impedance.py Documentation

Release 1.4.1

Name

Apr 25, 2022

Contents

`impedance.py` is a Python package for making electrochemical impedance spectroscopy (EIS) analysis reproducible and easy-to-use.

Aiming to create a consistent, scikit-learn-like API for impedance analysis, `impedance.py` contains modules for data preprocessing, validation, model fitting, and visualization.

If you have a feature request or find a bug, please file an issue or, better yet, make the code improvements and submit a pull request! The goal is to build an open-source tool that the entire impedance community can improve and use!

CHAPTER 1

Installation

The easiest way to install `impedance.py` is from PyPI using pip:

```
pip install impedance
```

See *Getting started with impedance.py* for instructions on getting started from scratch.

1.1 Dependencies

impedance.py requires:

- Python (>=3.7)
- SciPy (>=1.0)
- NumPy (>=1.14)
- Matplotlib (>=3.0)
- Altair (>=3.0)

Several example notebooks are provided in the examples/ directory. Opening these will require Jupyter notebook or Jupyter lab.

CHAPTER 2

Examples and Documentation

Getting started with impedance.py contains a detailed walk through of how to get started from scratch. If you're already familiar with Jupyter/Python, several examples can be found in the `examples/` directory (*Fitting impedance spectra* is a great place to start). The documentation can be found at impedancepy.readthedocs.io.

2.1 Getting started with `impedance.py`

`impedance.py` is a Python package for analyzing electrochemical impedance spectroscopy (EIS) data. The following steps will show you how to get started analyzing your own data using `impedance.py` in a Jupyter notebook.

Hint: If you get stuck or believe you have found a bug, please feel free to open an issue on GitHub.

2.1.1 Step 1: Installation

If you already are familiar with managing Python packages, feel free to skip straight to *Installing packages*. Otherwise, what follows is a quick introduction to the Python package ecosystem:

Installing Miniconda

One of the easiest ways to get started with Python is using Miniconda. Installation instructions for your OS can be found at https://conda.io/miniconda.html.

After you have installed conda, you can run the following commands in your Terminal/command prompt/Git BASH to update and test your installation:

1. Update conda's listing of packages for your system: `conda update conda`
2. Test your installation: `conda list`

   For a successful installation, a list of installed packages appears.

3. Test that Python 3 is your default Python: `python -V`

   You should see something like Python 3.x.x :: Anaconda, Inc.

You can interact with Python at this point, by simply typing `python`.

Setting up a conda environment

(Optional) It is recommended that you use virtual environments to keep track of the packages you've installed for a particular project. Much more info on how conda makes this straightforward is given here.

We will start by creating an environment called `impedance-analysis` which contains all the Python base distribution:

```
conda create -n impedance-analysis python=3
```

After conda creates this environment, we need to activate it before we can install anything into it by using:

```
conda activate impedance-analysis
```

We've now activated our conda environment and are ready to install `impedance.py`!

Installing packages

The easiest way to install `impedance.py` and it's dependencies (`scipy`, `numpy`, and `matplotlib`) is from PyPI using pip:

```
pip install impedance
```

For this example we will also need Jupyter Lab which we can install with:

```
conda install jupyter jupyterlab
```

We've now got everything in place to start analyzing our EIS data!

Note: The next time you want to use this same environment, all you have to do is open your terminal and type `conda activate impedance-analysis`.

Open Jupyter Lab

(Optional) Create a directory in your documents folder for this example:

```
mkdir ~/Documents/impedance-example

cd ~/Documents/impedance-example
```

Next, we will launch an instance of Jupyter Lab:

```
jupyter lab
```

which should open a new tab in your browser. A tutorial on Jupyter Lab from the Electrochemical Society HackWeek can be found here.

Tip: The code below can be found in the getting-started.ipynb notebook

2.1.2 Step 2: Import your data

This example will assume the following dataset is located in your current working directory (feel free to replace it with your data): `exampleData.csv`

For this dataset which simply contains impedance data in three columns (frequency, Z_real, Z_imag), importing the data looks something like:

```
from impedance import preprocessing

# Load data from the example EIS data
frequencies, Z = preprocessing.readCSV('./exampleData.csv')

# keep only the impedance data in the first quandrant
frequencies, Z = preprocessing.ignoreBelowX(frequencies, Z)
```

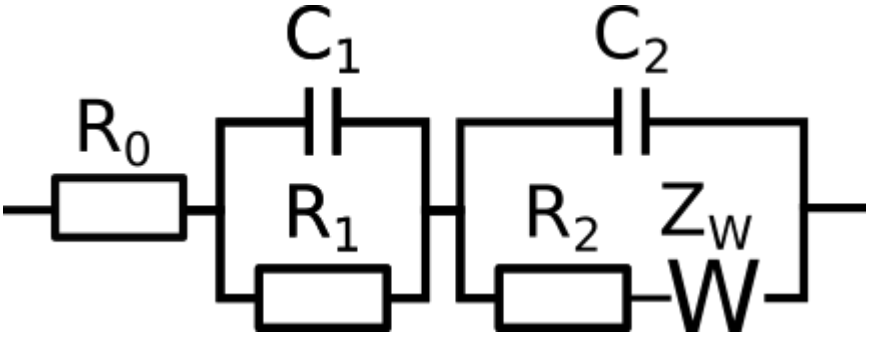
Tip: Functions for reading in files from a variety of vendors (ZPlot, Gamry, Parstat, Autolab, . . .) can be found in the preprocessing module!

2.1.3 Step 3: Define your impedance model

Next we want to define our impedance model. In order to enable a wide variety of researchers to use the tool, `impedance.py` allows you to define a custom circuit with any combination of circuit elements.

The circuit is defined as a string (i.e. using `''` in Python), where elements in series are separated by a dash (`-`), and elements in parallel are wrapped in a `p( , )`. Each element is defined by the function (in circuit-elements.py) followed by a single digit identifier.

For example, the circuit below:



would be defined as `R0-p(R1,C1)-p(R2-Wo1,C2)`.

Each circuit, we want to fit also needs to have an initial guess for each of the parameters. These inital guesses are passed in as a list in order the parameters are defined in the circuit string. For example, a good guess for this battery data is `initial_guess = [.01, .01, 100, .01, .05, 100, 1]`.

We create the circuit by importing the CustomCircuit object and passing it our circuit string and initial guesses.

```
from impedance.models.circuits import CustomCircuit

circuit = 'R0-p(R1,C1)-p(R2-Wo1,C2)'
initial_guess = [.01, .01, 100, .01, .05, 100, 1]

circuit = CustomCircuit(circuit, initial_guess=initial_guess)
```

2.1.4 Step 4: Fit the impedance model to data

Once we've defined our circuit, fitting it to impedance data is as easy as calling the *.fit()* method and passing it our experimental data!

```
circuit.fit(frequencies, Z)
```

We can access the fit parameters with `circuit.parameters_` or by printing the circuit object itself, `print(circuit)`.

2.1.5 Step 5: Analyze/Visualize the results

For this dataset, the resulting fit parameters are

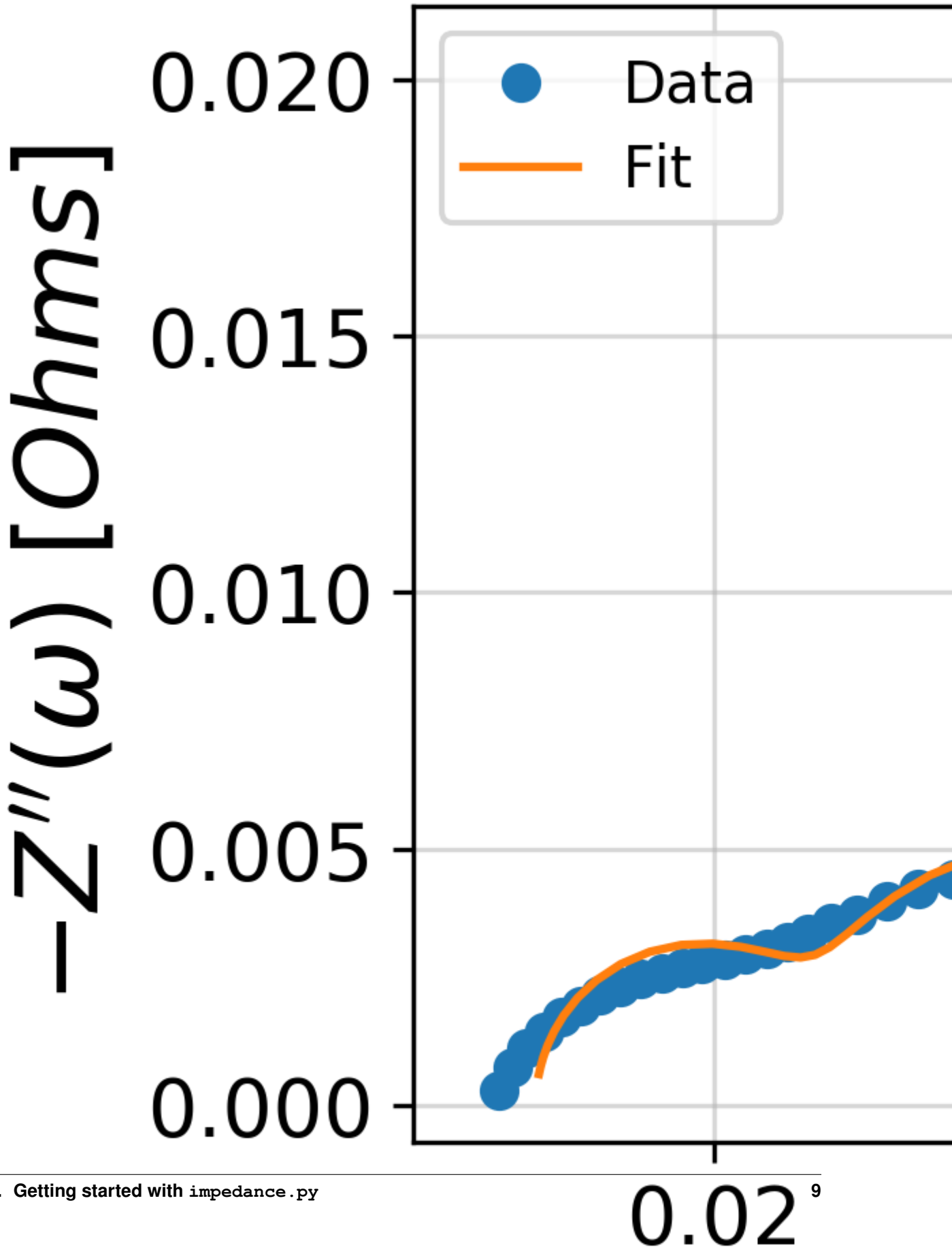
Parameter	Value
R_0	1.65e-02
R_1	8.68e-03
C_1	3.32e+00
R_2	5.39e-03
$Wo_{1,0}$	6.31e-02
$Wo_{1,1}$	2.33e+02
C_2	2.20e-01

