (*float, optional*) – If you want to use another frequency than the maximum frequency in the dataset.
- **data_sets** (*int, optional*) – Use only a certain number of data sets instead of all in directory.
- **current_threshold** (*float, optional*) – Use only for data from E4980AL LCR meter to check current. If the current is not close to the threshold, the data point will be neglected.

- **voltage_threshold** (*float, optional*) – Use only for data from E4980AL LCR meter to check voltage. If the voltage is not close to the threshold, the data point will be neglected.
- **E4980AL_tolerance** (*float, optional*) – Set tolerance for *current_threshold* and/or *voltage_threshold*.
- **write_output** (*bool, optional*) – Decide if you want to dump output to file. Default is False
- **fileList** (*list of strings, optional*) – provide a list of files that exclusively should be processed. No other files will be processed. This option is particularly good if you have a common fileending of your data (e.g., *.csv*)
- **show** (*bool, optional*) – Decide if you want to see the plots during the fitting procedure.
- **savefig** (*bool, optional*) – Decide if you want to save the plots. Default is False.
- **trace_b** (*string, optional*) – For TXT files, which contain more than one trace. The data is only read in until `trace_b` is found. Default is None (then trace_b does not have any effect).
- **skiprows_txt** (*int, optional*) – Number of header rows inside a TXT file. Default is 1.
- **skiprows_trace** (*int, optional*) – Lines between traces blocks in a TXT file. Default is None (then skiprows_trace does not have any effect).
- **delimiter** (*str, optional*) – Only for TXT and CSV files. If the TXT/CSV file is not tab-separated, this can be specified here.

Variables

- **omega_dict** (*dict*) – Contains frequency lists that were found in the individual files. The keys are the file names, the values the frequencies.
- **Z_dict** (*dict*) – Contains corresponding impedances. Note that the values might be lists when there was more than one impedance data set in the file.
- **fit_data** (*dict*) – Contains the fitting results for each individual file. In case of a sequential run, the dictionary contains two sub-dictionaries with keys *model1* and *model2* and the results.
- **fittedValues** (`lmfit.model.ModelResult`) – The fitting result of the last data set that was fitted. Exists only when *run()* was called.
- **fittedValues1** (`lmfit.model.ModelResult`) – The fitting result of the last data set that was fitted. Exists only when *sequential_run()* was called and corresponds to the first model in this run.
- **fittedValues2** (`lmfit.model.ModelResult`) – The fitting result of the last data set that was fitted. Exists only when *sequential_run()* was called and corresponds to the second model in this run.

cluster_emcee_result (*constant=100.0*, *show=False*)

Apply clustering to eliminate low-probability samples.

Parameters

- **constant** (*float*) – The constant, which is used to define the threshold from which on walkers are eliminated.
- **show** (*bool, optional*) – Plot clustering result.

Notes

The clustering approach described in [Hou2012] is implemented in this function. The walkers are sorted by probability and subsequently the difference between adjacent walker probabilities Δ_j is evaluated. Then the average difference between the current and the first walkeri ($\bar{\Delta}_j$) is evaluated. Both differences are compared and a threshold is defined:

$$\Delta_j > \text{constant} \cdot \bar{\Delta}_j$$

When this inequality becomes true, all walkers with $k > j$ are thrown away.

References

emcee_conf_interval (*result*)
Compute emcee confidence intervals.

The 1σ to 3σ confidence intervals are computed for a fitting result generated by emcee since this case is not covered by the original LMFIT implementation.

Parameters **result** (`lmfit.model.ModelResult`) – Result from fit.

Returns

Dictionary containing limits of confidence intervals for all free parameters. The limits are structured in a list with 7 items, which are ordered as follows:

1. lower limit of 3σ confidence interval.
2. lower limit of 2σ confidence interval.
3. lower limit of 1σ confidence interval.
4. median.
5. upper limit of 1σ confidence interval.
6. upper limit of 2σ confidence interval.
7. upper limit of 3σ confidence interval.

Return type dict

emcee_report ()
Reports acceptance fraction and autocorrelation times.

initialize_model (*modelname*, *log=False*, *eps=False*)
Interface to LMFIT model class.

The equivalent circuit (represented as a string) is parsed and a LMFIT Model is returned. This can be useful if one wants to compute the impedance values for a given model and use it in a different context.

Parameters

- **modelname** (*string*) – Provide equivalent circuit model to be parsed.
- **log** (*bool*) – Decide, if logscale is used for the model.
- **diel** (*bool*) – Convert to complex permittivity and fit this instead of impedance.

Returns **model** – The resulting LMFIT model.

Return type `lmfit.model.Model`

linkk_test(*capacitance=False*, *inductance=False*, *c=0.85*, *maxM=100*, *show=True*, *limits=[- 2, 2]*, *weighting='modulus'*)

Lin-KK test to check Kramers-Kronig validity.

Parameters

- **capacitance** (*bool, optional*) – Add extra capacitance to circuit.
- **inductance** (*bool, optional*) – Add extra inductance to circuit.
- **c** (*double, optional*) – Set threshold for algorithm as described in [Schoenleber2014]. Must be between 0 and 1.
- **maxM** (*int, optional*) – Maximum number of RC elements. Default is 100.
- **show** (*bool*) – Show plots of test result. Default is True.
- **limits** (*list, optional*) – Lower and upper limit of residual.
- **weighting** (*"modulus" or None*) – Apply modulus weighting (as in [Schoenleber2014]) or process unweighted data.

Returns

- **results** (*dict*) – Values of Lin-KK test for each file.
- **mus** (*dict*) – All *mu* values during Lin-KK run for each file.
- **residuals** (*dict*) – Least-squares residuals during Lin-KK run for each file.

Notes

The implementation of the algorithm follows [Schoenleber2014] closely.

If the option *savefig* is generally enabled, the plot result of the LinKK-Test will be saved to a pdf-file.

References

plot_initial_best_fit(*sequential=False*)

Plot initial and best fit together.

This method reveals how good the initial fit was.

Parameters **sequential** (*bool, optional*) – If a `sequential_run()` was performed, set this value to True.

plot_uncertainty_interval(*sigma=1*, *sequential=False*)

Plot uncertainty interval around best fit.

Parameters

- **sigma** (*{1, 2, 3}, optional*) – Choose sigma for confidence interval.
- **sequential** (*bool, optional*) – Set to True if you performed a sequential run before.

prepare_emcee_run(*leastsquaresresult*, *lnsigma={'max': 0.6931471805599453, 'min': -6.907755278982137, 'value': -2.3025850929940455}*, *nwalkers=100*, *radius=0.0001*, *weighted=False*, *burn=500*, *steps=10000.0*, *thin=10*, *fix_parameters=[]*)

Prepare initial configuration based on previous least squares run.

Parameters

- **leastsquaresresult** (`lmfit.ModelResult`) – Result of previous least squares run on same model. Basically the initial setting. Could also stem from a previous MCMC run.
- **lnsigma** (*dict*) – Value of initial guess for experimental uncertainty.
- **nwalkers** (*int*) – Number of walkers.
- **radius** (*float*) – Radius of ball around initial guess.
- **burn** (*int*) – Number of steps to be removed from MCMC chain in burn-in phase.
- **steps** (*int*) – Length of MCMC chain.
- **thin** (*int*) – Take only every *thin*-th step into account.
- **weighted** (*bool*) – If *False*, *lnsigma* will be used.
- **fix_parameters** (*list*) – Tell emcee to fix certain parameters to their least squares result.

Notes

The result from a previous least squares run is taking and the emcee run is prepared. The walkers are created and their initial positions are assigned. The initial positions are placed in a tight Gaussian ball around the least squares guess. Their radius can be set manually. Important parameters for the MCMC chains are set.

Returns dictionary, which can be passed to run method via *solver_kwargs* keyword.

Return type dict

process_data_from_file(*filename*, *model*, *parameters*, *residual='parts'*, *limits_residual=None*, *weighting_model=False*)

Fit data from input file to model.

Wrapper for LMFIT fitting routine.

Parameters

- **filename** (*str*) – Filename, which is contained in the data dictionaries `omega_dict` and `z_dict`.
- **model** (`lmfit.model.Model` or `lmfit.model.CompositeModel`) – The model to fit to.
- **parameters** (`lmfit.parameter.Parameters`) – The model parameters to be used.
- **residual** (*str*) – Plot relative difference w.r.t. real and imaginary part if *parts*. Plot relative difference w.r.t. absolute value if *absolute*. Plot difference (residual) if *diff*.
- **limits_residual** (*list, optional*) – List with entries *[bottom, top]* for y-axis of residual plot.

Returns Result of fit as `lmfit.model.ModelResult` object.

Return type `lmfit.model.ModelResult`

run(*modelname*, *solver=None*, *parameters=None*, *solver_kwargs={}*, *log=False*, *weighting=None*, *show=False*, *report=False*, *savemodelresult=True*, *eps=False*, *residual='parts'*, *limits_residual=None*, *weighting_model=False*)

Main function that iterates through all data sets provided.

Parameters

- **modelname** (*string*) – Name of the model to be parsed. Must be built by those provided in *`impedancefitter.utils.available_models()`* and using + and *parallel(x, y)* as possible representations of series or parallel circuit.

