ImpedanceFitter Documentation

Release 2.0.0

Julius Zimmermann, Leonard Thiele

May 11, 2020

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. 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

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

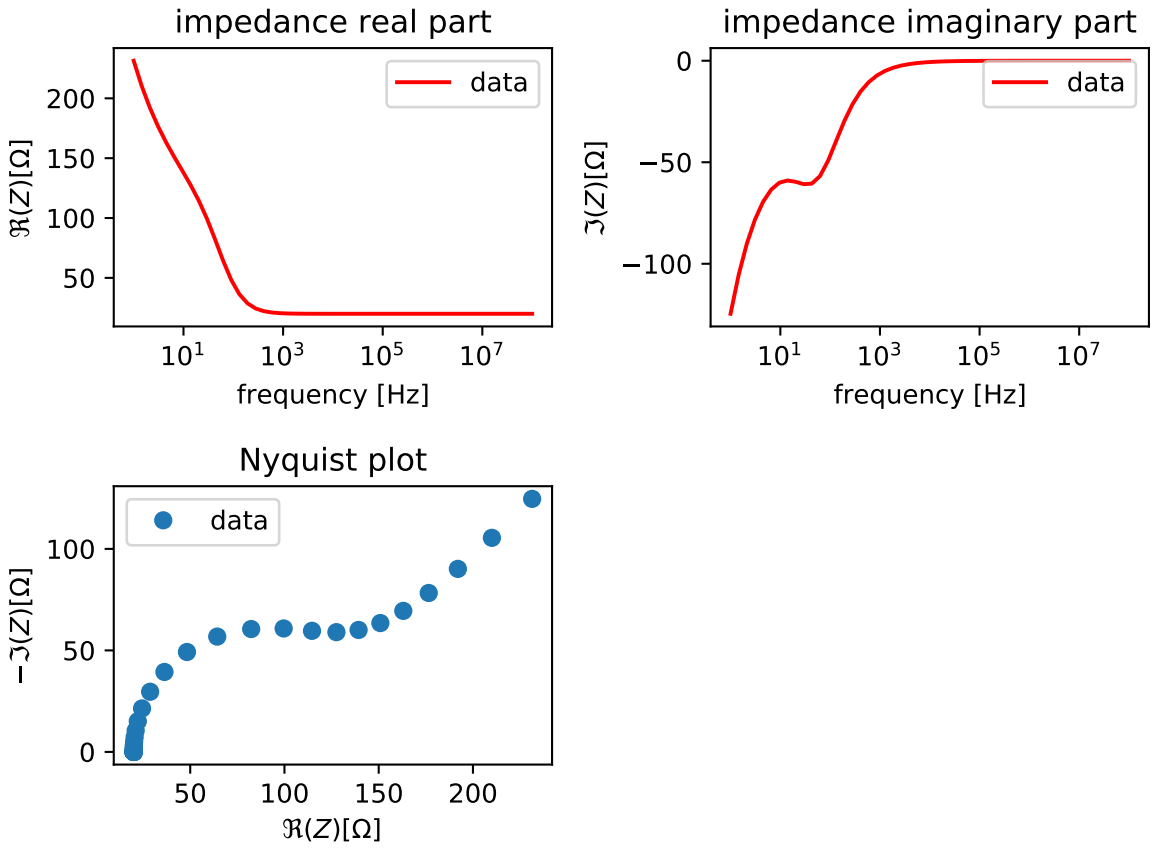
```
model = 'Randles'
lmfit_model = impedancefitter.get_equivalent_circuit_model(model)
Z = lmfit_model.eval(omega=2. * numpy.pi * frequencies,
                      Rct=Rct, Rs=Rs,
                      C0=C0, Aw=Aw)
```

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*.

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 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.

```
import impedancefitter
import numpy
import os
import pandas
from matplotlib import rcParams
```

(continues on next page)

(continued from previous page)

```
rcParams['figure.figsize'] = [15, 10]


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


# generate model by user-defined circuit
model = 'R_s + parallel(R_ct + W, C)'
lmfit_model = impedancefitter.get_equivalent_circuit_model(model)
# generate data (without noise)
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)

# initialise fitter with verbose output
fitter = impedancefitter.Fitter('CSV', LogLevel='DEBUG')
os.remove('test.csv')

# define model and initial guess
model = 'Randles'
parameters = {'Rct': {'value': 3. * Rct},
              'Rs': {'value': 0.5 * Rs},
              'C0': {'value': 0.1 * C0},
              'Aw': {'value': 1.2 * Aw}}

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

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}}
```

1.2.1 See Also

`examples/Randles/randles_data.py`.