We can get the resulting fit impedance by passing a list of frequencies to the `.predict()` method.

```
Z_fit = circuit.predict(frequencies)
```

To easily visualize the fit, the `plot_nyquist()` function can be handy.

```
import matplotlib.pyplot as plt
from impedance.visualization import plot_nyquist

fig, ax = plt.subplots()
plot_nyquist(ax, Z, fmt='o')
plot_nyquist(ax, Z_fit, fmt='-')

plt.legend(['Data', 'Fit'])
plt.show()
```


Data
Fit
0.020
0.015
0.010
0.005
0.000
$-Z''(\omega)$ [$Ohms$]
0.02

Important: Congratulations! You're now up and running with impedance.py

2.2 Examples

2.2.1 Fitting impedance spectra

1. Import and initialize equivalent circuit(s)

To begin we will import the Randles' circuit and a custom circuit from the impedance package. A full list of currently available circuits are available in the documentation.

[1]:
```
from impedance.models.circuits import Randles, CustomCircuit
```

The classes we just imported represent different equivalent circuit models. To actually use them we want to initialize a specific instance and provide an initial guess for the parameters and any other options.

E.g. for the randles circuit, one of the options is for a constant phase element (CPE) instead of an ideal capacitor.

[2]:
```
randles = Randles(initial_guess=[.01, .005, .001, 200, .1])
randlesCPE = Randles(initial_guess=[.01, .005, .001, 200, .1, .9], CPE=True)
```

Defining the custom circuit works a little differently. Here we pass a string comprised of the circuit elements grouped either in series (separated with a `-`) or in parallel (using the form `p(X,Y)`). Each element can be appended with an integer (e.g. `R0`) or an underscore and an integer (e.g. `CPE_1`) to make keeping track of multiple elements of the same type easier.

[3]:
```
customCircuit = CustomCircuit(initial_guess=[.01, .005, .1, .005, .1, .001, 200],
                              circuit='R_0-p(R_1,C_1)-p(R_2,C_2)-Wo_1')
```

As of version 0.4, you can now specify values you want to hold constant. For example,

[4]:
```
customConstantCircuit = CustomCircuit(initial_guess=[None, .005, .1, .005, .1, .001,␣
↪None],
                                      constants={'R_0': 0.02, 'Wo_1_1': 200},
                                      circuit='R_0-p(R_1,C_1)-p(R_2,C_2)-Wo_1')
```

Each of the circuit objects we create can be printed in order to see the properties that have been defined for that circuit.

[5]:
```
print(customConstantCircuit)
```

```
Circuit string: R_0-p(R_1,C_1)-p(R_2,C_2)-Wo_1
Fit: False

Constants:
    R_0 = 2.00e-02 [Ohm]
  Wo_1_1 = 2.00e+02 [sec]

Initial guesses:
    R_1 = 5.00e-03 [Ohm]
    C_1 = 1.00e-01 [F]
    R_2 = 5.00e-03 [Ohm]
    C_2 = 1.00e-01 [F]
```

(continues on next page)

(continued from previous page)

```
  Wo_1_0 = 1.00e-03 [Ohm]
```

2. Formulate data

Several convenience functions for importing data exist in the impedance.preprocessing module, including one for reading simple `.csv` files where frequencies are stored in the first column, real parts of the impedance are in the second column, and imaginary parts of the impedance are in the third column.

[6]:
```
from impedance import preprocessing

frequencies, Z = preprocessing.readCSV('../../../data/exampleData.csv')

# keep only the impedance data in the first quandrant
frequencies, Z = preprocessing.ignoreBelowX(frequencies, Z)
```

3. Fit the equivalent circuits to a spectrum

Each of the circuit classes has a `.fit()` method which finds the best fitting parameters.

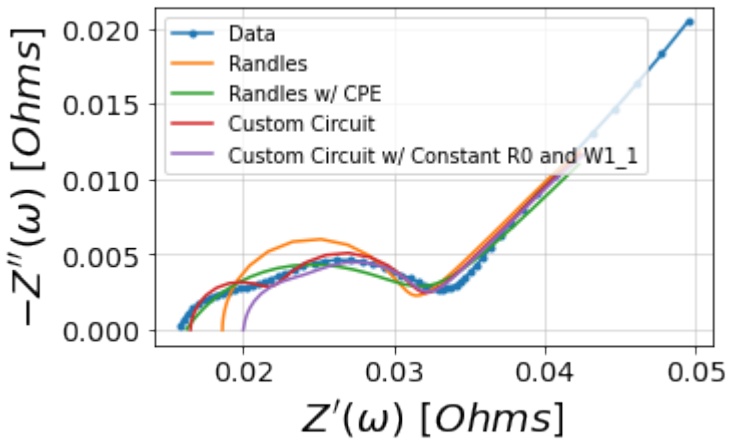
After fitting a circuit, the fit parameters rather that the inital guesses are shown when printing.

[7]:
```
randles.fit(frequencies, Z)
randlesCPE.fit(frequencies, Z)
customCircuit.fit(frequencies, Z)
customConstantCircuit.fit(frequencies, Z)

print(customConstantCircuit)
```

```
Circuit string: R_0-p(R_1,C_1)-p(R_2,C_2)-Wo_1
Fit: True

Constants:
    R_0 = 2.00e-02 [Ohm]
  Wo_1_1 = 2.00e+02 [sec]

Initial guesses:
    R_1 = 5.00e-03 [Ohm]
    C_1 = 1.00e-01 [F]
    R_2 = 5.00e-03 [Ohm]
    C_2 = 1.00e-01 [F]
  Wo_1_0 = 1.00e-03 [Ohm]

Fit parameters:
    R_1 = 6.79e-03  (+/- 1.08e-03) [Ohm]
    C_1 = 5.62e+00  (+/- 1.96e+00) [F]
    R_2 = 3.91e-03  (+/- 1.09e-03) [Ohm]
    C_2 = 1.36e+00  (+/- 2.61e-01) [F]
  Wo_1_0 = 5.88e-02  (+/- 1.25e-03) [Ohm]
```

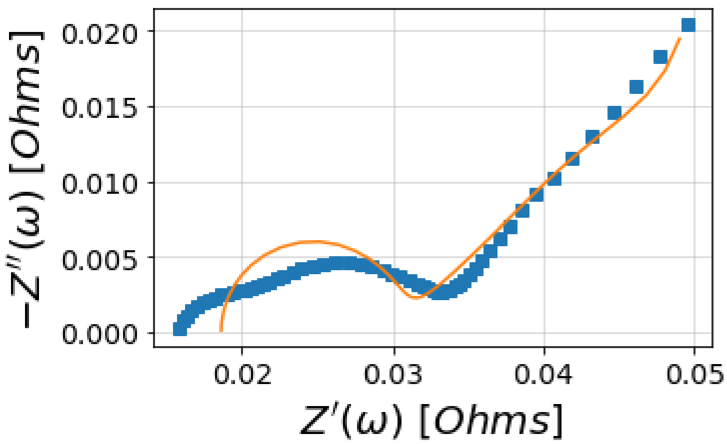
4a. Predict circuit model and visualize with matplotlib

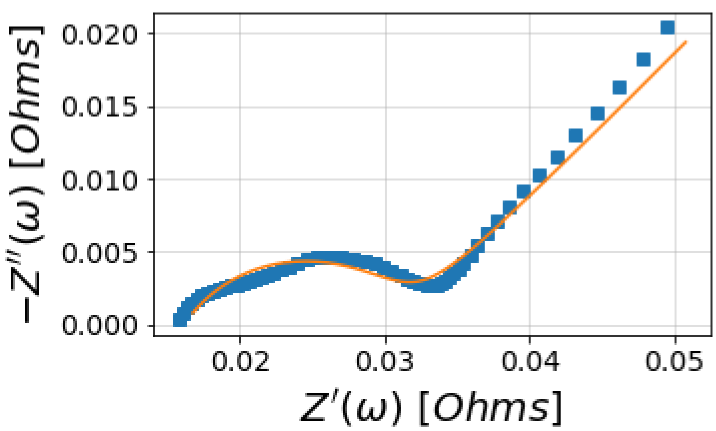
[8]:
```
import matplotlib.pyplot as plt
import numpy as np
from impedance.visualization import plot_nyquist

f_pred = np.logspace(5,-2)

randles_fit = randles.predict(f_pred)
randlesCPE_fit = randlesCPE.predict(f_pred)
customCircuit_fit = customCircuit.predict(f_pred)
customConstantCircuit_fit = customConstantCircuit.predict(f_pred)

fig, ax = plt.subplots(figsize=(5,5))

plot_nyquist(ax, Z)
plot_nyquist(ax, randles_fit, fmt='-')
plot_nyquist(ax, randlesCPE_fit, fmt='-')
plot_nyquist(ax, customCircuit_fit, fmt='-')
plot_nyquist(ax, customConstantCircuit_fit, fmt='-')

ax.legend(['Data', 'Randles', 'Randles w/ CPE', 'Custom Circuit', 'Custom Circuit w/␣
↪Constant R0 and W1_1'])
plt.show()
```