- **solver** (*string, optional*) – Choose an optimizer. Must be available in LMFIT. Default is least_squares
- **parameters** (*dict, optional*) – Provide parameters if you do not want to read them from a yaml file (for instance in parallel UQ runs).
- **solver_kwargs** (*dict, optional*) – Customize the employed solver. Interface to the LMFIT routine.
- **weighting** (*str, optional*) – Choose a weighting scheme. Default is unit weighting. Also possible: proportional and modulus weighting. See [Barsoukov2018] and [Orazem2017] for more information. Moreover, weighted least squares is possible. This should only be chosen if the data can be averaged. The average data will be fitted and the standard deviation is used for the weights. The keyword for this is WLS. In this case, you need to provide weights.
- **savemodelresult** (*bool, optional*) – Saves all `lmfit.model.ModelResult` instances to plot or evaluate uncertainty later.
- **residual** (*str*) – Plot relative difference w.r.t. real and imaginary part if *parts*. Plot relative difference w.r.t. absolute value if *absolute*. Plot difference (residual) if *diff*.
- **limits_residual** (*list, optional*) – List with entries *[bottom, top]* for y-axis of residual plot.

sequential_run(*model1*, *model2*, *communicate*, *solver=None*, *solver_kwargs={}*, *parameters1=None*, *parameters2=None*, *weighting=None*)

Main function that iterates through all data sets provided.

Here, two models are fitted sequentially and fitted parameters can be communicated from one model to the other.

Parameters

- **model1** (*string*) – Name of first model. Must be built by those provided in *impedancefitter.utils.available_models()* and using + and *parallel(x, y)* as possible representations of series or parallel circuit
- **model2** (*string*) – Name of second model. Must be built by those provided in *impedancefitter.utils.available_models()* and using + and *parallel(x, y)* as possible representations of series or parallel circuit
- **communicate** (*list of strings*) – Names of parameters that should be communicated from model1 to model2. Requires that model2 contains a parameter that is named appropriately.
- **solver** (*string, optional*) – choose an optimizer. Must be available in LMFIT. Default is least_squares
- **solver_kwargs** (*dict, optional*) – Customize the employed solver. Interface to the LMFIT routine.
- **parameters1** (*dict, optional*) – Parameters of model1. Provide parameters if you do not want to use a yaml file.
- **parameters2** (*dict, optional*) – Parameters of model2. Provide parameters if you do not want to use a yaml file.
- **weighting** (*str, optional*) – Choose a weighting scheme. Default is unit weighting. Also possible: proportional weighting. See [Barsoukov2018] for more information.

visualize_data(*savefig=False*, *Zlog=False*, *allinone=False*, *plottype='impedance'*, *show=True*, *legend=True*)

Visualize impedance data.

Parameters

- **savefig** (*bool, optional*) – Decide if plots should be saved as pdf. Default is False. Saves the file as *filename* + index of dataset + *_impedance_overview.pdf*. If *allinone* is True, then it is *allinone_impedance_overview.pdf*.
- **Zlog** (*bool, optional*) – Plot impedance on logscale.
- **show** (*bool, optional*) – Decide if you want to immediately see the plot.
- **allinone** (*bool, optional*) – Visualize all data sets in one plot
- **plottype** (*str, optional*) – Choose between standard impedance plot ('impedance'), resistance / capacitance ("RC") and bode plot ('bode').
- **legend** (*str, optional*) – Choose if a legend should be shown. Recommended to switch to False when using large datasets.

CHAPTER

THREE

STATISTICAL ANALYSIS

To analyse the fit results when the data set is rather large, there exists an interface to OpenTurns. It can generate histograms or find the best distribution to describe a certain fit parameter of the data set.

3.1 API Reference

class impedancefitter.postprocess.**PostProcess**(*fitresult=None*, *yamlfile=False*)
This class provides the possibility to statistically analyse the fitted data.

Parameters

- **fitresult** (*dict*) – Result of the fit.
- **yamlfile** (*bool*) – Provide the link to a file from which you want to read the results.

Notes

Provide either *fitresult* or *yamlfile*.

best_model_bic(*parameter*, *distributions*, *showQQ=False*)
Test, which distribution models your data best based on the Bayesian information criterion.

Parameters

- **parameter** (*string*) – Parameter, whose distribution is to be found.
- **distributions** (*list*) – List with strings describing valid OpenTURNS distributions such as *['Normal', 'Uniform']*

Returns

- openturns.Distribution
- *float*

best_model_chisquared(*parameter*, *distributions*, *showQQ=False*)
Test, which distribution models your data best based on the chisquared test.

Parameters

- **parameter** (*string*) – Parameter, whose distribution is to be found.
- **distributions** (*list*) – List with strings describing valid OpenTURNS distributions such as *['Normal', 'Uniform']*

Returns

- openturns.Distribution

- openturns.TestResult

best_model_kolmogorov (*parameter*, *distributions*, *showQQ=False*)
Test, which distribution models your data best based on the kolmogorov test.

Parameters

- **parameter** (*string*) – Parameter, whose distribution is to be found.
- **distributions** (*list*) – List with strings describing valid OpenTURNS distributions such as *['Normal', 'Uniform']*

Returns

- openturns.Distribution
- openturns.TestResult

See also:

openturns.FittingTest_BestModelKolmogorov()

best_model_lilliefors (*parameter*, *distributions*, *showQQ=False*)
Test, which distribution models your data best based on the Lilliefors test.

Parameters

- **parameter** (*string*) – Parameter, whose distribution is to be found.
- **distributions** (*list*) – List with strings describing valid OpenTURNS distributions such as *['Normal', 'Uniform']*

Returns

- openturns.Distribution
- openturns.TestResult

See also:

openturns.FittingTest_BestModelLilliefors()

fit_to_histogram_distribution (*parameter*, *showQQ=False*)
Generate histogram from results.

Parameters **parameter** (*string*) – Parameter, whose distribution is to be found.

Returns

Return type openturns.Distribution

fit_to_normal_distribution (*parameter*, *showQQ=False*)
Fit results for to normal distribution.

Parameters

- **parameter** (*string*) – Parameter, whose distribution is to be found.
- **showQQ** (*bool, optional*) – Decide if you want to check the fit visually

Returns

Return type openturns.Distribution

plot_histograms (*savefig=False*, *show=True*)
Plot histograms for all determined parameters.

Parameters

- **savefig** (*bool, optional*) – Set to True if you want to save the figure *histograms.pdf*.
- **show** (*bool, optional*) – Switch on or off if figures is shown.

Notes

Fails if values are too close to each other, i.e. the variance is very small.

CHAPTER

FOUR

CIRCUIT ELEMENTS

The following elements are available. Since prefixes are possible, each element is referred to as by a special name. The elements' parameters are called as in the original function. This is the concept of LMFIT.

4.1 Names for building the model

Name	Corresponding function
R	*impedancefitter.elements.Z_R()*
C	*impedancefitter.elements.Z_C()*
L	*impedancefitter.elements.Z_L()*
W	*impedancefitter.elements.Z_w()*
Wo	*impedancefitter.elements.Z_wo()*
Ws	*impedancefitter.elements.Z_ws()*
Cstray	*impedancefitter.elements.Z_stray()*

4.2 API reference

impedancefitter.elements.**Z_C**(*omega*, *C*)
Capacitor impedance

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **C** (*double*) – capacitance of capacitor

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.elements.**Z_CPE**(*omega*, *k*, *alpha*)
CPE impedance

$$Z_{\mathrm{CPE}} = k(j\omega)^{-\alpha}$$

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **k** (*double*) – CPE factor
- **alpha** (*double*) – CPE phase

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.elements.**Z_L**(*omega*, *L*)
Impedance of an inductor.

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **L** (*double*) – inductance

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.elements.**Z_R**(*omega*, *R*)
Create array for a resistor.

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **R** (*double*) – Resistance.

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.elements.**Z_stray**(*omega*, *Cs*)
Stray capacitance in pF

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **Cs** (*double*) – Stray capacitance, for numerical reasons in pF.

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.elements.**Z_w**(*omega*, *Aw*)
Warburg element

$$Z_{\mathrm{W}} = A_{\mathrm{W}} \frac{1-j}{\sqrt{\omega}}$$

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **Aw** (*double*) – Warburg coefficient

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.elements.**Z_wo**(*omega*, *Aw*, *B*)
Warburg open element

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **Aw** (*double*) – Warburg coefficient
- **B** (*double*) – Second coefficient

Returns Impedance array

Return type numpy.ndarray, complex

Notes

This element is also referred to as finite-length Warburg element with reflective boundary [Barsoukov2018]. Here, the formulation

$$Z_\mathrm{W} = \frac{A_\mathrm{W}}{\tanh\left(B\sqrt{j\omega}\right)\sqrt{j\omega}}$$

is implemented.

impedancefitter.elements.**Z_ws**(*omega*, *Aw*, *B*)
Warburg short element

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **Aw** (*double*) – Warburg coefficient
- **B** (*double*) – Second coefficient

Returns Impedance array

Return type numpy.ndarray, complex

Notes

This element is also referred to as finite-length Warburg element with transmissive boundary [Barsoukov2018]. Here, the formulation

$$Z_\mathrm{W} = A_\mathrm{W}\frac{\tanh\left(B\sqrt{j\omega}\right)}{\sqrt{j\omega}}$$

is implemented.

impedancefitter.elements.**eps**(*Z1*, *make_eps*)
Return complex permittivity for LMFIT

Parameters

- **Z1** (numpy.ndarray, complex) – Impedance 1 (model fit)
- **make_eps** (numpy.ndarray, complex) – Output of function impedancefitter.utils.make_eps()

impedancefitter.elements.**log**(*Z1*, *dummy*)
Return logarithm of impedance for LMFIT

Parameters

- **Z1** (numpy.ndarray, complex) – Impedance 1 (model fit)
- **dummy** (numpy.ndarray, complex) – Array full of ones, such that np.log10(dummy) is zero.

impedancefitter.elements.**parallel**(*Z1*, *Z2*)
Return values of parallel circuit.

Parameters

- **Z1** (numpy.ndarray, complex) – Impedance 1
- **Z2** (numpy.ndarray, complex) – Impedance 2

Returns Impedance array

Return type numpy.ndarray, complex

CHAPTER

FIVE

CIRCUITS

There exist a few predefined circuits that were implemented based on published papers. Usually, those circuits are rather complex and cannot be built by the existing elements or feature parameters in certain ranges or units that are not consistent with the generally chosen unit set.