1.3 Customizing the solver

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

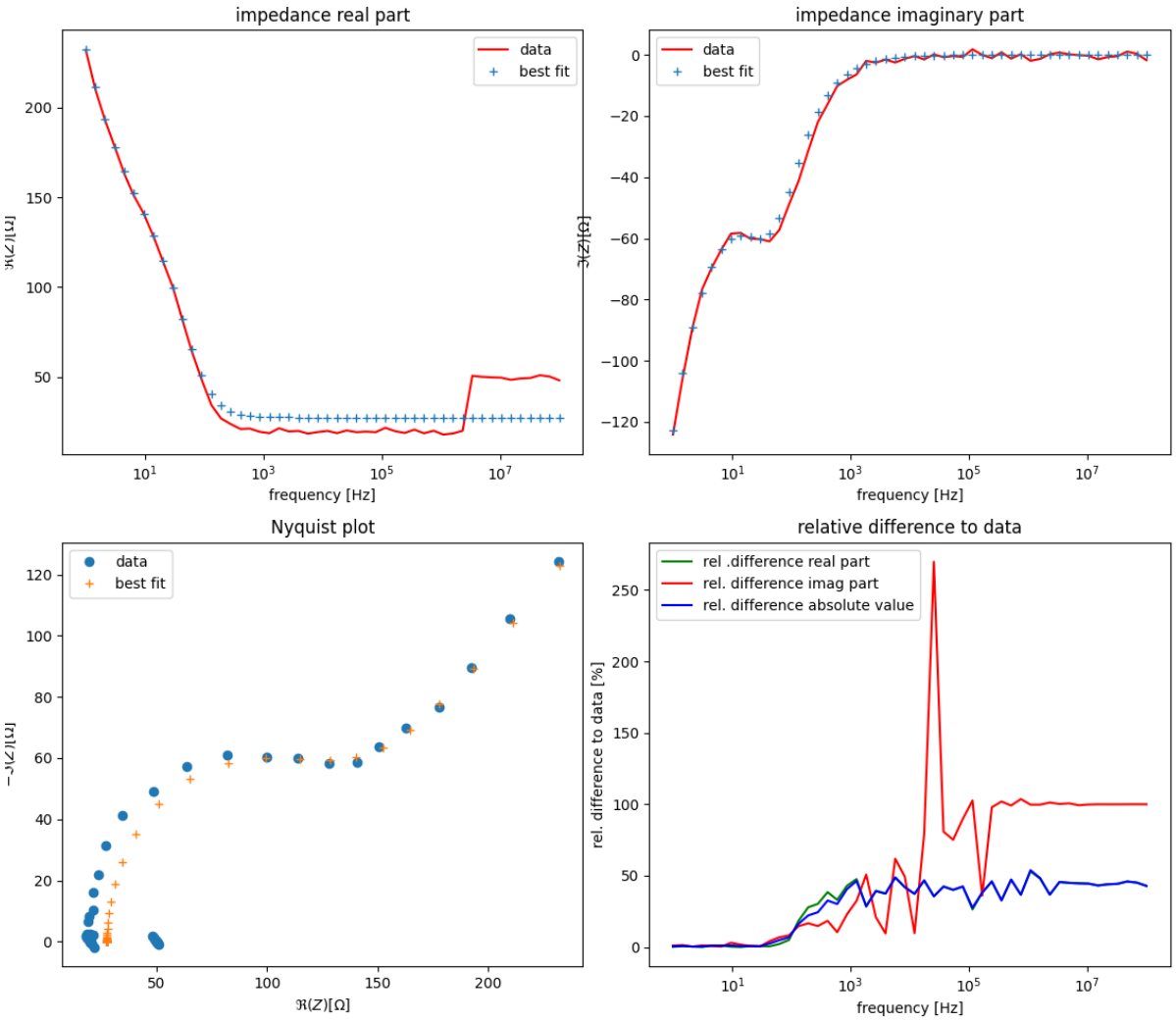
This time, noise and outliers are added to the data

```
# generate noise
rnd = numpy.random.RandomState(42)
noise = (1. + 1j) * rnd.randn(Z.size)

# let's have 10 outliers at the last frequencies
n_outliers = 10
Z[-n_outliers:] += 30

# add noise
Z += noise
```