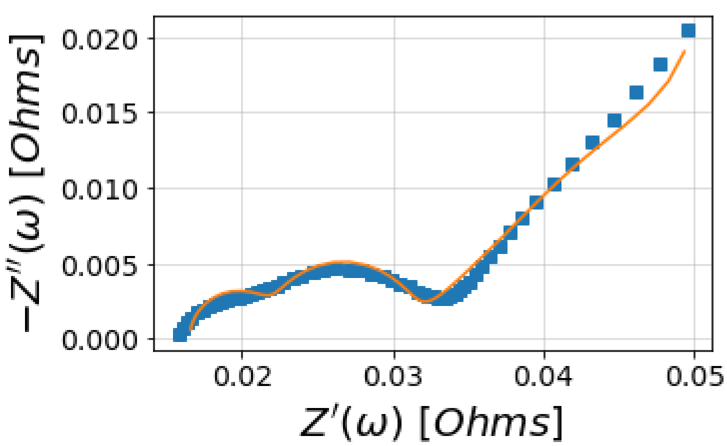


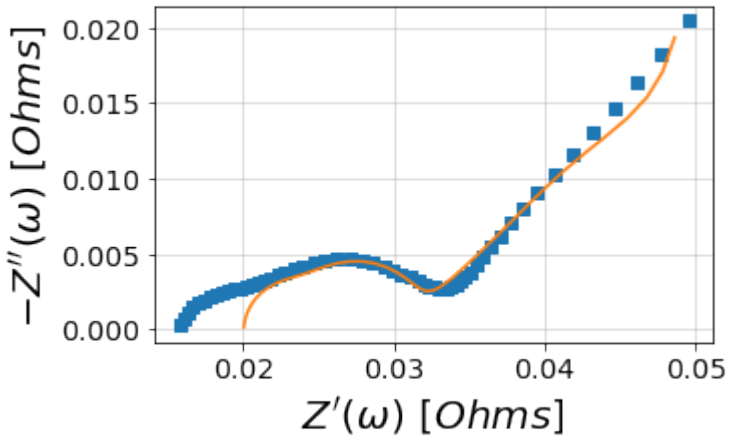
4b. Or use the convenient plotting method included in the package

[9]:
```
randles.plot(f_data=frequencies, Z_data=Z, kind='nyquist')
randlesCPE.plot(f_data=frequencies, Z_data=Z, kind='nyquist')
customCircuit.plot(f_data=frequencies, Z_data=Z, kind='nyquist')
customConstantCircuit.plot(f_data=frequencies, Z_data=Z, kind='nyquist')

plt.show()
```


0.020
0.015
0.010
0.005
0.000
0.02
0.03
0.04
0.05
$-Z''(\omega)$ $[Ohms]$
$Z'(\omega)$ $[Ohms]$


0.020
0.015
0.010
0.005
0.000
0.02
0.03
0.04
0.05
$-Z''(\omega)$ $[Ohms]$
$Z'(\omega)$ $[Ohms]$


0.020
0.015
0.010
0.005
0.000
0.02
0.03
0.04
0.05
$-Z''(\omega)$ $[Ohms]$
$Z'(\omega)$ $[Ohms]$



[ ]:

2.2.2 Visualizing impedance spectra

Plotting a basically formated impedance plot is as easy as 1, 2, 3…

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

from impedance.models.circuits import CustomCircuit
from impedance import preprocessing
```

1. Read in data

[2]:
```
frequencies, Z = preprocessing.readCSV('../../../data/exampleData.csv')

# keep only the impedance data in the first quandrant
frequencies, Z = preprocessing.ignoreBelowX(frequencies, Z)
```

2. Fit a custom circuit

(If you want to just plot experimental data without fitting a model you should check out the `visualization.plot_*()` functions)

[3]:
```
circuit = CustomCircuit(initial_guess=[.01, .005, .1, .005, .1, .001, 200], circuit=
↪'R_0-p(R_1,C_1)-p(R_1,C_1)-Wo_1')

circuit.fit(frequencies, Z)

print(circuit)
```

```
Circuit string: R_0-p(R_1,C_1)-p(R_1,C_1)-Wo_1
Fit: True
```

(continues on next page)

(continued from previous page)

```
Initial guesses:
    R_0 = 1.00e-02 [Ohm]
    R_1 = 5.00e-03 [Ohm]
    C_1 = 1.00e-01 [F]
    R_1 = 5.00e-03 [Ohm]
    C_1 = 1.00e-01 [F]
  Wo_1_0 = 1.00e-03 [Ohm]
  Wo_1_1 = 2.00e+02 [sec]

Fit parameters:
    R_0 = 1.65e-02  (+/- 1.54e-04) [Ohm]
    R_1 = 5.31e-03  (+/- 2.06e-04) [Ohm]
    C_1 = 2.32e-01  (+/- 1.90e-02) [F]
    R_1 = 8.77e-03  (+/- 1.89e-04) [Ohm]
    C_1 = 3.28e+00  (+/- 1.85e-01) [F]
  Wo_1_0 = 6.37e-02  (+/- 2.03e-03) [Ohm]
  Wo_1_1 = 2.37e+02  (+/- 1.72e+01) [sec]
```

3. Plot the data and fit

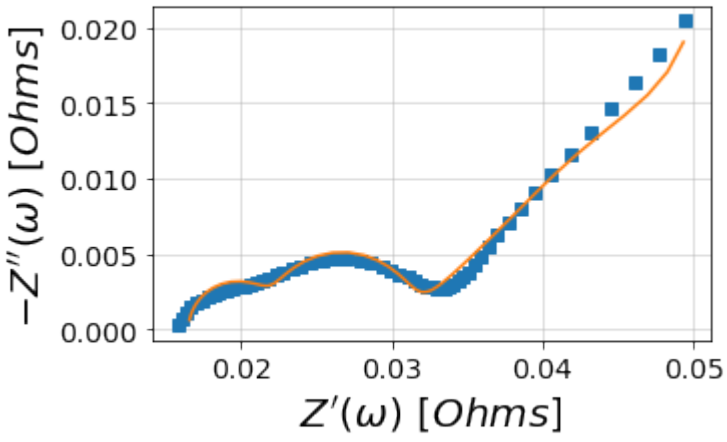
a. Interactive altair plot

[4]:
```
circuit.plot(f_data=frequencies, Z_data=Z)
```

[4]:
```
alt.HConcatChart(...)
```

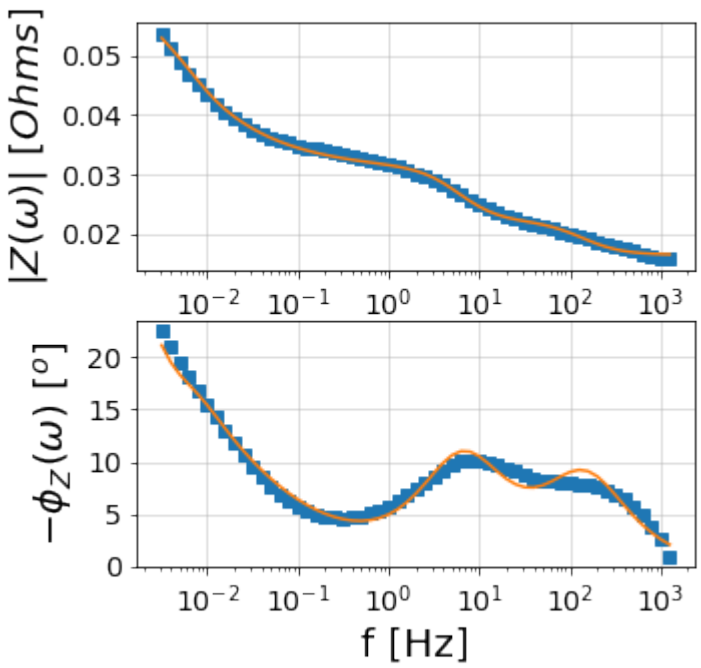
b. Nyquist plot via matplotlib

[5]:
```
circuit.plot(f_data=frequencies, Z_data=Z, kind='nyquist')
plt.show()
```



c. Bode plot via matplotlib

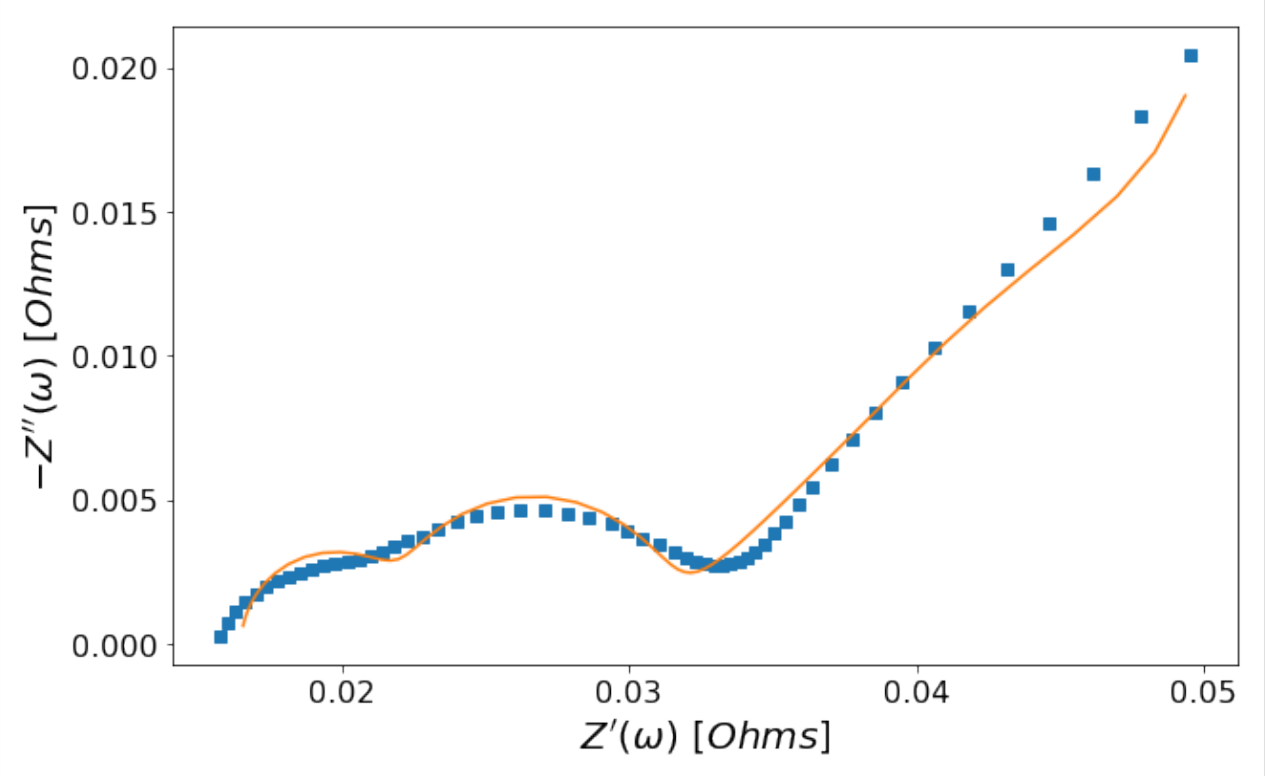
[6]:
```
circuit.plot(f_data=frequencies, Z_data=Z, kind='bode')
plt.show()
```



Bonus: Easy access to all the customization of matplotlib

Here we plot the data, changing the size of the figure, axes label fontsize, and turning off the grid by accessing the plt.Axes() object, ax

[7]:
```
fig, ax = plt.subplots(figsize=(10,10))
ax = circuit.plot(ax, frequencies, Z, kind='nyquist')