5.1 Names for building the model

To build your equivalent circuit model, choose a model from the following list. The function, which tells you the parameter names for the respective model is linked to each model.

Name	Corresponding function
ColeCole	*impedancefitter.cole_cole.cole_cole_model()*
ColeCole2	*impedancefitter.cole_cole.cole_cole_2_model()*
ColeCole2tissue	*impedancefitter.cole_cole.cole_cole_2tissue_model()*
ColeCole3	*impedancefitter.cole_cole.cole_cole_3_model()*
ColeCole4	*impedancefitter.cole_cole.cole_cole_4_model()*
ColeColeR	*impedancefitter.cole_cole.cole_cole_R_model()*
Randles	*impedancefitter.randles.Z_randles()*
RandlesCPE	*impedancefitter.randles.Z_randles_CPE()*
DRC	*impedancefitter.RC.drc_model()*
RCfull	*impedancefitter.RC.RC_model()*
RC	*impedancefitter.RC.rc_model()*
RCtau	*impedancefitter.RC.rc_tau_model()*
SingleShell	*impedancefitter.single_shell.single_shell_model()*
DoubleShell	*impedancefitter.double_shell.double_shell_model()*
CPE	*impedancefitter.cpe.cpe_model()*
CPECT	*impedancefitter.cpe.cpe_ct_model()*
CPECTW	*impedancefitter.cpe.cpe_ct_w_model()*
CPETissue	*impedancefitter.cpe.cpetissue_model()*
CPECTTissue	*impedancefitter.cpe.cpe_ct_tissue_model()*
CPEonset	*impedancefitter.cpe.cpe_onset_model()*
LR	*impedancefitter.loss.Z_in()*
LCR	*impedancefitter.loss.Z_loss()*
HavriliakNegami	*impedancefitter.cole_cole.havriliak_negami()*
HavriliakNegamiTissue	*impedancefitter.cole_cole.havriliak_negamitissue()*

5.2 Cole-Cole circuits

impedancefitter.cole_cole.**cole_cole_2_model**(*omega*, *c0*, *epsinf*, *deps1*, *deps2*, *tau1*, *tau2*, *a1*, *a2*, *sigma*)

Standard 2-Cole-Cole impedance model.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for unit capacitance, in pF
- **epsinf** (*double*) – value for ε_∞
- **deps1** (*double*) – value for $\Delta\varepsilon_1$
- **deps2** (*double*) – value for $\Delta\varepsilon_2$
- **deps3** (*double*) – value for $\Delta\varepsilon_3$
- **tau1** (*double*) – value for τ_1, in ps
- **tau2** (*double*) – value for τ_2, in ns
- **a1** (*double*) – value for $1 - \alpha_1 = a$
- **a2** (*double*) – value for $1 - \alpha_2 = a$
- **sigma** (*double*) – conductivity value

Returns Impedance array

Return type numpy.ndarray, complex

Notes

The original model has been described in [Gabriel1996]. Here, two instead of four dispersions are used.

impedancefitter.cole_cole.**cole_cole_2tissue_model**(*omega*, *c0*, *epsinf*, *deps1*, *deps2*, *tau1*, *tau2*, *a1*, *a2*, *sigma*)

2-Cole-Cole impedance model for tissues.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for unit capacitance, in pF
- **epsinf** (*double*) – value for ε_∞
- **deps1** (*double*) – value for $\Delta\varepsilon_1 \cdot 10^3$
- **deps2** (*double*) – value for $\Delta\varepsilon_2 \cdot 10^6$
- **tau1** (*double*) – value for τ_1, in us
- **tau2** (*double*) – value for τ_2, in ms
- **a1** (*double*) – value for $1 - \alpha_1 = a$
- **a2** (*double*) – value for $1 - \alpha_2 = a$
- **sigma** (*double*) – conductivity value

Returns Impedance array

Return type numpy.ndarray, complex

Notes

The original model has been described in [Gabriel1996]. Here, two instead of four dispersions are used.

impedancefitter.cole_cole.**cole_cole_3_model**(*omega*, *c0*, *epsinf*, *deps1*, *deps2*, *deps3*, *tau1*, *tau2*, *tau3*, *a1*, *a2*, *a3*, *sigma*)

Standard 3-Cole-Cole impedance model.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for unit capacitance, in pF
- **epsinf** (*double*) – value for ε_∞
- **deps1** (*double*) – value for $\Delta\varepsilon_1$
- **deps2** (*double*) – value for $\Delta\varepsilon_2$
- **deps3** (*double*) – value for $\Delta\varepsilon_3$
- **tau1** (*double*) – value for τ_1, in ps
- **tau2** (*double*) – value for τ_2, in ns
- **tau3** (*double*) – value for τ_3, in us
- **a1** (*double*) – value for $1 - \alpha_1 = a$
- **a2** (*double*) – value for $1 - \alpha_2 = a$
- **a3** (*double*) – value for $1 - \alpha_3 = a$
- **sigma** (*double*) – conductivity value

Returns Impedance array

Return type numpy.ndarray, complex

Notes

The original model has been described in [Gabriel1996]. Here, three instead of four dispersions are used.

impedancefitter.cole_cole.**cole_cole_4_model**(*omega*, *c0*, *epsinf*, *deps1*, *deps2*, *deps3*, *deps4*, *tau1*, *tau2*, *tau3*, *tau4*, *a1*, *a2*, *a3*, *a4*, *sigma*)

Standard 4-Cole-Cole impedance model.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for unit capacitance, in pF
- **epsinf** (*double*) – value for ε_∞
- **deps1** (*double*) – value for $\Delta\varepsilon_1$
- **deps2** (*double*) – value for $\Delta\varepsilon_2$
- **deps3** (*double*) – value for $\Delta\varepsilon_3$
- **deps4** (*double*) – value for $\Delta\varepsilon_4$
- **tau1** (*double*) – value for τ_1, in ps
- **tau2** (*double*) – value for τ_2, in ns

- **tau3** (*double*) – value for τ_3, in us
- **tau4** (*double*) – value for τ_4, in ms
- **a1** (*double*) – value for $1 - \alpha_1 = a$
- **a2** (*double*) – value for $1 - \alpha_2 = a$
- **a3** (*double*) – value for $1 - \alpha_3 = a$
- **a4** (*double*) – value for $1 - \alpha_4 = a$
- **sigma** (*double*) – conductivity value

Returns Impedance array

Return type `numpy.ndarray`, complex

Notes

The original model has been described in [Gabriel1996].

References

`impedancefitter.cole_cole.`**`cole_cole_R_model`**(*omega*, *Rinf*, *R0*, *tau*, *a*)

Standard Cole-Cole circuit for macroscopic quantities.

See for example [Schwan1957] for more information.

Parameters

- **omega** (`numpy.ndarray`, double) – list of frequencies
- **Rinf** (*double*) – value for R_∞
- **R0** (*double*) – value for R_0
- **tau** (*double*) – value for τ, in ns
- **a** (*double*) – value for $1 - \alpha = a$

Returns Impedance array

Return type `numpy.ndarray`, complex

Notes

Equation for calculations:

$$Z_{\mathrm{Cole}} = R_\infty + \frac{R_0 - R_\infty}{1 + (j\omega\tau)^a}$$

Warning: The time constant tau is in ns!

References

impedancefitter.cole_cole.**cole_cole_model**(*omega*, *c0*, *epsl*, *tau*, *a*, *sigma*, *epsinf*)
Cole-Cole model for dielectric properties.

The model was implemented as presented in [Sabuncu2012]. You need to provide the unit capacitance of your device to get the dielectric properties of the Cole-Cole model.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for c_0, unit capacitance in pF
- **epsinf** (*double*) – value for ε_∞
- **epsl** (*double*) – value for ε_l
- **tau** (*double*) – value for τ, in ns
- **sigma** (*double*) – value for σ_dc
- **a** (*double*) – value for $1 - \alpha = a$

Returns Impedance array

Return type numpy.ndarray, complex

Notes

> **Warning:** The unit capacitance is in pF! The time constant tau is in ns!

Equations for calculations:

$$\varepsilon^* = \varepsilon_\infty + \frac{\varepsilon_\mathrm{l} - \varepsilon_\infty}{1 + (j\omega\tau)^a} - \frac{j\sigma_\mathrm{dc}}{\omega\varepsilon_0}$$

$$Z = \frac{1}{j\varepsilon^*\omega c_0}$$

References

impedancefitter.cole_cole.**havriliak_negami**(*omega*, *c0*, *epsinf*, *deps*, *tau*, *a*, *beta*, *sigma*)
Havriliak-Negami relaxation.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for c_0, unit capacitance in pF
- **epsinf** (*double*) – value for ε_∞
- **deps** (*double*) – value for $\Delta\varepsilon$
- **tau** (*double*) – value for τ, in ns
- **sigma** (*double*) – value for σ_dc
- **a** (*double*) – value for $1 - \alpha = a$

- **beta** (*double*) – value for β

Returns Impedance array

Return type `numpy.ndarray`, complex

Notes

Warning: The unit capacitance is in pF! The time constant tau is in ns!

Equations for calculations:

$$\varepsilon^* = \varepsilon_\infty + \frac{\Delta\varepsilon}{(1+(j\omega\tau)^a)^\beta} - \frac{j\sigma_{\mathrm{DC}}}{\omega\varepsilon_0} \ ,$$

$$Z = \frac{1}{j\varepsilon^*\omega c_0}$$

`impedancefitter.cole_cole.`**`havriliak_negamitissue`**(*omega*, *c0*, *epsinf*, *deps*, *tau*, *a*, *beta*, *sigma*)

Havriliak-Negami relaxation.