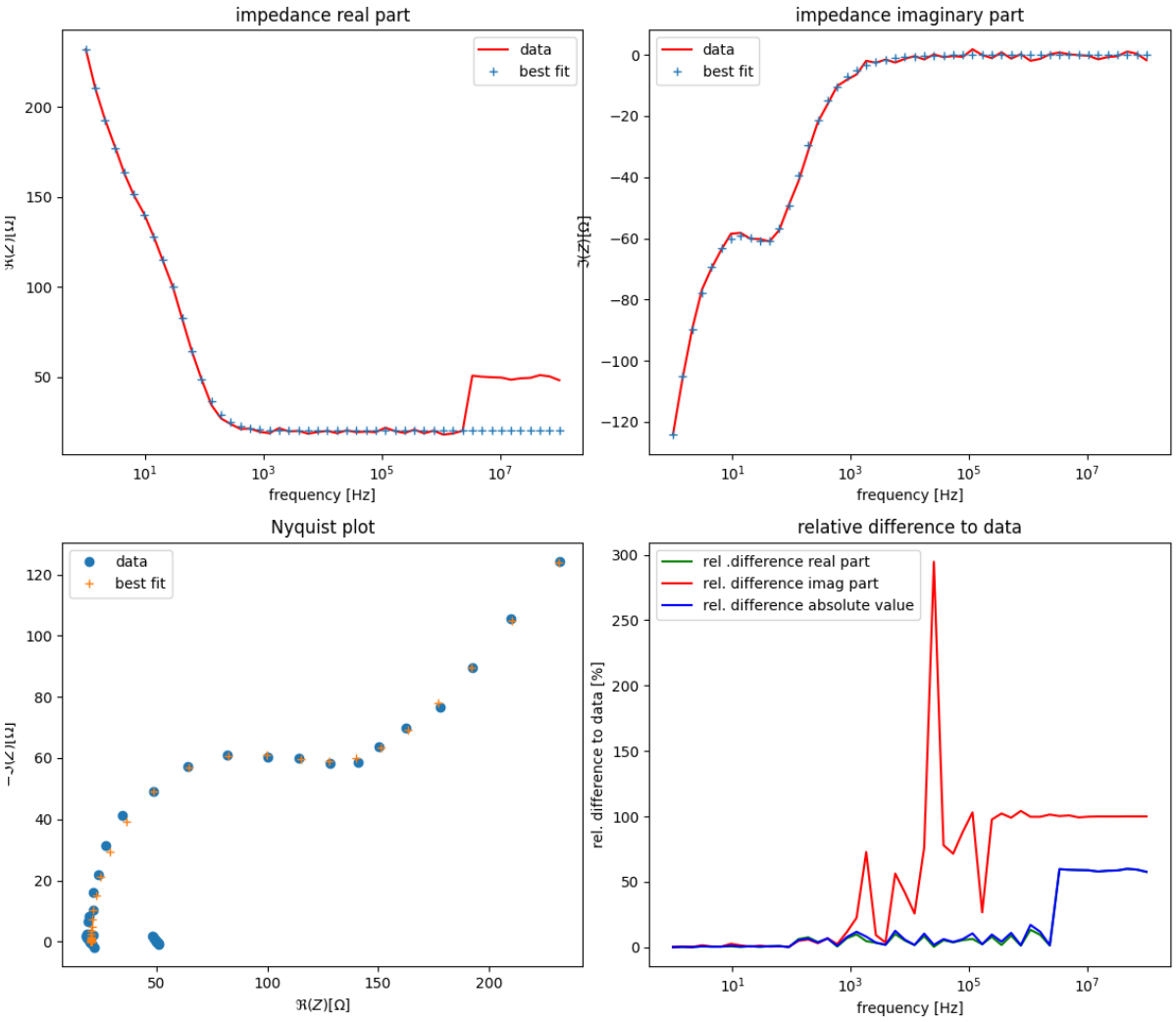
When running the fit with the standard least-squares solver, we obtain