ax.tick_params(axis='both', which='major', labelsize=16)
ax.grid(False)

plt.show()
```



2.2.3 Model Saving/Loading Example

This set of examples shows how to load and import template models in order to make setting up and reproducing circuit fits easier.

[1]:
```
# Load libraries

import impedance.preprocessing as preprocessing
import impedance.models.circuits as circuits
from impedance.visualization import plot_nyquist
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
# Load data from the example EIS result
frequencies, Z = preprocessing.readCSV('../../../data/exampleData.csv')

# keep only the impedance data in the first quandrant
frequencies, Z = preprocessing.ignoreBelowX(frequencies, Z)
```

Example 1. Importing and Exporting Models

Call the `circuit.save()` function to export the model to a human readable JSON file. The following code generates a test circuit and export it as a template. Here we are using an unfitted model as a template.

[3]:
```
test_circuit = circuits.CustomCircuit(initial_guess=[.01, .005, .1, .005, .1, .001,␣
↪200],
                                      circuit='R0-p(R1,C1)-p(R2,C2)-Wo1')
```

(continues on next page)

(continued from previous page)

```
print(test_circuit)

test_circuit.save('template_model.json')
```

```
Circuit string: R0-p(R1,C1)-p(R2,C2)-Wo1
Fit: False

Initial guesses:
     R0 = 1.00e-02 [Ohm]
     R1 = 5.00e-03 [Ohm]
     C1 = 1.00e-01 [F]
     R2 = 5.00e-03 [Ohm]
     C2 = 1.00e-01 [F]
  Wo1_0 = 1.00e-03 [Ohm]
  Wo1_1 = 2.00e+02 [sec]
```

Call the `model_io.model_import` function to import the model back as a template.

[4]:
```
loaded_template = circuits.CustomCircuit()
loaded_template.load('template_model.json')

print("Loaded Template")
print(loaded_template)
```

```
R0-p(R1,C1)-p(R2,C2)-Wo1
Loaded Template

Circuit string: R0-p(R1,C1)-p(R2,C2)-Wo1
Fit: False

Initial guesses:
     R0 = 1.00e-02 [Ohm]
     R1 = 5.00e-03 [Ohm]
     C1 = 1.00e-01 [F]
     R2 = 5.00e-03 [Ohm]
     C2 = 1.00e-01 [F]
  Wo1_0 = 1.00e-03 [Ohm]
  Wo1_1 = 2.00e+02 [sec]
```

Example 2. Using imported template model to fit data

After the model has been imported as a template, it can be used as a starting point to fit data. This saves on needing to configure the initial parameters each time a fit is performed and to persist starting conditions across several fitting sessions.

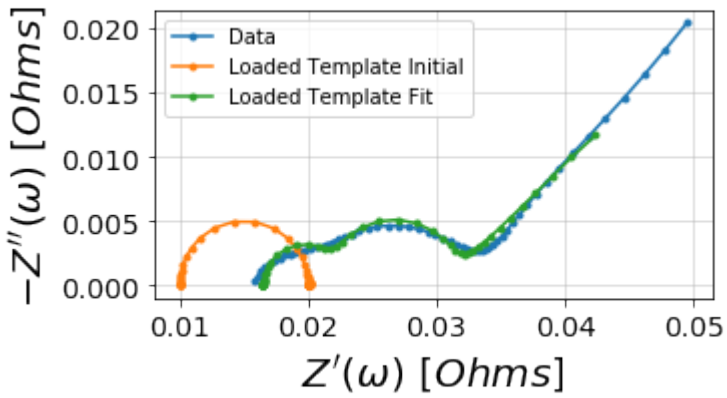
[5]:
```
fig, ax = plt.subplots(figsize=(5,5))
f_pred = np.logspace(5,-2)
loaded_template.fit(frequencies, Z)

imported_circuit_init = loaded_template.predict(f_pred, use_initial = True)
imported_circuit_fit = loaded_template.predict(f_pred)
```

(continues on next page)

(continued from previous page)

```
plot_nyquist(ax, Z)
plot_nyquist(ax, imported_circuit_init)
plot_nyquist(ax, imported_circuit_fit)

ax.legend(['Data', 'Loaded Template Initial', 'Loaded Template Fit'])

plt.show()
print(loaded_template)
```

```
Simulating circuit based on initial parameters
```




```
Circuit string: R0-p(R1,C1)-p(R2,C2)-Wo1
Fit: True

Initial guesses:
     R0 = 1.00e-02 [Ohm]
     R1 = 5.00e-03 [Ohm]
     C1 = 1.00e-01 [F]
     R2 = 5.00e-03 [Ohm]
     C2 = 1.00e-01 [F]
  Wo1_0 = 1.00e-03 [Ohm]
  Wo1_1 = 2.00e+02 [sec]

Fit parameters:
     R0 = 1.65e-02  (+/- 1.54e-04) [Ohm]
     R1 = 8.77e-03  (+/- 1.89e-04) [Ohm]
     C1 = 3.28e+00  (+/- 1.85e-01) [F]
     R2 = 5.31e-03  (+/- 2.06e-04) [Ohm]
     C2 = 2.32e-01  (+/- 1.90e-02) [F]
  Wo1_0 = 6.37e-02  (+/- 2.03e-03) [Ohm]
  Wo1_1 = 2.37e+02  (+/- 1.72e+01) [sec]
```

Example 3. Using fitted data as a starting point for new fits

Consider the case where a successful fit has been performed and a new set of EIS data is obtained which is similar to the first spectrum. It is useful to use the successfully fitted parameters as a starting point for subsequent fits.

[6]:
```
# Export the fitted model as a template
```

(continues on next page)

(continued from previous page)

```
loaded_template.save('fitted_template.json')
```

Using the exported model's fitted parameters, generate a new circuit using the fitted parameters as initial guesses by suppling the `fitted_as_initial` parameter as `True`.

[7]:
```
fitted_template = circuits.CustomCircuit()
fitted_template.load('fitted_template.json', fitted_as_initial=True)
print(fitted_template)
```

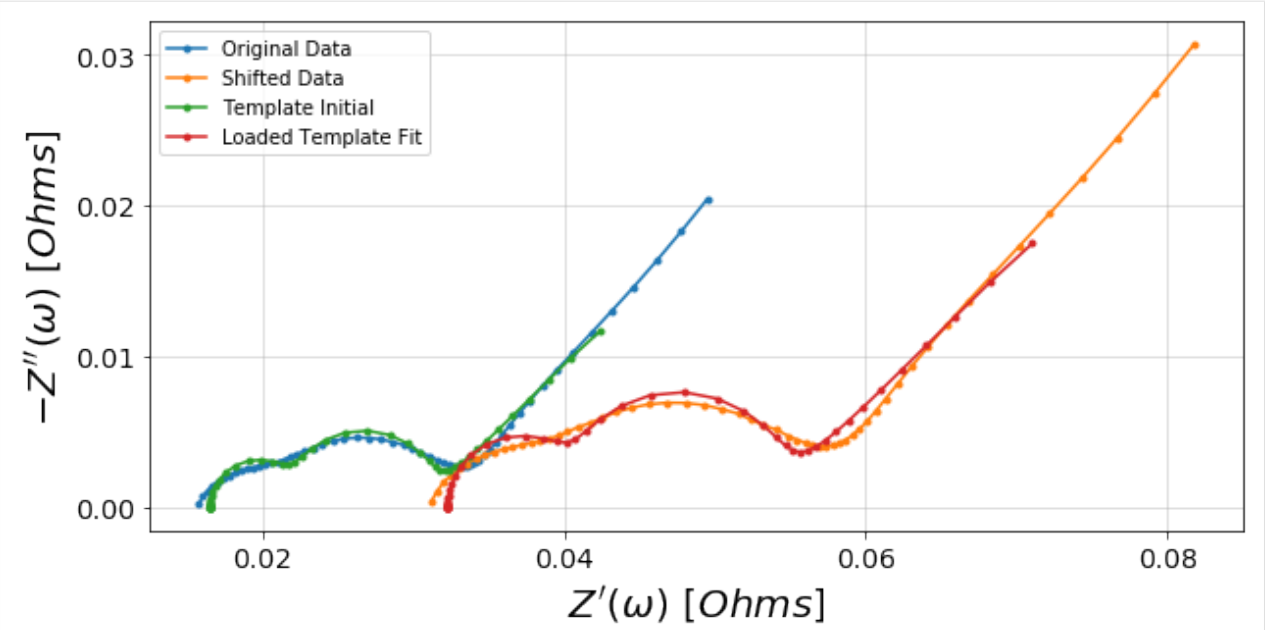
```
R0-p(R1,C1)-p(R2,C2)-Wo1