Parameters

- **omega** (`numpy.ndarray`, double) – list of frequencies
- **c0** (*double*) – value for c_0, unit capacitance in pF
- **epsinf** (*double*) – value for ε_∞
- **deps** (*double*) – value for $\Delta\varepsilon$
- **tau** (*double*) – value for τ, in μs
- **sigma** (*double*) – value for σ_{dc}
- **a** (*double*) – value for $1-\alpha = a$
- **beta** (*double*) – value for β

Returns Impedance array

Return type `numpy.ndarray`, complex

Notes

Warning: The unit capacitance is in pF! The time constant tau is in μs!

Equations for calculations:

$$\varepsilon^* = \varepsilon_\infty + \frac{\Delta\varepsilon}{(1+(j\omega\tau)^a)^\beta} - \frac{j\sigma_{\mathrm{DC}}}{\omega\varepsilon_0} \ ,$$

$$Z = \frac{1}{j\varepsilon^*\omega c_0}$$

impedancefitter.cole_cole.**raicu**(*omega*, *c0*, *epsinf*, *deps*, *tau*, *alpha*, *beta*, *gamma*, *sigma*)
Raicu relaxation.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for c_0, unit capacitance in pF
- **epsinf** (*double*) – value for ε_∞
- **deps** (*double*) – value for $\Delta\varepsilon$
- **tau** (*double*) – value for τ, in μs
- **sigma** (*double*) – value for σ_{dc}
- **a** (*double*) – value for α
- **beta** (*double*) – value for β
- **gamma** (*double*) – value for γ

Returns Impedance array

Return type numpy.ndarray, complex

Notes

Model was described in [Raicu1999].

Warning: The unit capacitance is in pF! The time constant tau is in μs!

Equations for calculations:

$$\varepsilon^* = \varepsilon_\infty + \frac{\Delta\varepsilon}{[(j\omega\tau)^\alpha + (j\omega\tau)^{1-\beta}])^\gamma} - \frac{j\sigma_{\mathrm{DC}}}{\omega\varepsilon_0} \ ,$$

$$Z = \frac{1}{j\varepsilon^*\omega c_0}$$

Note that $f_c = 2\pi/\tau$. The formulation using *fc* has for example been used in [Stoneman2007].

References

5.3 Single-Shell model

impedancefitter.single_shell.**eps_cell_single_shell**(*omega*, *em*, *km*, *kcp*, *ecp*, *dm*, *Rc*)
Single Shell model

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **em** (*double*) – membrane permittivity, value for ε_{m}
- **km** (*double*) – membrane conductivity, value for σ_{m}
- **ecp** (*double*) – cytoplasm permittivity, value for $\varepsilon_{\mathrm{cp}}$
- **kcp** (*double*) – cytoplasm conductivity, value for σ_{cp}

- **dm** (*double*) – membrane thickness, value for d_m
- **Rc** (*double*) – cell radius, value for R_c

Returns Complex permittivity array

Return type numpy.ndarray, complex

impedancefitter.single_shell.**single_shell_model**(*omega*, *em*, *km*, *kcp*, *ecp*, *kmed*, *emed*, *p*, *c0*, *dm*, *Rc*)

Single Shell model

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for c_0, unit capacitance in pF
- **em** (*double*) – membrane permittivity, value for ε_m
- **km** (*double*) – membrane conductivity, value for σ_m in μS/m
- **ecp** (*double*) – cytoplasm permittivity, value for ε_cp
- **kcp** (*double*) – cytoplasm conductivity, value for σ_cp
- **emed** (*double*) – medium permittivity, value for ε_med
- **kmed** (*double*) – medium conductivity, value for σ_med
- **p** (*double*) – volume fraction
- **dm** (*double*) – membrane thickness, value for d_m
- **Rc** (*double*) – cell radius, value for R_c

Returns Impedance array

Return type numpy.ndarray, complex

Notes

Warning: The unit capacitance is in pF!

Equations for the single-shell-model [Feldman2003]:

$$\nu_1 = \left(1 - \frac{d_\mathrm{m}}{R_\mathrm{c}}\right)^3$$

$$\varepsilon_\mathrm{m} = \varepsilon_\mathrm{m} - j\frac{\sigma_\mathrm{m}}{\varepsilon_0 \omega}$$

$$\varepsilon_\mathrm{cp} = \varepsilon_\mathrm{cp} - j\frac{\sigma_\mathrm{cp}}{\varepsilon_0 \omega}$$

$$\varepsilon^*_\mathrm{cell} = \varepsilon^*_\mathrm{m}\frac{2(1-\nu_1) + (1+2\nu_1)E_1}{(2+\nu_1) + (1-\nu_1)E_1}$$

$$E_1 = \frac{\varepsilon^*_\mathrm{cp}}{\varepsilon^*_\mathrm{m}}$$

$$\varepsilon^*_\mathrm{sus} = \varepsilon^*_\mathrm{med}\frac{(2\varepsilon^*_\mathrm{med} + \varepsilon^*_\mathrm{cell}) - 2p(\varepsilon^*_\mathrm{med} - \varepsilon^*_\mathrm{cell})}{(2\varepsilon^*_\mathrm{med} + \varepsilon^*_\mathrm{cell}) + p(\varepsilon^*_\mathrm{med} - \varepsilon^*_\mathrm{cell})}$$

$$Z = \frac{1}{j\varepsilon^*_\mathrm{sus}\omega c_0}$$

References

See also:

impedancefitter.double_shell.double_shell_model()

5.4 Double-Shell model

impedancefitter.double_shell.**double_shell_model**(*omega*, *km*, *em*, *kcp*, *ecp*, *ene*, *kne*, *knp*, *enp*, *kmed*, *emed*, *p*, *c0*, *dm*, *Rc*, *dn*, *Rn*)

Double Shell model.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies
- **c0** (*double*) – value for c_0, unit capacitance in pF
- **em** (*double*) – membrane permittivity,membrane permittivity, value for ε_m
- **km** (*double*) – membrane conductivity, value for σ_m in μS/m
- **ecp** (*double*) – cytoplasm permittivity, value for ε_cp
- **kcp** (*double*) – cytoplasm conductivity, value for σ_cp
- **ene** (*double*) – nuclear envelope permittivity, value for ε_ne
- **kne** (*double*) – nuclear envelope conductivity, value for σ_ne in mS/m
- **enp** (*double*) – nucleoplasm permittivity, value for ε_np
- **knp** (*double*) – nucleoplasm conductivity, value for σ_np
- **emed** (*double*) – medium permittivity, value for ε_med
- **kmed** (*double*) – medium conductivity, value for σ_med
- **p** (*double*) – volume fraction
- **dm** (*double*) – membrane thickness, value for d_m
- **Rc** (*double*) – cell radius, value for R_c
- **dn** (*double*) – nuclear envelope thickness, value for d_n
- **Rn** (*double*) – nucleus radius, value for R_n

Returns Impedance array

Return type numpy.ndarray, complex

Notes

Warning: The unit capacitance is in pF!

Equations for the single-shell-model [Feldman2003]:

$$\nu_1 = \left(1 - \frac{d_\mathrm{m}}{R_\mathrm{c}}\right)^3$$

$$\nu_2 = \left(\frac{R_\mathrm{n}}{R-d}\right)^3$$

$$\nu_3 = \left(1 - \frac{d_\mathrm{n}}{R_\mathrm{n}}\right)^3$$

$$\varepsilon_\mathrm{c}^* = \varepsilon_\mathrm{m}^* \frac{2(1-\nu_1) + (1+2\nu_1)E_1}{(2+\nu_1) + (1-\nu_1)E_1}$$

$$E_1 = \frac{\varepsilon_\mathrm{cp}^*}{\varepsilon_\mathrm{m}^*} \frac{2(1-\nu_2) + (1+2\nu_2)E_2}{(2+\nu_2) + (1-\nu_2)E_2}$$

$$E_2 = \frac{\varepsilon_\mathrm{ne}^*}{\varepsilon_\mathrm{cp}^*} \frac{2(1-\nu_3) + (1+2\nu_3)E_3}{(2+\nu_3) + (1-\nu_3)E_3}$$

$$E_3 = \frac{\varepsilon_\mathrm{np}^*}{\varepsilon_\mathrm{ne}^*}$$

$$\varepsilon_\mathrm{m} = \varepsilon_\mathrm{m} - j\frac{\sigma_\mathrm{m}}{\varepsilon_0 \omega}$$

$$\varepsilon_\mathrm{cp} = \varepsilon_\mathrm{cp} - j\frac{\sigma_\mathrm{cp}}{\varepsilon_0 \omega}$$

$$\varepsilon_\mathrm{ne} = \varepsilon_\mathrm{ne} - j\frac{\sigma_\mathrm{ne}}{\varepsilon_0 \omega}$$

$$\varepsilon_\mathrm{np} = \varepsilon_\mathrm{np} - j\frac{\sigma_\mathrm{np}}{\varepsilon_0 \omega}$$

$$\varepsilon_\mathrm{cell}^* = \varepsilon_\mathrm{m}^* \frac{2(1-\nu_1) + (1+2\nu_1)E_1}{(2+\nu_1) + (1-\nu_1)E_1}$$

$$\varepsilon_\mathrm{sus}^* = \varepsilon_\mathrm{med}^* \frac{(2\varepsilon_\mathrm{med}^* + \varepsilon_\mathrm{cell}^*) - 2p(\varepsilon_\mathrm{med}^* - \varepsilon_\mathrm{cell}^*)}{(2\varepsilon_\mathrm{med}^* + \varepsilon_\mathrm{cell}^*) + p(\varepsilon_\mathrm{med}^* - \varepsilon_\mathrm{cell}^*)}$$

$$Z = \frac{1}{j\varepsilon_\mathrm{sus}^* \omega c_0}$$

In [Ermolina2000] , there have been reported upper/lower limits for certain parameters. They could act as a first guess for the bounds of the optimization method.