Instead, we can customize the solver:

```
solver = "least_squares"
solver_kwargs = {'loss': 'soft_l1',
                 'f_scale': 2.}
fitter.run(model, parameters=parameters,
           solver=solver, solver_kwargs=solver_kwargs)
```

and obtain



1.3.1 See Also

`examples/Randles/randles_solver.py`.

1.4 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.

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

(continues on next page)

(continued from previous page)

```
# 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')
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))
```

1.4.1 See Also

examples/Statistics/statistics.py.

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()*.
- **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.
- **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.
- **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*)
- **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).

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*)

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.

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*)
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.

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

Return type `lmfit.model.Model`

model_iterations(*modelclass*)

Information about number of iterations if there is an iterative scheme for a modelclass.

Parameters **modelclass** (*str*) – Name of the modelclass. This means that this model is represented in the equivalent circuit.

Returns Number of iteration steps.

Return type int

Notes

Double-Shell model

The following iterative procedure is applied:

1. 1st Fit: The data is fitted against a model comprising the double-shell model. Parameters to be determined in this fitting round: *kmed* and *emed*.
2. 2nd Fit: The parameters *kmed* and *emed* are fixed and the data is fitted again. To be determined in this fit: *km* and *em*.
3. 3rd Fit: In addition, the parameters *km* and *em* are fixed and the data is fitted again. To be determined in this fit: *kcp*.
4. last Fit: In addition, he parameter *kcp* is fixed. To be determined in this fit: all remaining parameters.

Single-Shell model

The following iterative procedure is applied:

1. 1st Fit: The data is fitted against a model comprising the single-shell model. Parameters to be determined in this fitting round: *kmed* and *emed*.
2. 2nd Fit: The parameters *kmed* and *emed* are fixed and the data is fitted again.

Cole-Cole model

1. 1st Fit: The data is fitted against a model comprising the Cole-Cole model. Parameters to be determined in this fitting round: *kdc* and *eh*.
2. 2nd Fit: The parameters *kdc* and *eh* are fixed and the data is fitted again.

See also:

impedancefitter.double_shell.double_shell_model(), *impedancefitter.single_shell.single_shell_model()*, *impedancefitter.cole_cole.cole_cole_model()*

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()
Prepare initial configuration.

Todo: Implement.

---

process_data_from_file(*filename*, *model*, *parameters*, *modelclass=None*)
Fit data from input file to model.

Wrapper for LMFIT fitting routine. If `LogLevel` is *DEBUG*, the fit result is visualised.

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.
- **modelclass** (*str, optional*) – For an iterative scheme, the modelclass is passed to this function.

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

Return type `lmfit.model.ModelResult`

run(*modelname*, *solver=None*, *parameters=None*, *protocol=None*, *solver_kwargs={}*, *modelclass='none'*)
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).
- **protocol** (*string, optional*) – Choose 'Iterative' for repeated fits with changing parameter sets, customized approach. If not specified, there is always just one fit for each data set.
- **solver_kwargs** (*dict, optional*) – Customize the employed solver. Interface to the LMFIT routine.
- **modelclass** (*str, optional*) – Pass a modelclass for which the iterative scheme should be used. This is experimental support for iterative schemes, where parameters can be fixed during the fitting routine. In the future, a more intelligent approach could be found. See `impedancefitter.Fitter.model_iterations()`

sequential_run(*model1*, *model2*, *communicate*, *solver=None*, *solver_kwargs={}*, *parameters1=None*, *parameters2=None*, *modelclass1=None*, *modelclass2=None*, *protocol=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.
- **modelclass1** (*str, optional*) – Pass a modelclass for which the iterative scheme should be used. This is experimental support for iterative schemes, where parameters can be fixed during the fitting routine. In the future, a more intelligent approach could be found.
- **modelclass2** (*str, optional*) – Pass a modelclass for which the iterative scheme should be used. This is experimental support for iterative schemes, where parameters can be fixed during the fitting routine. In the future, a more intelligent approach could be found.
- **protocol** (*string, optional*) – Choose 'Iterative' for repeated fits with changing parameter sets, customized approach. If not specified, there is always just one fit for each data set.

visualize_data(*savefig=False*)

Visualize impedance data.