Circuit string: R0-p(R1,C1)-p(R2,C2)-Wo1
Fit: False

Initial guesses:
     R0 = 1.65e-02 [Ohm]
     R1 = 8.77e-03 [Ohm]
     C1 = 3.28e+00 [F]
     R2 = 5.31e-03 [Ohm]
     C2 = 2.32e-01 [F]
  Wo1_0 = 6.37e-02 [Ohm]
  Wo1_1 = 2.37e+02 [sec]
```

Z2 is a similar impedance spectra that we can fit using the previous fitted parameters as starting points. It has been shifted by 5 mOhm in the real axis and the data has been scaled by 1.5x.

[8]:
```
Z2 = (0.005 + Z.real)*1.5 + 1.5j*Z.imag
```

[9]:
```
fig, ax = plt.subplots(figsize=(10,10))
f_pred = np.logspace(5,-2)
fitted_template.fit(frequencies, Z2)

imported_circuit_init = fitted_template.predict(f_pred, use_initial = True)
imported_circuit_fit = fitted_template.predict(f_pred)

plot_nyquist(ax, Z)
plot_nyquist(ax, Z2)
plot_nyquist(ax, imported_circuit_init)
plot_nyquist(ax, imported_circuit_fit)

ax.legend(['Original Data', 'Shifted Data', 'Template Initial', 'Loaded Template Fit
↪'])

plt.show()

print(fitted_template)
```

```
Simulating circuit based on initial parameters
```




```
Circuit string: R0-p(R1,C1)-p(R2,C2)-Wo1
Fit: True

Initial guesses:
     R0 = 1.65e-02 [Ohm]
     R1 = 8.77e-03 [Ohm]
     C1 = 3.28e+00 [F]
     R2 = 5.31e-03 [Ohm]
     C2 = 2.32e-01 [F]
  Wo1_0 = 6.37e-02 [Ohm]
  Wo1_1 = 2.37e+02 [sec]

Fit parameters:
     R0 = 3.22e-02  (+/- 2.31e-04) [Ohm]
     R1 = 1.31e-02  (+/- 2.84e-04) [Ohm]
     C1 = 2.19e+00  (+/- 1.24e-01) [F]
     R2 = 7.96e-03  (+/- 3.10e-04) [Ohm]
     C2 = 1.55e-01  (+/- 1.26e-02) [F]
  Wo1_0 = 9.56e-02  (+/- 3.05e-03) [Ohm]
  Wo1_1 = 2.38e+02  (+/- 1.73e+01) [sec]
```

2.2.4 Validation of EIS data

The Kramers-Kronig Relations

Electrochemical impedance spectroscopy (EIS) is built on linear systems theory which requires that the system satisfy conditions of causality, linearity, and stability. The Kramers-Kronig relations consist of a set of transformations that can be used to predict one component of the impedance from the other over the frequency limits from zero to infinity. For example, one might calculate the imaginary component of the impedance from the measured real component,

$$Z''(\omega) = -\frac{2\omega}{\pi}\int_0^\infty \frac{Z'(x) - Z'(\omega)}{x^2 - \omega^2}dx$$

where $Z'(\omega)$ and $Z''(\omega)$ are the real and imaginary components of the impedance as a function of frequency, ω. Similarly, the real part of the impedance spectrum can be calculated from the imaginary part by

$$Z'(\omega) = Z'(\infty) + \frac{2}{\pi}\int_0^\infty \frac{xZ''(x) - \omega Z''(\omega)}{x^2 - \omega^2} dx$$

The residual error between the predicted and measured impedance can then be used to determine consistency with the Kramers-Kronig relations.

Practically, however, the 0 to ∞ frequency range required for integration can be difficult to measure experimentally, so several other methods have been developed to ensure Kramers-Kronig relations are met:

Measurement models

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

from impedance.models.circuits import CustomCircuit
from impedance import preprocessing
```

[2]:
```
# Load data from the example EIS result
f, Z = preprocessing.readCSV('../../../data/exampleData.csv')

# keep only the impedance data in the first quandrant
f, Z = preprocessing.ignoreBelowX(f, Z)

mask = f < 1000
f = f[mask]
Z = Z[mask]
```

[3]:
```
N = 10

circuit = 'R_0'
initial_guess = [.015]
for i in range(N):
    circuit += f'-p(R_{i % 9 + 1},C_{i % 9 + 1})'
    initial_guess.append(.03/N)
    initial_guess.append(10**(3 - 6*i/N))

meas_model = CustomCircuit(initial_guess=initial_guess, circuit=circuit)
```

[4]:
```
meas_model.fit(f, Z)

print(meas_model)
```

```
Circuit string: R_0-p(R_1,C_1)-p(R_2,C_2)-p(R_3,C_3)-p(R_4,C_4)-p(R_5,C_5)-p(R_6,C_6)-
↪p(R_7,C_7)-p(R_8,C_8)-p(R_9,C_9)-p(R_1,C_1)
Fit: True

Initial guesses:
    R_0 = 1.50e-02 [Ohm]
    R_1 = 3.00e-03 [Ohm]
```

(continues on next page)

(continued from previous page)

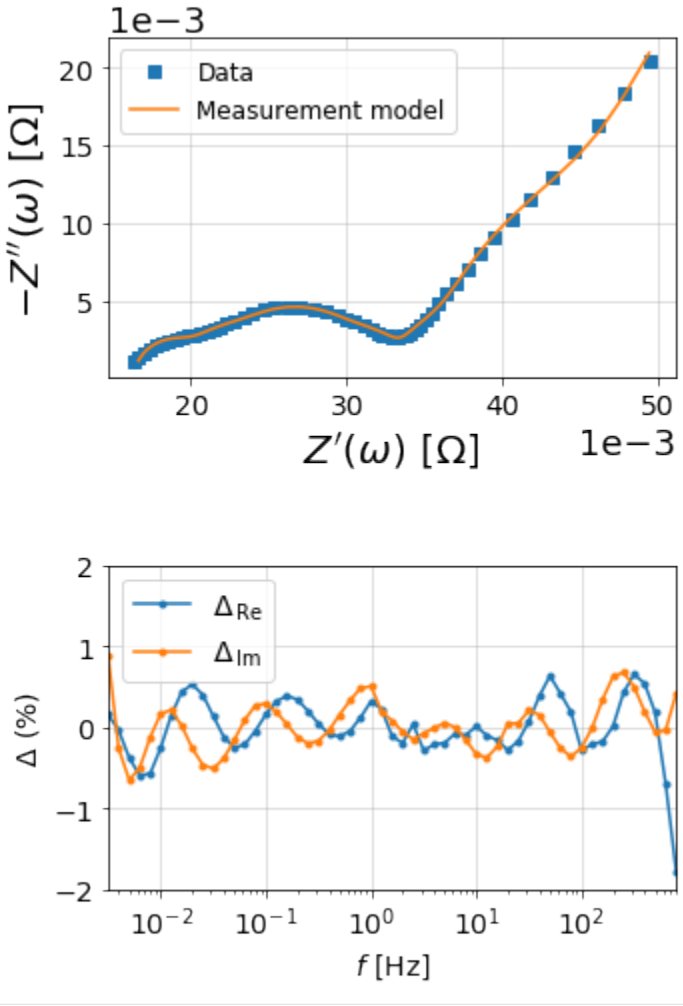
```
    C_1 = 1.00e+03 [F]
    R_2 = 3.00e-03 [Ohm]
    C_2 = 2.51e+02 [F]
    R_3 = 3.00e-03 [Ohm]
    C_3 = 6.31e+01 [F]
    R_4 = 3.00e-03 [Ohm]
    C_4 = 1.58e+01 [F]
    R_5 = 3.00e-03 [Ohm]
    C_5 = 3.98e+00 [F]
    R_6 = 3.00e-03 [Ohm]
    C_6 = 1.00e+00 [F]
    R_7 = 3.00e-03 [Ohm]
    C_7 = 2.51e-01 [F]
    R_8 = 3.00e-03 [Ohm]
    C_8 = 6.31e-02 [F]
    R_9 = 3.00e-03 [Ohm]
    C_9 = 1.58e-02 [F]
    R_1 = 3.00e-03 [Ohm]
    C_1 = 3.98e-03 [F]

Fit parameters:
    R_0 = 1.36e-02  (+/- 8.23e+05) [Ohm]
    R_1 = 1.06e-01  (+/- 7.66e-03) [Ohm]
    C_1 = 2.81e+03  (+/- 5.00e+01) [F]
    R_2 = 1.12e-02  (+/- 2.56e-04) [Ohm]
    C_2 = 1.51e+03  (+/- 4.39e+01) [F]
    R_3 = 3.02e-03  (+/- 1.51e-04) [Ohm]
    C_3 = 6.50e+02  (+/- 7.27e+01) [F]
    R_4 = 1.09e-05  (+/- 7.03e+01) [Ohm]
    C_4 = 2.76e-02  (+/- 2.38e+05) [F]
    R_5 = 2.87e-03  (+/- 4.34e-03) [Ohm]
    C_5 = 6.33e+01  (+/- 1.35e+02) [F]
    R_6 = 6.50e-03  (+/- 3.53e-03) [Ohm]
    C_6 = 4.37e+00  (+/- 1.09e+00) [F]
    R_7 = 3.25e-03  (+/- 4.16e-04) [Ohm]
    C_7 = 1.64e+00  (+/- 1.89e-01) [F]
    R_8 = 6.28e-04  (+/- 2.25e-03) [Ohm]
    C_8 = 1.21e+02  (+/- 8.47e+02) [F]
    R_9 = 3.82e-03  (+/- 1.82e-04) [Ohm]
    C_9 = 1.84e-01  (+/- 1.99e-02) [F]
    R_1 = 2.70e-03  (+/- 8.23e+05) [Ohm]
    C_1 = 3.08e-07  (+/- 1.88e+02) [F]
```