Parameter	lower limit	upper limit
ε_{m}	1.4	16.8
σ_{m}	8e-8	5.6e-5
$\varepsilon_{\mathrm{cp}}$	60	77
σ_{cp}	0.033	1.1
$\varepsilon_{\mathrm{ne}}$	6.8	100
σ_{ne}	8.3e-5	7e-3
$\varepsilon_{\mathrm{np}}$	32	300
σ_{np}	0.25	2.2
R	3.5e-6	10.5e-6
R_{n}	2.95e-6	8.85e-6
d	3.5e-9	10.5e-9
d_{n}	2e-8	6e-8

See also:

impedancefitter.single_shell.single_shell_model()

impedancefitter.double_shell.**eps_cell_double_shell**(*omega*, *km*, *em*, *kcp*, *ecp*, *ene*, *kne*, *knp*, *enp*, *dm*, *Rc*, *dn*, *Rn*)

Double Shell model.

Parameters

- **omega** (numpy.ndarray, double) – list of frequencies value for c_0, unit capacitance in pF
- **em** (*double*) – membrane permittivity,membrane permittivity, value for ε_{m}
- **km** (*double*) – membrane conductivity, value for σ_{m} in μS/m
- **ecp** (*double*) – cytoplasm permittivity, value for $\varepsilon_{\mathrm{cp}}$
- **kcp** (*double*) – cytoplasm conductivity, value for σ_{cp}
- **ene** (*double*) – nuclear envelope permittivity, value for $\varepsilon_{\mathrm{ne}}$
- **kne** (*double*) – nuclear envelope conductivity, value for σ_{ne} in mS/m
- **enp** (*double*) – nucleoplasm permittivity, value for $\varepsilon_{\mathrm{np}}$
- **knp** (*double*) – nucleoplasm conductivity, value for σ_{np}
- **dm** (*double*) – membrane thickness, value for d_{m}
- **Rc** (*double*) – cell radius, value for R_{c}
- **dn** (*double*) – nuclear envelope thickness, value for d_{n}
- **Rn** (*double*) – nucleus radius, value for R_{n}

Returns Permittivity array

Return type numpy.ndarray, complex

5.5 Inductance circuits

impedancefitter.loss.**Z_in**(*omega*, *L*, *R*)
Lead inductance of wires connecting DUT.

Described for instance in [Kordzadeh2016].

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **L** (*double*) – inductance
- **R** (*double*) – resistance

Returns Impedance array

Return type numpy.ndarray, complex

References

Notes

As mentioned in [Kordzadeh2016], the unit of the inductance is nH.

impedancefitter.loss.**Z_loss**(*omega*, *L*, *C*, *R*)
Impedance for high loss materials, where LCR are in parallel.

Described for instance in [Kordzadeh2016].

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **L** (*double*) – inductance
- **C** (*double*) – capacitance
- **R** (*double*) – resistance

Returns Impedance array

Return type numpy.ndarray, complex

Notes

As mentioned in [Kordzadeh2016], the unit of the capacitance is pF and the unit of the inductance is nH.

impedancefitter.loss.**Z_skin**(*omega*, *L*, *Rb*, *gamma*)
Lead inductance of wires connecting DUT considering skin effect.

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **L** (*double*) – inductance
- **Rb** (*double*) – resistance of wire, about mOhm
- **gamma** (*double*) – exponent (should be about 0.5 probably)

Returns Impedance array

Return type numpy.ndarray, complex

References

Notes

Described for instance in [Levitskaya2000]. The idea is to take frequency-dependent resistance.

The unit of the inductance is nH. The wire resistance is about mOhm. The impedance is computed by

$$Z = R_b \omega^\gamma + j\omega L$$

Note that the frequency omega in this formula is in MHz.

5.6 CPE circuits

impedancefitter.cpe.**cpe_ct_model**(*omega*, *k*, *alpha*, *Rct*)

Constant Phase Element in parallel with charge transfer resistance.

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **k** (*double*) – CPE factor
- **alpha** (*double*) – CPE phase
- **Rct** (*double*) – charge transfer resistance

Returns Impedance array

Return type numpy.ndarray, complex

See also:

impedancefitter.cpe.cpe_model()

impedancefitter.cpe.**cpe_ct_tissue_model**(*omega*, *k*, *alpha*, *Rct*)

Constant Phase Element in parallel with charge transfer resistance.

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **k** (*double*) – CPE factor
- **alpha** (*double*) – CPE phase
- **Rct** (*double*) – charge transfer resistance

Returns Impedance array

Return type numpy.ndarray, complex

See also:

impedancefitter.cpe.cpe_model()

impedancefitter.cpe.**cpe_ct_w_model**(*omega*, *k*, *alpha*, *Rct*, *Aw*)

Constant Phase Element in parallel with charge transfer resistance, which is in series with Warburg element.

Parameters

- **omega** (numpy.ndarray) – List of frequencies.

- **k** (*double*) – CPE factor
- **alpha** (*double*) – CPE phase
- **Rct** (*double*) – charge transfer resistance
- **Aw** (*double*) – Warburg coefficient

Returns Impedance array

Return type numpy.ndarray, complex

See also:

impedancefitter.cpe.cpe_model()

impedancefitter.cpe.**cpe_model**(*omega*, *k*, *alpha*)
Constant Phase Element.

$$Z_{\mathrm{CPE}} = k(j\omega)^{-\alpha}$$

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **k** (*double*) – CPE factor
- **alpha** (*double*) – CPE phase

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.cpe.**cpe_onset_model**(*omega*, *f0*, *nu*, *Z0*)
Alternative Constant Phase Element Formulation.

Notes

$$Z_{\mathrm{CPE}} = Z_0 \left(j \frac{\omega}{2\pi f_0} \right)^{-\nu}$$

See for example [Ishai2012].

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **k** (*double*) – CPE factor
- **alpha** (*double*) – CPE phase

Returns Impedance array

Return type numpy.ndarray, complex

References

impedancefitter.cpe.**cpetissue_model**(*omega*, *k*, *alpha*)
Constant Phase Element.

$$Z_{\mathrm{CPE}} = k(j\omega)^{-\alpha}$$

Parameters

- **omega** (numpy.ndarray) – List of frequencies.
- **k** (*double*) – CPE factor
- **alpha** (*double*) – CPE phase

Returns Impedance array

Return type numpy.ndarray, complex

5.7 RC circuits

impedancefitter.RC.**RC_model**(*omega*, *Rd*, *Cd*)
Simple RC model, resistor in parallel with capacitor.

Parameters

- **omega** (*double or array of double*) – list of frequencies
- **Rd** (*complex*) – Resistance.
- **Cd** (*double*) – Capacitance

Notes

> **Warning:** *Cd* is in pF!

Returns Impedance array

Return type numpy.ndarray, complex

impedancefitter.RC.**drc_model**(*omega*, *RE*, *tauE*, *alpha*, *beta*)
Distributed RC circuit.

Parameters

- **omega** (*double or array of double*) – list of frequencies
- **RE** (*double*) – resistance
- **tauE** (*double*) – relaxation time, in ns
- **alpha** (*double*) – Cole-Cole exponent
- **beta** (*double*) – DRC exponent, beta = 1 equals Cole-Cole model

Returns Impedance array

Return type numpy.ndarray, complex

Notes

Described for example in [Emmert2011].

Warning: The time constant tauE is in ns!

References

impedancefitter.RC.**rc_model**(*omega*, *c0*, *kdc*, *eps*)

Simple RC model to obtain dielectric properties.

Parameters

- **omega** (*double or array of double*) – list of frequencies
- **c0** (*double*) – unit capacitance in pF
- **eps** (*double*) – relative permittivity
- **kdc** (*double*) – conductivity

Returns Impedance array

Return type numpy.ndarray, complex

Notes

Warning: C0 is in pF!

impedancefitter.RC.**rc_tau_model**(*omega*, *Rk*, *tauk*)

Compute RC model without explicit capacitance.

Parameters

- **omega** (*double or array of double*) – list of frequencies
- **Rk** (*double*) – resistance
- **tauk** (*double*) – relaxation time

Notes

Returns Impedance array

Return type numpy.ndarray, complex

Notes

Used for example in the Lin-KK test [Schoenleber2014].

The impedance reads

$$Z = \frac{R_\mathrm{k}}{1 + j\omega\tau_\mathrm{k}}$$

5.8 Randles circuits

impedancefitter.randles.**Z_randles**(*omega*, *Rct*, *Rs*, *Aw*, *C0*)

Randles circuit.

Parameters

- **omega** (*double or array of double*) – list of frequencies
- **Rct** (*double*) – Resistance in series with Warburg element, e.g. charge transfer resistance
- **Rs** (*double*) – Resistance of the DUT, e.g. electrolyte resistance
- **Aw** (*double*) – Warburg coefficient
- **C0** (*double*) – capacitance

Returns Impedance array

Return type numpy.ndarray, complex

Notes

Function holding the Randles equation with capacitor in parallel to resistor in series with Warburg element and another resistor in series.

Equations for calculations:

Impedance of resistor and Warburg element:

$$Z_\mathrm{RW} = R_0 + A_\mathrm{W}\frac{1-j}{\sqrt{\omega}}$$

Impedance of capacitor:

$$Z_C = (j\omega C_0)^{-1}$$

$$Z_\mathrm{fit} = R_s + \frac{Z_C Z_\mathrm{RW}}{Z_C + Z_\mathrm{RW}}$$

impedancefitter.randles.**Z_randles_CPE**(*omega*, *Rct*, *Rs*, *Aw*, *k*, *alpha*)

Randles circuit with CPE instead of capacitor.