Parameters **savefig** (*bool, optional*) – Decide if plots should be saved as pdf. Default is False.

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()

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

UTILITITES

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.

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

impedancefitter.utils.**get_equivalent_circuit_model**(*modelname*)
Get LMFIT CompositeModel.

Parameters **modelname** (*str*) – String representation of the equivalent circuit.

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

$$Z = (j\omega\varepsilon^*)^{-1},$$

where ε^* is the complex permittivity (see for instance [Grant1958] for further explanation).

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

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

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.**set_parameters**(*model*, *parameterdict=None*, *emcee=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

Returns **params** – LMFIT Parameters object.

Return type lmfit.parameter.Parameters

CHAPTER

FIVE

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.

5.1 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()*

5.2 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

$$Z_{\mathrm{CPE}} = k^{-1}(j\omega)^{-\alpha}$$

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*, *C_stray*)
Stray capacitance in pF

Parameters

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

Todo: Better documentation needed.

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

Todo: Better documentation needed.

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

SIX

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.

6.1 Names for building the model

Name	Corresponding function
ColeCole	*impedancefitter.cole_cole.cole_cole_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()*
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()*

6.2 Cole-Cole circuits

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:

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

References

impedancefitter.cole_cole.**cole_cole_model**(*omega*, *c0*, *el*, *tau*, *a*, *kdc*, *eh*)
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
- **eh** (*double*) – value for ε_h
- **el** (*double*) – value for ε_l
- **tau** (*double*) – value for τ, in ns
- **kdc** (*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:

$$\varepsilon_\mathrm{s} = \varepsilon_\mathrm{h} + \frac{\varepsilon_\mathrm{l} - \varepsilon_\mathrm{h}}{1 + (j\omega\tau)^a}$$

$$Z_\mathrm{s} = \frac{1}{j\varepsilon_\mathrm{s}\omega c_0 + \frac{\sigma_\mathrm{dc} c_0}{\varepsilon_0}}$$

References

6.3 Single-Shell model

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
- **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()

6.4 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
- **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
- **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]:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

References

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()

6.5 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
- **C** (*double*) – capacitance
- **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

References

Notes

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

6.6 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_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.

$$Z_{\mathrm{CPE}} = k^{-1}(j\omega)^{-\alpha}$$

Parameters

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

Returns Impedance array

Return type numpy.ndarray, complex

6.7 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

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!

CHAPTER

SEVEN

OVERVIEW

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

Data analysis in IS relies often on non-linear least squares for parameter estimation of equivalent circuit models. ImpedanceFitter is a software that facilitates parameter estimation for various mechanistic models. The mechanistic model is based on an equivalent circuit that 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 to experimental impedance data. The underlying fitting software is LMFIT [Newville2019], which offers an interface to different optimization and curve-fitting methods. ImpedanceFitter allows one to build a custom equivalent circuit, fit an arbitrary amount of data sets and perform statistical analysis of the results.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[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

[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

[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

[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

[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

[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

[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

[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

[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

[Newville2019] Matt Newville, Renee Otten, Andrew Nelson, Antonino Ingargiola, Till Stensitzki, Dan Allan, Austin Fox, Faustin Carter, Michał, Dima Pustakhod, Yoav Ram, Glenn, Christoph Deil, Stuermer, Alexandre Beelen, Oliver Frost, Nicholas Zobrist, Gustavo Pasquevich, Allan L. R. Hansen, Tim Spillane, Shane Caldwell, Anthony Polloreno, andrewhannum, Julius Zimmermann, Jose Borreguero, Jonathan Fraine, deep-42-thought, Benjamin F. Maier, Ben Gamari, Anthony Almarza. (2019, December 20). lmfit/lmfit-py 1.0.0 (Version 1.0.0). Zenodo. http://doi.org/10.5281/zenodo.3588521

PYTHON MODULE INDEX

i

INDEX

P

R

S

V

Z