[5]:
```
from impedance.visualization import plot_nyquist, plot_residuals

res_meas_real = (Z - meas_model.predict(f)).real/np.abs(Z)
res_meas_imag = (Z - meas_model.predict(f)).imag/np.abs(Z)

fig = plt.figure(figsize=(5,8))
gs = fig.add_gridspec(3, 1)
ax1 = fig.add_subplot(gs[:2,:])
ax2 = fig.add_subplot(gs[2,:])

# plot original data
plot_nyquist(ax1, Z, fmt='s')
```

(continues on next page)

(continued from previous page)

```
# plot measurement model
plot_nyquist(ax1, meas_model.predict(f), fmt='-', scale=1e3, units='\Omega')

ax1.legend(['Data', 'Measurement model'], loc=2, fontsize=12)

# Plot residuals
plot_residuals(ax2, f, res_meas_real, res_meas_imag, y_limits=(-2,2))

plt.tight_layout()
plt.show()
```



The Lin-KK method

The lin-KK method from Schönleber et al. [1] is a quick test for checking the validity of EIS data. The validity of an impedance spectrum is analyzed by its reproducibility by a Kramers-Kronig (KK) compliant equivalent circuit. In particular, the model used in the lin-KK test is an ohmic resistor, R_{Ohm}, and M RC elements.

$$\hat{Z} = R_{Ohm} + \sum_{k=1}^{M} \frac{R_k}{1 + j\omega\tau_k}$$

The M time constants, τ_k, are distributed logarithmically,

$$\tau_1 = \frac{1}{\omega_{max}}; \tau_M = \frac{1}{\omega_{min}}; \tau_k = 10^{\log(\tau_{min}) + \frac{k-1}{M-1}\log(\frac{\tau_{max}}{\tau_{min}})}$$

and are not fit during the test (only R_{Ohm} and R_k are free parameters).

In order to prevent under- or over-fitting, Schönleber et al. propose using the ratio of positive resistor mass to negative resistor mass as a metric for finding the optimal number of RC elements.

$$\mu = 1 - \frac{\sum_{R_k \geq 0} |R_k|}{\sum_{R_k < 0} |R_k|}$$

The argument `c` defines the cutoff value for μ. The algorithm starts at `M = 3` and iterates up to `max_M` until a $\mu < c$ is reached. The default of 0.85 is simply a heuristic value based off of the experience of Schönleber et al.

If the argument `c` is `None`, then the automatic determination of RC elements is turned off and the solution is calculated for `max_M` RC elements. This manual mode should be used with caution as under- and over-fitting should be avoided.

[1] Schönleber, M. et al. A Method for Improving the Robustness of linear Kramers-Kronig Validity Tests. Electrochimica Acta 131, 20–27 (2014) doi: 10.1016/j.electacta.2014.01.034.

[6]:
```
import matplotlib.pyplot as plt
import numpy as np

import sys
sys.path.append('../../../')

from impedance.validation import linKK
```

[7]:
```
# Load data from the example EIS result
f, Z = preprocessing.readCSV('../../../data/exampleData.csv')

# keep only the impedance data in the first quandrant
f, Z = preprocessing.ignoreBelowX(f, Z)

mask = f < 1000
f = f[mask]
Z = Z[mask]
```

[8]:
```
M, mu, Z_linKK, res_real, res_imag = linKK(f, Z, c=.5, max_M=100, fit_type='complex',␣
↪add_cap=True)

print('\nCompleted Lin-KK Fit\nM = {:d}\nmu = {:.2f}'.format(M, mu))
```

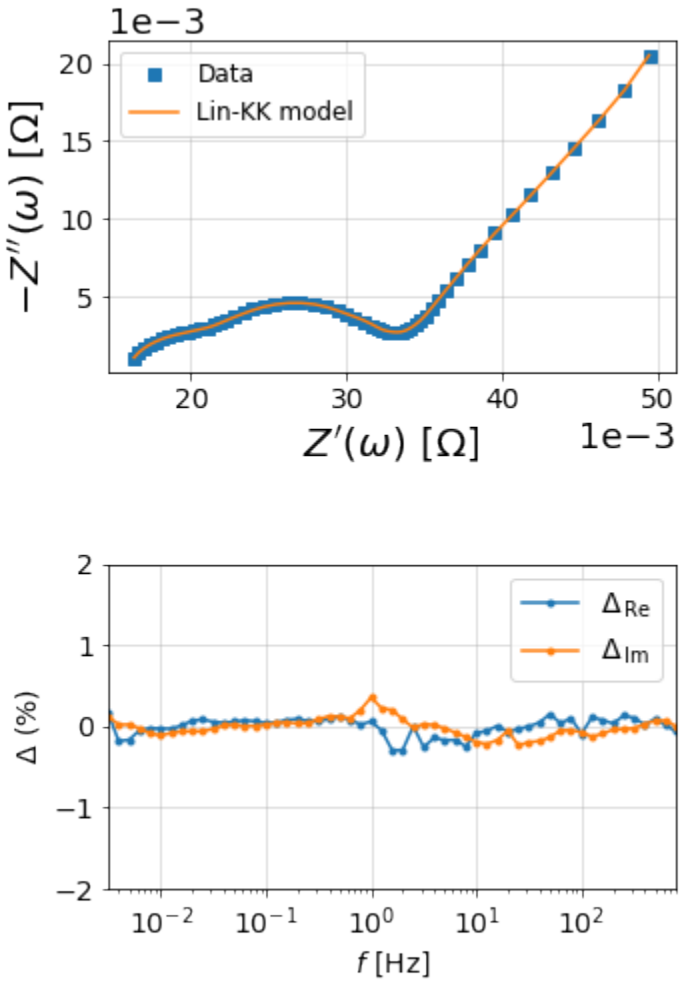
```
10 1.0 8.144660459071784e-05
20 0.8929547569244768 5.1002168389472366e-05

Completed Lin-KK Fit
M = 26
mu = 0.31
```

[9]:
```
from impedance.visualization import plot_nyquist, plot_residuals

fig = plt.figure(figsize=(5,8))
gs = fig.add_gridspec(3, 1)
ax1 = fig.add_subplot(gs[:2,:])
ax2 = fig.add_subplot(gs[2,:])
```

(continues on next page)

(continued from previous page)

```
# plot original data
plot_nyquist(ax1, Z, fmt='s')

# plot measurement model
plot_nyquist(ax1, Z_linKK, fmt='-', scale=1e3, units='\Omega')

ax1.legend(['Data', 'Lin-KK model'], loc=2, fontsize=12)

# Plot residuals
plot_residuals(ax2, f, res_real, res_imag, y_limits=(-2,2))

plt.tight_layout()
plt.show()
```



2.2.5 Looping through and fitting multiple impedance data sets

[1]:
```
import glob
import numpy as np
import os
```

1. Find all files that match a specified pattern

Using a search string to find .z files that contain "Circuit" at the beginning and EIS towards the end

[2]:
```
directory = r'../../../data/'
all_files = glob.glob(os.path.join(directory, 'Circuit*EIS*.z'))
all_files
```

[2]:
```
['../../../data/Circuit3_EIS_1.z',
 '../../../data/Circuit1_EIS_2.z',
 '../../../data/Circuit2_EIS_1.z',
 '../../../data/Circuit3_EIS_2.z',
 '../../../data/Circuit1_EIS_1.z',
 '../../../data/Circuit2_EIS_2.z']
```

2. Use preprocessing module to read in ZPlot data

[3]:
```
from impedance import preprocessing
# Initialize some empty lists for the frequencies and Z data
freqs = []
Zs = []

# Now loop through file names in our list and extract data one by one
for filename in all_files:
    f, Z = preprocessing.readZPlot(filename)
    freqs.append(f)
    Zs.append(Z)

# Check to see if we extracted data for all the files
print(np.shape(Zs), np.shape(all_files))
```

```
(6,) (6,)
```

3. Create a list of circuit models

[4]:
```
from impedance.models.circuits import CustomCircuit
# This data comes from dummy circuits I made to check measurement bias in
# our potentiostat, so I know a priori its an R-RC circuit

circuits = []

circ_string = 'R0-p(R1,C1)'
initial_guess = [100, 400, 1e-5]

# Now loop through data list to create circuits and fit them
for f, Z, filename in zip(freqs, Zs, all_files):
    name = filename.split('/')[-1]
    circuit = CustomCircuit(circ_string, initial_guess=initial_guess, name=name)
    circuit.fit(f, Z)
    circuits.append(circuit)
```

We now have a list of our circuit class objects, all fit to different sets of data. As you may notice from the file names, there are three unique circuits each with a replicate set of data. We expect each of the replicates to fit similarly.

[5]:
```
for circuit in circuits:
    print(circuit)
```

```
Name: Circuit3_EIS_1.z
Circuit string: R0-p(R1,C1)
Fit: True

Initial guesses:
     R0 = 1.00e+02 [Ohm]
     R1 = 4.00e+02 [Ohm]
     C1 = 1.00e-05 [F]

Fit parameters:
     R0 = 1.51e+03  (+/- 2.62e+00) [Ohm]
     R1 = 4.63e+03  (+/- 3.14e+00) [Ohm]
     C1 = 2.02e-08  (+/- 5.39e-11) [F]


Name: Circuit1_EIS_2.z
Circuit string: R0-p(R1,C1)
Fit: True

Initial guesses:
     R0 = 1.00e+02 [Ohm]
     R1 = 4.00e+02 [Ohm]
     C1 = 1.00e-05 [F]

Fit parameters:
     R0 = 2.91e+01  (+/- 3.58e-02) [Ohm]
     R1 = 4.67e+01  (+/- 4.64e-02) [Ohm]
     C1 = 1.04e-05  (+/- 2.91e-08) [F]


Name: Circuit2_EIS_1.z
Circuit string: R0-p(R1,C1)
Fit: True

Initial guesses:
     R0 = 1.00e+02 [Ohm]
     R1 = 4.00e+02 [Ohm]
     C1 = 1.00e-05 [F]

Fit parameters:
     R0 = 1.50e+02  (+/- 3.23e-01) [Ohm]
     R1 = 5.02e+02  (+/- 3.57e-01) [Ohm]
     C1 = 3.12e-08  (+/- 7.79e-11) [F]


Name: Circuit3_EIS_2.z
Circuit string: R0-p(R1,C1)
Fit: True

Initial guesses:
     R0 = 1.00e+02 [Ohm]
     R1 = 4.00e+02 [Ohm]
     C1 = 1.00e-05 [F]
```