Parameters

- **omega** (*double or array of double*) – list of frequencies
- **Rct** (*double*) – Resistance in series with Warburg element, e.g. charge transfer resistance
- **Rs** (*double*) – Resistance of the DUT, e.g. electrolyte resistance
- **Aw** (*double*) – Warburg coefficient

- **k** (*double*) – CPE coefficient
- **alpha** (*double*) – CPE exponent

Returns Impedance array

Return type `numpy.ndarray`, complex

See also:

`Z_randles()`, *`impedancefitter.elements.Z_CPE()`*

CHAPTER

SIX

UTILITITES

impedancefitter.utils.**KK_integral_transform**(*omega*, *Z*)

Kramers-Kronig integral transform.

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array

Returns The transformed impedance array.

Return type numpy.ndarray, complex

Notes

Implementation following [Urquidi1990]

impedancefitter.utils.**available_file_format**()

List available file formats.

Currently available:

XLSX and **CSV**:

The file is structured like: frequency, real part of impedance, imaginary part of impedance. There may be many different sets of impedance data, i.e. there may be more columns with the real and the imaginary part. Then, the frequencies column must not be repeated. In fact, the number of columns equals the number of impedance data sets plus one (for the frequency).

Note: A single header line is needed in a CSV and XLSX file. It may contain for example *frequency, Real Part, Imag Part*. Otherwise the read-in function will fail.

CSV_E4980AL:

Read in data that is structured in 5 columns: frequency, real part, imaginary part of the impedance, voltage, current

---

Note: There is always only one data set in a file.

TXT:

These files contain frequency, real and imaginary part of the impedance (i.e., 3 columns). The TXT files may contain two traces; only one of them is read in. For TXT files you can specify the number of rows to skip. Moreover, the file ending is not strictly enforced here.

See also:

impedancefitter.fitter.Fitter

impedancefitter.utils.**available_models**()
return list of available models

impedancefitter.utils.**check_parameters**(*bufdict*)
Check parameters for physical correctness.

Parameters **bufdict** (*dict*) – Contains all parameters and their values

Notes

All parameters are forced to be greater or equal zero. There are only two exceptions.

impedancefitter.utils.**convert_diel_properties_to_impedance**(*omega*, *eps_r*, *sigma*, *c0*)
Return impedance from dielectric properties.

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **eps_r** (numpy.ndarray, double) – relative permittivity
- **sigma** (numpy.ndarray, double) – conductivity in S/m
- **c0** (*double*) – unit capacitance of device

Returns impedance array

Return type numpy.ndarray, complex

Notes

Use that the impedance is

$$Z = (j\omega\varepsilon_\mathrm{r}^* c_0)^{-1},$$

where ε_r^* is the relative complex permittivity (see for instance [Grant1958] for further explanation). Note that the vacuum permittivity ε_0 is contained in c_0.

In the function, the variable *epsc* describes the term

$$\omega\varepsilon_\mathrm{r}^*$$

impedancefitter.utils.**draw_scheme**(*modelname*, *show=True*, *save=False*)
Show (and save) SchemDraw drawing.

Parameters

- **modelname** (*str*) – String representation of the equivalent circuit.
- **show** (*bool, optional*) – Show scheme in matplotlib window.
- **save** (*bool, optional*) – Save scheme to file. File is called *scheme.svg*.

impedancefitter.utils.**get_equivalent_circuit_model**(*modelname*, *logscale=False*, *diel=False*)
Get LMFIT CompositeModel.

Parameters

- **modelname** (*str*) – String representation of the equivalent circuit.
- **logscale** (*bool*) – Convert to logscale.
- **diel** (*bool*) – Convert to complex permittivity and fit this instead of impedance.

Returns the final model of the entire circuit

Return type `lmfit.model.CompositeModel` or `lmfit.model.Model`

Notes

The parser is based on Pyparsing. It is sensitive towards extra *(* or *)* or +. Thus, keep the circuit simple.

`impedancefitter.utils.`**`get_labels`**(*params*)
return the labels for every parameter in LaTex code.

Parameters **params** (*list of string*) – list with parameters names (possible prefixes included)

Returns **labels** – dictionary with parameter names as keys and LaTex code as values.

Return type dict

`impedancefitter.utils.`**`return_diel_properties`**(*omega*, *Z*, *c0*)
Return relative permittivity and conductivity from impedance spectrum in cavity with known unit capacitance.

Notes

Use that the impedance is

$$Z = (j\omega\varepsilon_\mathrm{r}^* c_0)^{-1},$$

where ε_r^* is the relative complex permittivity (see for instance [Grant1958] for further explanation). Note that the vacuum permittivity ε_0 is contained in c_0.

When the unit capacitance c_0 of the device is known, a direct mapping from impedance to relative complex permittivity is possible:

$$\varepsilon_\mathrm{r}^* = (j\omega Z c_0)^{-1} = \frac{\varepsilon^*}{\varepsilon_0}$$

The unit capacitance (or air capacitance) of the device is defined as

$$c_0 = \frac{\varepsilon_0 A}{d}$$

for a parallel-plate capacitor with electrode area *A* and spacing *d* but can also be measured in a calibration step.

The relative permittivity is the real part of ε_r^* and the conductivity is the negative imaginary part times the frequency and the vacuum permittivity.

Parameters

- **omega** (`numpy.ndarray`, double) – frequency array
- **Z** (`numpy.ndarray`, complex) – impedance array
- **c0** (*double*) – unit capacitance of device

Returns

- **eps_r** (`numpy.ndarray`, double) – relative permittivity
- **conductivity** (`numpy.ndarray`, double) – conductivity in S/m

References

impedancefitter.utils.**return_dielectric_modulus**(*omega*, *Z*, *c0*)
Return dielectric modulus

Notes

The dielectric modulus is $M = 1/\varepsilon_r^*$. See [Bordi2001] for further explanation.

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array
- **c0** (*double*) – unit capacitance of device

Returns

- **ReM** (numpy.ndarray, double) – real part of modulus
- **ImM** (numpy.ndarray, double) – imaginary part of modulus

References

impedancefitter.utils.**save_impedance**(*omega*, *impedance*, *format='CSV'*, *filename='impedance'*)
Save impedance to CSV or XLSX file.

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **impedance** (numpy.ndarray, complex) – impedance array
- **format** (*str*) – use either CSV or XLSX format. Based on the format, the correct ending is chosen.
- **filename** (*str*) – specify a filename (without ending!). the default is impedance.csv or impedance.xlsx

impedancefitter.utils.**set_parameters**(*model*, *parameterdict=None*, *emcee=False*, *weighting_model=False*)

Parameters

- **model** (lmfit.model.Model) – The LMFIT model used for fitting.
- **parameterdict** (*dict, optional*) – A dictionary containing parameters for model with *min*, *max*, *vary* info for LMFIT. If it is None (default), the parameters are read in from a yaml-file.
- **emcee** (*bool, optional*) – if emcee is used, an additional *__lnsigma* parameter will be set
- **weighting_model** (*bool, optional*) – if a weighting model is used, the variance will be fit as well

Returns **params** – LMFIT Parameters object.

Return type lmfit.parameter.Parameters

CHAPTER

SEVEN

PLOTTING

7.1 API Reference

impedancefitter.plotting.**emcee_plot**(*res*, *clustered=False*, ***corner_kwargs*)

Create corner plot.

Parameters

- **res** (*dict*) – Dictionary containing values and flatchain
- **corner_kwargs** (*dict, optional*) – Dictionary with further corner plot options

impedancefitter.plotting.**plot_admittance**(*omega*, *Z*, *title=''*, *Z_fit=None*, *show=True*, *save=False*, *Z_comp=None*, *labels=['Data', 'Best fit', 'Init fit']*, *residual='parts'*, *sign=False*, *Zlog=False*, *append=False*, *limits_residual=None*, *omega_fit=None*, *omega_comp=None*, *legend=True*, *compare=True*)

Plot the admittance and compare it to data 1/*Z*.

Generates 4 subplots showing the real and imaginary parts over the frequency; a Nyquist plot of real and negative imaginary part and the relative differences of real and imaginary part as well as absolute value of admittance.

Parameters

- **omega** (numpy.ndarray, double) – Frequency array
- **Z** (numpy.ndarray, complex) – Impedance array, experimental data or data to compare to.
- **Z_fit** (numpy.ndarray, complex) – Impedance array, fit result. If provided, the difference between data and fit will be shown.
- **title** (*str*) – Title of plot.
- **show** (*bool, optional*) – Show figure (default is True).
- **save** (*bool, optional*) – Save figure to pdf (default is False). Name of figure starts with *title* and ends with *_admittance_overview.pdf*.
- **Z_comp** (numpy.ndarray, complex, optional) – Complex-valued impedance array. Might be used to compare the properties of two data sets.
- **labels** (*list*) – List of labels for three plots. Must have length 3 always. Is ordered like: *[Z, Z_fit, Z_comp]*
- **residual** (*str*) – Plot relative difference w.r.t. real and imaginary part if *parts*. Plot relative difference w.r.t. absolute value if *absolute*. Plot difference (residual) if *diff*.

- **sign** (*bool, optional*) – Use sign of residual. Default is False, i.e. absolute value is plotted.
- **Zlog** (*bool, optional*) – Log-scale of impedance
- **append** (*bool, optional*) – Decide if you want to show plot or add line to existing plot.
- **omega_fit** (numpy.ndarray, double, optional) – Frequency array, provide only if fitted impedance was evaluated at different frequencies than the experimental data
- **omega_comp** (numpy.ndarray, double, optional) – Frequency array, provide only if fitted impedance was evaluated at different frequencies than the experimental data
- **legend** (*str, optional*) – Choose if a legend should be shown. Recommended to switch to False when using large datasets.
- **compare** (*bool, optional*) – Choose if the difference between fit and data should be computed.

impedancefitter.plotting.**plot_bode**(*omega*, *Z*, *title=''*, *Z_fit=None*, *show=True*, *save=False*, *Z_comp=None*, *labels=['Data', 'Best fit', 'Init fit']*, *append=False*, *legend=True*)

Bode plot of impedance.