(continues on next page)

(continued from previous page)

```
Fit parameters:
     R0 = 1.51e+03  (+/- 2.68e+00) [Ohm]
     R1 = 4.63e+03  (+/- 3.21e+00) [Ohm]
     C1 = 2.02e-08  (+/- 5.52e-11) [F]


Name: Circuit1_EIS_1.z
Circuit string: R0-p(R1,C1)
Fit: True

Initial guesses:
     R0 = 1.00e+02 [Ohm]
     R1 = 4.00e+02 [Ohm]
     C1 = 1.00e-05 [F]

Fit parameters:
     R0 = 2.91e+01  (+/- 3.63e-02) [Ohm]
     R1 = 4.67e+01  (+/- 4.69e-02) [Ohm]
     C1 = 1.04e-05  (+/- 2.95e-08) [F]


Name: Circuit2_EIS_2.z
Circuit string: R0-p(R1,C1)
Fit: True

Initial guesses:
     R0 = 1.00e+02 [Ohm]
     R1 = 4.00e+02 [Ohm]
     C1 = 1.00e-05 [F]

Fit parameters:
     R0 = 1.50e+02  (+/- 3.19e-01) [Ohm]
     R1 = 5.02e+02  (+/- 3.53e-01) [Ohm]
     C1 = 3.12e-08  (+/- 7.70e-11) [F]
```

Now we'll get the impedance predicted by the fit parameters

[6]:
```
fits = []
for f, circuit in zip(freqs, circuits):
    fits.append(circuit.predict(f))
```

4. Plot the data and fits

Now we can visualize the data and fits. For now we'll place them all on the same axis

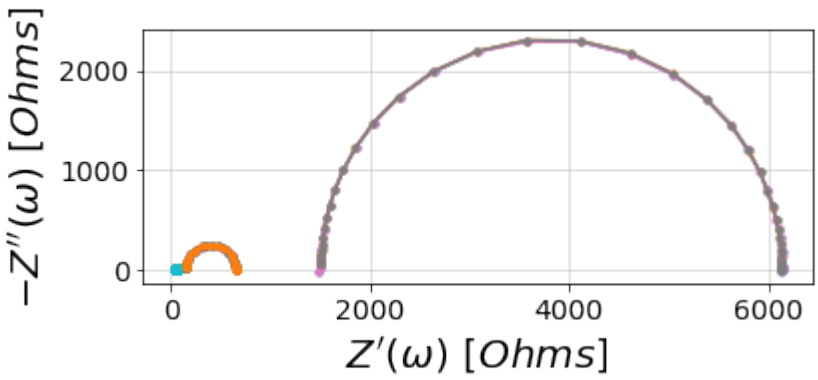
[7]:
```
import matplotlib.pyplot as plt
from impedance.visualization import plot_nyquist, plot_bode
```

[8]:
```
fig, ax = plt.subplots()

for fit, Z in zip(fits, Zs):
    # Plotting data
    plot_nyquist(ax, Z)
```

(continues on next page)

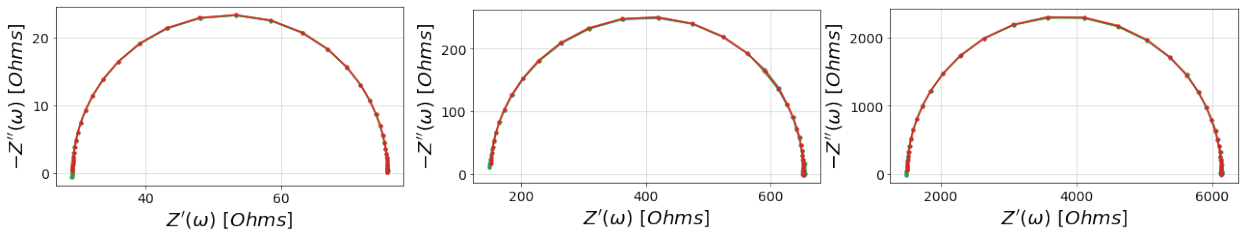
(continued from previous page)

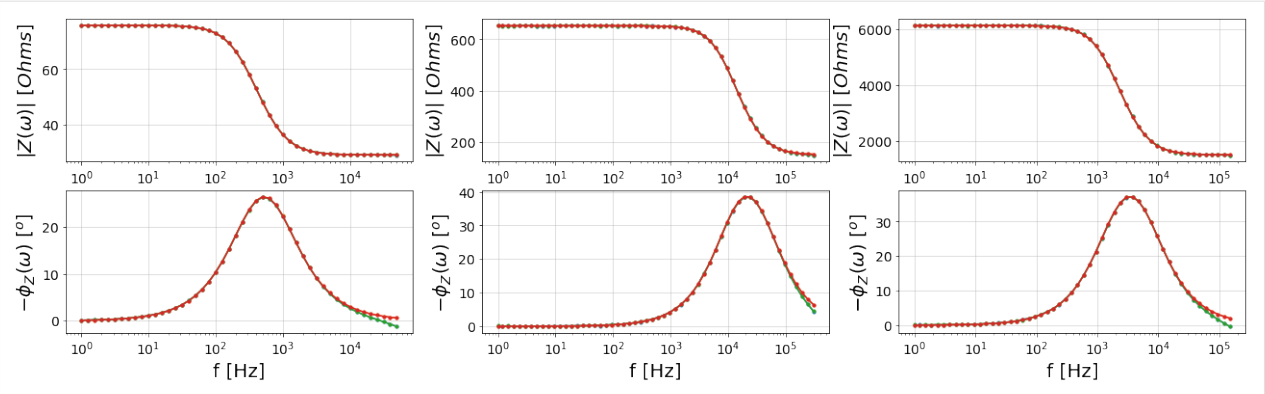
```
    # Plotting fit
    plot_nyquist(ax, fit)

plt.show()
```



Since the circuits have different orders of magnitude impedance, this looks bad so let's put each pair of data on separate axes.

[9]:
```
# Nyquist plots
fig, axes = plt.subplots(ncols=3, figsize=(22,6))
for circuit, Z, fit in zip(circuits, Zs, fits):
    n = int(circuit.name.split('Circuit')[-1].split('_')[0])
    plot_nyquist(axes[n - 1], Z)
    plot_nyquist(axes[n - 1], fit)

# Bode plots
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(22,6))
for circuit, f, Z, fit in zip(circuits, freqs, Zs, fits):
    n = int(circuit.name.split('Circuit')[-1].split('_')[0])
    plot_bode([axes[0][n - 1], axes[1][n - 1]], f, Z)
    plot_bode([axes[0][n - 1], axes[1][n - 1]], f, fit)

plt.show()
```





2.3 Preprocessing

Methods for preprocessing impedance data from instrument files

`impedance.preprocessing.`**`cropFrequencies`**(*frequencies*, *Z*, *freqmin=0*, *freqmax=None*)
Trim out all data points below the X-axis

Parameters

frequencies [np.ndarray] Array of frequencies

Z [np.ndarray of complex numbers] Array of complex impedances

freqmin [float] Minimum frequency, omit for no lower frequency limit

freqmax [float] Max frequency, omit for no upper frequency limit

Returns

frequencies_final [np.ndarray] Array of frequencies after filtering

Z_final [np.ndarray of complex numbers] Array of complex impedances after filtering

`impedance.preprocessing.`**`ignoreBelowX`**(*frequencies*, *Z*)
Trim out all data points below the X-axis

Parameters

frequencies [np.ndarray] Array of frequencies

Z [np.ndarray of complex numbers] Array of complex impedances

Returns

frequencies [np.ndarray] Array of frequencies after filtering

Z [np.ndarray of complex numbers] Array of complex impedances after filtering

`impedance.preprocessing.`**`readAutolab`**(*filename*)
function for reading comma-delimited files from Autolab

Parameters

filename: string Filename of file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readBioLogic**(*filename*)
function for reading the .mpt file from Biologic EC-lab software

Parameters

filename: string Filename of .mpt file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readCHInstruments**(*filename*)
function for reading the .txt file from CHInstruments

Parameters

filename: string Filename of .txt file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readCSV**(*filename*)
function for reading plain csv files

Parameters

filename: string Filename of .csv file to extract impedance data from where the file has three columns (frequency, Z_real, Z_imag)

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readFile**(*filename*, *instrument=None*)
A wrapper for reading in many common types of impedance files

Parameters

filename: string Filename to extract impedance data from

instrument: string Type of instrument file

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readGamry**(*filename*)
function for reading the .DTA file from Gamry

Parameters

filename: string Filename of .DTA file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readParstat**(*filename*)
function for reading the .txt file from Parstat

Parameters

filename: string Filename of .txt file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readPowerSuite**(*filename*)
function for reading the .txt file from Parstat

Parameters

filename: string Filename of .txt file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readVersaStudio**(*filename*)
function for reading the .PAR file from VersaStudio

Parameters

filename: string Filename of .PAR file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**readZPlot**(*filename*)
function for reading the .z file from Scribner's ZPlot

Parameters

filename: string Filename of .z file to extract impedance data from

Returns

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

impedance.preprocessing.**saveCSV**(*filename*, *frequencies*, *impedances*, ***kwargs*)
saves frequencies and impedances to a csv

Parameters

filename: string Filename of .csv file to save impedance data to

frequencies [np.ndarray] Array of frequencies

impedance [np.ndarray of complex numbers] Array of complex impedances

kwargs Keyword arguments passed to np.savetxt

2.4 Validation

Interpreting EIS data fundamentally relies on the the system conforming to conditions of causality, linearity, and stability. For an example of how the adherence to the Kramers-Kronig relations, see the *Validation Example Jupyter Notebook*

2.4.1 Lin-KK method

Validating your data with the lin-KK model requires fitting an optimal number of RC-elements and analysis of the residual errors.