Plots phase and log of magnitude over log of frequency.

Parameters

- **omega** (numpy.ndarray, double) – Frequency array
- **Z** (numpy.ndarray, complex) – Impedance array, experimental data or data to compare to.
- **Z_fit** (numpy.ndarray, complex) – Impedance array, fit result. If provided, the difference between data and fit will be shown.
- **title** (*str*) – Title of plot.
- **show** (*bool, optional*) – Show figure (default is True).
- **save** (*bool, optional*) – Save figure to pdf (default is False). Name of figure starts with *title*.
- **Z_comp** (numpy.ndarray, complex, optional) – Complex-valued impedance array. Might be used to compare the properties of two data sets.
- **labels** (*list*) – List of labels for three plots. Must have length 3 always. Is ordered like: *[Z, Z_fit, Z_comp]*
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_bode_plot.pdf*.
- **append** (*bool, optional*) – Decide if you want to show plot or add line to existing plot.
- **legend** (*str, optional*) – Choose if a legend should be shown. Recommended to switch to False when using large datasets.

impedancefitter.plotting.**plot_cole_cole**(*omega*, *Z*, *c0*, *Z_comp=None*, *append=False*, *legend=True*, *markers=[None, None]*, *title=''*, *show=True*, *save=False*, *labels=None*, *limits=None*)

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array
- **c0** (*double*) – unit capacitance of device

- **Z_comp** (numpy.ndarray, complex, optional) – complex-valued impedance array. Might be used to compare the properties of two data sets.
- **title** (*str, optional*) – title of plot. Default is an empty string.
- **show** (*bool, optional*) – show figure (default is True)
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_dielectric_properties.pdf*.
- **logscale** (*str, optional*) – Decide what you want to plot using log scale. Possible are *permittivity*, *conductivity* and *both*
- **labels** (*list, optional*) – Give custom labels. Needs to be a list of length 2.

impedancefitter.plotting.**plot_compare_to_data**(*omega*, *Z*, *Z_fit*, *subplot=None*, *title=''*, *show=True*, *save=False*, *residual='parts'*, *sign=False*, *limits=None*, *impedance_threshold=1.0*, *legend=True*)

plots the difference of the fitted function to the data

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array, experimental data or data to compare to
- **Z_fit** (numpy.ndarray, complex) – impedance array, fit result
- **subplot** (*optional*) – decide whether it is a new figure or a subplot. Default is None (yields new figure). Otherwise it can be an integer to denote the subfigure.
- **title** (*str, optional*) – title of plot. Default is an empty string.
- **show** (*bool, optional*) – show figure (default is True). Only has an effect when *subplot* is None.
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_relative_difference_to_data.pdf* or *_difference_to_data.pdf*.
- **relative** (*str, optional*) – Plot relative difference if True, else plot residual (i.e. just difference).
- **sign** (*bool, optional*) – Use sign of residual. Default is False, i.e. absolute value is plotted.
- **residual** (*str*) – Plot relative difference w.r.t. real and imaginary part if *parts*. Plot relative difference w.r.t. absolute value if *absolute*. Plot difference (residual) if *diff*.
- **limits** (*list, optional*) – List with entries *[bottom, top]* for y-axis of residual plot.
- **impedance_threshold** (*double, optional*) – Threshold for impedance around 0, which is disregarded in the relative differences plot. Default is that impedances, with an absolute value less than 0 are not considered.
- **legend** (*str, optional*) – Choose if a legend should be shown. Recommended to switch to False when using large datasets.

Notes

When computing the relative difference, impedances between -1 and 1 Ohm are not considered since they might lead to a blow up of the relative difference (close to division by 0). Instead of this quantitative measure, qualitative checks should be done.

impedancefitter.plotting.**plot_complex_permittivity**(*omega*, *Z*, *c0*, *Z_comp=None*, *title=''*, *show=True*, *save=False*, *logscale='permittivity'*, *labels=None*)

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array
- **c0** (*double*) – unit capacitance of device
- **Z_comp** (numpy.ndarray, complex, optional) – complex-valued impedance array. Might be used to compare the properties of two data sets.
- **title** (*str, optional*) – title of plot. Default is an empty string.
- **show** (*bool, optional*) – show figure (default is True)
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_dielectric_properties.pdf*.
- **logscale** (*str, optional*) – Decide what you want to plot using log scale. Possible are *permittivity*, *loss* and *both*
- **labels** (*list, optional*) – Give custom labels. Needs to be a list of length 2.

impedancefitter.plotting.**plot_dielectric_dispersion**(*omega*, *Z*, *c0*, *Z_comp=None*, *title=''*, *show=True*, *save=False*, *logscale='permittivity'*, *labels=None*, ***plotkwargs*)

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array
- **c0** (*double*) – unit capacitance of device
- **Z_comp** (numpy.ndarray, complex, optional) – complex-valued impedance array. Might be used to compare the properties of two data sets.
- **title** (*str, optional*) – title of plot. Default is an empty string.
- **show** (*bool, optional*) – show figure (default is True)
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_dielectric_properties.pdf*.
- **logscale** (*str, optional*) – Decide what you want to plot using log scale. Possible are *permittivity*, *conductivity* and *both*
- **labels** (*list, optional*) – Give custom labels. Needs to be a list of length 2.

impedancefitter.plotting.**plot_dielectric_modulus**(*omega*, *Z*, *c0*, *Z_comp=None*, *title=''*, *show=True*, *save=False*, *logscale=None*, *labels=None*)

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array
- **c0** (*double*) – unit capacitance of device
- **Z_comp** (numpy.ndarray, complex, optional) – complex-valued impedance array. Might be used to compare the properties of two data sets.
- **title** (*str, optional*) – title of plot. Default is an empty string.
- **show** (*bool, optional*) – show figure (default is True)
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_dielectric_properties.pdf*.
- **logscale** (*str, optional*) – Decide what you want to plot using log scale. Possible are *ReM*, *ImM* and *both*
- **labels** (*list, optional*) – Give custom labels. Needs to be a list of length 2.

impedancefitter.plotting.**plot_dielectric_properties**(*omega*, *Z*, *c0*, *Z_comp=None*, *title=''*, *show=True*, *save=False*, *logscale='permittivity'*, *labels=None*, *append=False*, *markers=[None, None]*, *legend=True*, *limits=None*, ***plotkwargs*)

Parameters

- **omega** (numpy.ndarray, double) – frequency array
- **Z** (numpy.ndarray, complex) – impedance array
- **c0** (*double*) – unit capacitance of device
- **Z_comp** (numpy.ndarray, complex, optional) – complex-valued impedance array. Might be used to compare the properties of two data sets.
- **title** (*str, optional*) – title of plot. Default is an empty string.
- **show** (*bool, optional*) – show figure (default is True)
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_dielectric_properties.pdf*.
- **logscale** (*str, optional*) – Decide what you want to plot using log scale. Possible are *permittivity*, *conductivity* and *both*
- **labels** (*list, optional*) – Give custom labels. Needs to be a list of length 2.
- **append** (*bool, optional*) – Decide if you want to show plot or add line to existing plot.
- **legend** (*bool, optional*) – Switch legend on/off

impedancefitter.plotting.**plot_impedance**(*omega*, *Z*, *title=''*, *Z_fit=None*, *show=True*, *save=False*, *Z_comp=None*, *labels=['Data', 'Best fit', 'Init fit']*, *residual='parts'*, *sign=False*, *Zlog=False*, *append=False*, *limits_residual=None*, *omega_fit=None*, *omega_comp=None*, *legend=True*, *compare=True*)

Plot the *result* and compare it to data *Z*.

Generates 4 subplots showing the real and imaginary parts over the frequency; a Nyquist plot of real and negative imaginary part and the relative differences of real and imaginary part as well as absolute value of impedance.

Parameters

- **omega** (numpy.ndarray, double) – Frequency array
- **Z** (numpy.ndarray, complex) – Impedance array, experimental data or data to compare to.
- **Z_fit** (numpy.ndarray, complex) – Impedance array, fit result. If provided, the difference between data and fit will be shown.
- **title** (*str*) – Title of plot.
- **show** (*bool, optional*) – Show figure (default is True).
- **save** (*bool, optional*) – Save figure to pdf (default is False). Name of figure starts with *title* and ends with *_impedance_overview.pdf*.
- **Z_comp** (numpy.ndarray, complex, optional) – Complex-valued impedance array. Might be used to compare the properties of two data sets.
- **labels** (*list*) – List of labels for three plots. Must have length 3 always. Is ordered like: *[Z, Z_fit, Z_comp]*
- **residual** (*str*) – Plot relative difference w.r.t. real and imaginary part if *parts*. Plot relative difference w.r.t. absolute value if *absolute*. Plot difference (residual) if *diff*.
- **sign** (*bool, optional*) – Use sign of residual. Default is False, i.e. absolute value is plotted.
- **Zlog** (*bool, optional*) – Log-scale of impedance
- **append** (*bool, optional*) – Decide if you want to show plot or add line to existing plot.
- **omega_fit** (numpy.ndarray, double, optional) – Frequency array, provide only if fitted impedance was evaluated at different frequencies than the experimental data
- **omega_comp** (numpy.ndarray, double, optional) – Frequency array, provide only if fitted impedance was evaluated at different frequencies than the experimental data
- **legend** (*str, optional*) – Choose if a legend should be shown. Recommended to switch to False when using large datasets.
- **compare** (*bool, optional*) – Choose if the difference between fit and data should be computed.

impedancefitter.plotting.**plot_resistance_capacitance**(*omega*, *Z*, *title=''*, *Z_fit=None*, *show=True*, *save=False*, *Z_comp=None*, *labels=['Data', 'Best fit', 'Init fit']*, *append=False*, *legend=True*)

R-C plot of impedance.

Plots phase and log of magnitude over log of frequency.

Parameters

- **omega** (numpy.ndarray, double) – Frequency array
- **Z** (numpy.ndarray, complex) – Impedance array, experimental data or data to compare to.
- **Z_fit** (numpy.ndarray, complex) – Impedance array, fit result. If provided, the difference between data and fit will be shown.

- **title** (*str*) – Title of plot.
- **show** (*bool, optional*) – Show figure (default is True).
- **save** (*bool, optional*) – Save figure to pdf (default is False). Name of figure starts with *title*.
- **Z_comp** (numpy.ndarray, complex, optional) – Complex-valued impedance array. Might be used to compare the properties of two data sets.
- **labels** (*list*) – List of labels for three plots. Must have length 3 always. Is ordered like: *[Z, Z_fit, Z_comp]*
- **save** (*bool, optional*) – save figure to pdf (default is False). Name of figure starts with *title* and ends with *_bode_plot.pdf*.
- **append** (*bool, optional*) – Decide if you want to show plot or add line to existing plot.
- **legend** (*str, optional*) – Choose if a legend should be shown. Recommended to switch to False when using large datasets.

impedancefitter.plotting.**plot_uncertainty**(*omega*, *Zdata*, *Z*, *Z1*, *Z2*, *sigma*, *show=True*, *model=None*)

Plot best fit with uncertainty interval.

Parameters

- **Zdata** (numpy.ndarray, complex) – impedance array of experimental data
- **Z** (numpy.ndarray, complex) – impedance array of best fit
- **Z1** (numpy.ndarray, complex) – impedance array of upper uncertainty limit
- **Z2** (numpy.ndarray, complex) – impedance array of lower uncertainty limit
- **sigma** (*double*) – confidence level
- **show** (*bool, optional*) – show figure (default is True)
- **model** (*int, optional*) – numbering of model for sequential plotting

CHAPTER

EIGHT

OVERVIEW

Impedance spectroscopy (IS) is a great tool to analyse the behaviour of an electrical circuit, to characterise the response of a sample (e.g. biological tissue), to determine the dielectric properties of a sample, and much more [Barsoukov2018], [Orazem2017].

In IS, often (complex) non-linear least squares is used for parameter estimation of equivalent circuit models. ImpedanceFitter is a software that facilitates parameter estimation for arbitrary equivalent circuit models. The equivalent circuit may comprise different standard elements or other models that have been formulated in the context of impedance spectroscopy. The unknown parameters are found by fitting the model to experimental impedance data. The underlying fitting software is LMFIT [Newville2019], which offers an interface to different optimization and curve-fitting methods going beyond standard least-squares. ImpedanceFitter allows one to build a custom equivalent circuit, fit an arbitrary amount of data sets and perform statistical analysis of the results using OpenTurns [Baudin2017].

CHAPTER

NINE

QUICKSTART

Install the package either using pip

```
pip install impedancefitter
```

or from source

```
pip install .
```

Then continue with the *Examples*.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Hou2012] Hou, F., Goodman, J., Hogg, D. W., Weare, J., & Schwab, C. (2012). An affine-invariant sampler for exoplanet fitting and discovery in radial velocity data. Astrophysical Journal, 745(2). https://doi.org/10.1088/0004-637X/745/2/198

[Schoenleber2014] Schönleber, M., Klotz, D., & Ivers-Tiffée, E. (2014). A Method for Improving the Robustness of linear Kramers-Kronig Validity Tests. Electrochimica Acta, 131, 20–27. https://doi.org/10.1016/j.electacta.2014.01.034

[Gabriel1996] Gabriel, S., Lau, R. W., & Gabriel, C. (1996). The dielectric properties of biological tissues: III. Parametric models for the dielectric spectrum of tissues. Physics in Medicine and Biology, 41(11), 2271–2293. https://doi.org/10.1088/0031-9155/41/11/003

[Schwan1957] Schwan, H. P. (1957). Electrical properties of tissue and cell suspensions. Advances in biological and medical physics (Vol. 5). ACADEMIC PRESS INC. https://doi.org/10.1016/b978-1-4832-3111-2.50008-0

[Sabuncu2012] Sabuncu, A. C., Zhuang, J., Kolb, J. F., & Beskok, A. (2012). Microfluidic impedance spectroscopy as a tool for quantitative biology and biotechnology. Biomicrofluidics, 6(3). https://doi.org/10.1063/1.4737121

[Raicu1999] Raicu, V. (1999). Dielectric dispersion of biological matter: Model combining Debye-type and "universal" responses. Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics, 60(4), 4677–4680. https://doi.org/10.1103/PhysRevE.60.4677

[Stoneman2007] Stoneman, M. R., Kosempa, M., Gregory, W. D., Gregory, C. W., Marx, J. J., Mikkelson, W., … Raicu, V. (2007). Correction of electrode polarization contributions to the dielectric properties of normal and cancerous breast tissues at audio/radiofrequencies. Physics in Medicine and Biology, 52(22), 6589–6604. https://doi.org/10.1088/0031-9155/52/22/003

[Feldman2003] Feldman, Y., Ermolina, I., & Hayashi, Y. (2003). Time domain dielectric spectroscopy study of biological systems. IEEE Transactions on Dielectrics and Electrical Insulation, 10, 728–753. https://doi.org/10.1109/TDEI.2003.1237324

[Ermolina2000] Ermolina, I., Polevaya, Y., & Feldman, Y. (2000). Analysis of dielectric spectra of eukaryotic cells by computer modeling. European Biophysics Journal, 29(2), 141–145. https://doi.org/10.1007/s002490050259

[Kordzadeh2016] Kordzadeh, A., & De Zanche, N. (2016). Permittivity measurement of liquids, powders, and suspensions using a parallel-plate cell. Concepts in Magnetic Resonance Part B: Magnetic Resonance Engineering, 46(1), 19–24. https://doi.org/10.1002/cmr.b.21318

[Levitskaya2000] Levitskaya, T. M., & Sternberg, B. K. (2000). Laboratory measurement of material electrical properties: extending the application of lumped-circuit equivalent models to 1 GHz. Radio Science, 35(2), 371–383. https://doi.org/10.1029/1999RS002186

[Ishai2012] Ishai, P. Ben, Sobol, Z., Nickels, J. D., Agapov, A. L., & Sokolov, A. P. (2012). An assessment of comparative methods for approaching electrode polarization in dielectric permittivity measurements. Review of Scientific Instruments, 83(8). https://doi.org/10.1063/1.4746992

[Emmert2011] Emmert, S., Wolf, M., Gulich, R., Krohns, S., Kastner, S., Lunkenheimer, P., & Loidl, A. (2011). Electrode polarization effects in broadband dielectric spectroscopy. European Physical Journal B, 83(2), 157–165. https://doi.org/10.1140/epjb/e2011-20439-8

[Urquidi1990] Urquidi-Macdonald, M., Real, S., & Macdonald, D. D. (1990). Applications of Kramers-Kronig transforms in the analysis of electrochemical impedance data-III. Stability and linearity. Electrochimica Acta, 35(10), 1559–1566. https://doi.org/10.1016/0013-4686(90)80010-L

[Grant1958] Grant, F. A. (1958). Use of complex conductivity in the representation of dielectric phenomena. Journal of Applied Physics, 29(1), 76–80. https://doi.org/10.1063/1.1722949

[Bordi2001] Bordi, F., & Cametti, C. (2001). Occurrence of an intermediate relaxation process in water-in-oil microemulsions below percolation: The electrical modulus formalism. Journal of Colloid and Interface Science, 237(2), 224–229. https://doi.org/10.1006/jcis.2001.7456

[Barsoukov2018] Barsoukov, E., & Macdonald, J. R. (Eds.). (2018). Impedance Spectroscopy: Theory, Experiment, and Applications. (3rd ed.). Hoboken, NJ: John Wiley & Sons, Inc. https://doi.org/10.1002/9781119381860

[Orazem2017] Orazem, M. E., & Tribollet, B. (2017). Electrochemical Impedance Spectroscopy. (2nd ed.). Hoboken, NJ: John Wiley & Sons, Inc. https://doi.org/10.1002/9781119363682

[Newville2019] Newville, M., Otten, R., Nelson, A., Ingargiola, A., Stensitzki, T., Allan, D., Fox, A., Carter, F., Michał, Pustakhod, D., Ram, Y., Glenn, Deil, C., Stuermer, Beelen, A., Frost, O., Zobrist, N., Pasquevich, G., Hansen, A.L.R., Spillane, T., Caldwell, S., Polloreno, A., andrewhannum, Zimmermann, J., Borreguero, J., Fraine, J., deep-42-thought, Maier, B.F., Gamari, B., & Almarza, A. (2020, May 7). lmfit/lmfit-py 1.0.1 (Version 1.0.1). Zenodo. http://doi.org/10.5281/zenodo.3814709

[Baudin2017] Baudin, M., Dutfoy, A., Looss, B., & Popelin, A. L. (2017). OpenTURNS: An industrial software for uncertainty quantification in simulation. In Handbook of Uncertainty Quantification (pp. 2001–2038). https://doi.org/10.1007/978-3-319-12385-1_64

PYTHON MODULE INDEX

i

INDEX

I

K

L

M

P

R

S

V

Z