`impedance.validation.`**`calc_mu`**(*Rs*)
Calculates mu for use in LinKK

`impedance.validation.`**`eval_linKK`**(*elements*, *ts*, *f*)
Builds a circuit of RC elements to be used in LinKK

`impedance.validation.`**`fit_linKK`**(*f*, *ts*, *M*, *Z*, *fit_type='real'*, *add_cap=False*)
Fits the linKK model using linear regression

Parameters

f: np.ndarray measured frequencies

ts: np.ndarray logarithmically spaced time constants of RC elements

M: int the number of RC elements

Z: np.ndarray of complex numbers measured impedances

fit_type: str selects which components of data are fit ('real', 'imag', or 'complex')

add_cap: bool option to add a serial capacitance that helps validate data with no low-frequency intercept

Returns

elements: np.ndarray values of fit R_k in RC elements and series R_0, L, and optionally C.

mu: np.float under- or over-fitting measure

Notes

Since we have a system of equations, $Ax = b$, that's linear wrt R_k, we can fit the model by calculating the pseudo-inverse of A. Ax is our model fit, $\hat{Z}$, and b is the normalized real or imaginary component of the impedance data, $Re(Z)/|Z|$ or $Im(Z)/|Z|$, respectively.

$\hat{Z} = R_0 + \sum_{k=1}^{M}(R_k/|Z|(1 + j * w * \tau_k))$. x is an (M+1) $\times$ 1 matrix where the first row contains R_0 and subsequent rows contain R_k values. A is an N $\times$ (M+1) matrix, where N is the number of data points, and M is the number of RC elements.

Examples

Fitting the real part of data, the first column of A contains values of $\frac{1}{|Z|}$, the second column contains $Re(1/|Z|(1 + j * w * \tau_1))$, the third contains $Re(1/|Z|(1 + j * w * \tau_2))$ and so on. The R_k values within the x matrix are found using `numpy.linalg.pinv` when fit_type = 'real' or 'imag'. When fit_type = 'complex' the coefficients are found "manually" using $r = ||A'x - b'||^2 + ||A''x - b'||^2$ according to Eq 14 of Schonleber [1].

[1] Schönleber, M. et al. A Method for Improving the Robustness of linear Kramers-Kronig Validity Tests. Electrochimica Acta 131, 20–27 (2014) doi: 10.1016/j.electacta.2014.01.034.

impedance.validation.**get_tc_distribution**(*f*, *M*)
Returns the distribution of time constants for the linKK method

impedance.validation.**linKK**(*f*, *Z*, *c=0.85*, *max_M=50*, *fit_type='real'*, *add_cap=False*)
A method for implementing the Lin-KK test for validating linearity [1]

Parameters

f: np.ndarray measured frequencies

Z: np.ndarray of complex numbers measured impedances

c: np.float cutoff for mu

max_M: int the maximum number of RC elements

fit_type: str selects which components of data are fit ('real', 'imag', or 'complex')

add_cap: bool option to add a serial capacitance that helps validate data with no low-frequency intercept

Returns

M: int number of RC elements used

mu: np.float under- or over-fitting measure

Z_fit: np.ndarray of complex numbers impedance of fit at input frequencies

resids_real: np.ndarray real component of the residuals of the fit at input frequencies

resids_imag: np.ndarray imaginary component of the residuals of the fit at input frequencies

Notes

The lin-KK method from Schönleber et al. [1] is a quick test for checking the validity of EIS data. The validity of an impedance spectrum is analyzed by its reproducibility by a Kramers-Kronig (KK) compliant equivalent circuit. In particular, the model used in the lin-KK test is an ohmic resistor, R_{Ohm}, and M RC elements.

$$\hat{Z} = R_{Ohm} + \sum_{k=1}^{M} \frac{R_k}{1 + j\omega\tau_k}$$

The M time constants, τ_k, are distributed logarithmically,

$$\tau_1 = \frac{1}{\omega_{max}}; \tau_M = \frac{1}{\omega_{min}}; \tau_k = 10^{\log(\tau_{min}) + \frac{k-1}{M-1}\log(\frac{\tau_{max}}{\tau_{min}})}$$

and are not fit during the test (only R_{Ohm} and R_k are free parameters).

In order to prevent under- or over-fitting, Schönleber et al. propose using the ratio of positive resistor mass to negative resistor mass as a metric for finding the optimal number of RC elements.

$$\mu = 1 - \frac{\sum_{R_k \geq 0} |R_k|}{\sum_{R_k < 0} |R_k|}$$

The argument `c` defines the cutoff value for μ. The algorithm starts at `M = 3` and iterates up to `max_M` until a $\mu < c$ is reached. The default of 0.85 is simply a heuristic value based off of the experience of Schönleber et al., but a lower value may give better results.

If the argument `c` is `None`, then the automatic determination of RC elements is turned off and the solution is calculated for `max_M` RC elements. This manual mode should be used with caution as under- and over-fitting should be avoided.

[1] Schönleber, M. et al. A Method for Improving the Robustness of linear Kramers-Kronig Validity Tests. Electrochimica Acta 131, 20–27 (2014) doi: 10.1016/j.electacta.2014.01.034.

`impedance.validation.`**`residuals_linKK`**(*elements*, *ts*, *Z*, *f*, *residuals='real'*)

Calculates the residual between the data and a LinKK fit

2.5 Circuits

class `impedance.models.circuits.circuits.`**`BaseCircuit`**(*initial_guess=[]*, *constants=None*, *name=None*)

Base class for equivalent circuit models

Methods

fit(frequencies, impedance[, bounds, ...])	Fit the circuit model
get_param_names()	Converts circuit string to names and units
load(filepath[, fitted_as_initial])	Imports a model from JSON
plot([ax, f_data, Z_data, kind])	visualizes the model and optional data as a nyquist,
predict(frequencies[, use_initial])	Predict impedance using an equivalent circuit model
save(filepath)	Exports a model to JSON

`fit`(*frequencies*, *impedance*, *bounds=None*, *weight_by_modulus=False*, ***kwargs*)

Fit the circuit model

Parameters

frequencies: numpy array Frequencies

impedance: numpy array of dtype 'complex128' Impedance values to fit

bounds: 2-tuple of array_like, optional Lower and upper bounds on parameters. Defaults to bounds on all parameters of 0 and np.inf, except the CPE alpha which has an upper bound of 1

weight_by_modulus [bool, optional] Uses the modulus of each data (|Z|) as the weighting factor. Standard weighting scheme when experimental variances are unavailable. Only applicable when global_opt = False

kwargs Keyword arguments passed to impedance.models.circuits.fitting.circuit_fit, and subsequently to scipy.optimize.curve_fit or scipy.optimize.basinhopping

Returns

self: returns an instance of self

`get_param_names`()

Converts circuit string to names and units

`load`(*filepath*, *fitted_as_initial=False*)

Imports a model from JSON

Parameters

filepath: str filepath to JSON file to load model from

fitted_as_initial: bool If true, loads the model's fitted parameters as initial guesses

Otherwise, loads the model's initial and fitted parameters as a completed model

plot(*ax=None*, *f_data=None*, *Z_data=None*, *kind='altair'*, ***kwargs*)

visualizes the model and optional data as a nyquist, bode, or altair (interactive) plots

Parameters

ax: matplotlib.axes axes to plot on

f_data: np.array of type float Frequencies of input data (for Bode plots)

Z_data: np.array of type complex Impedance data to plot

kind: {'altair', 'nyquist', 'bode'} type of plot to visualize

Returns

ax: matplotlib.axes axes of the created nyquist plot

Other Parameters

****kwargs** [optional]

If kind is 'nyquist' or 'bode', used to specify additional *matplotlib.pyplot.Line2D* properties like linewidth, line color, marker color, and labels.

If kind is 'altair', used to specify nyquist height as *size*

predict(*frequencies*, *use_initial=False*)

Predict impedance using an equivalent circuit model

Parameters

frequencies: ndarray of numeric dtype

use_initial: boolean If true and the model was previously fit use the initial parameters instead

Returns

impedance: ndarray of dtype 'complex128' Predicted impedance

save(*filepath*)

Exports a model to JSON

Parameters

filepath: str Destination for exporting model object

class impedance.models.circuits.circuits.**CustomCircuit**(*circuit=''*, ***kwargs*)

Methods

fit(frequencies, impedance[, bounds, ...])	Fit the circuit model
get_param_names()	Converts circuit string to names and units
load(filepath[, fitted_as_initial])	Imports a model from JSON
plot([ax, f_data, Z_data, kind])	visualizes the model and optional data as a nyquist,
predict(frequencies[, use_initial])	Predict impedance using an equivalent circuit model

Continued on next page

Table 2 – continued from previous page

save(filepath)	Exports a model to JSON

class impedance.models.circuits.circuits.**Randles**(*CPE=False*, ***kwargs*)
A Randles circuit model class

Methods

fit(frequencies, impedance[, bounds, ...])	Fit the circuit model
get_param_names()	Converts circuit string to names and units
load(filepath[, fitted_as_initial])	Imports a model from JSON
plot([ax, f_data, Z_data, kind])	visualizes the model and optional data as a nyquist,
predict(frequencies[, use_initial])	Predict impedance using an equivalent circuit model
save(filepath)	Exports a model to JSON

2.6 Circuit Elements

impedance.models.circuits.elements.**C**(p, f)
defines a capacitor

$$Z = \frac{1}{C \times j2\pi f}$$

impedance.models.circuits.elements.**CPE**(p, f)
defines a constant phase element

Notes

$$Z = \frac{1}{Q \times (j2\pi f)^{\alpha}}$$

where Q = p[0] and α = p[1].

impedance.models.circuits.elements.**G**(p, f)
defines a Gerischer Element as represented in [1]

Notes

$$Z = \frac{R_G}{\sqrt{1 + j\,2\pi f\,t_G}}$$

where R_G = p[0] and t_G = p[1]

Gerischer impedance is also commonly represented as [2]:

$$Z = \frac{Z_o}{\sqrt{K + j\,2\pi f}}$$

where $Z_o = \frac{R_G}{\sqrt{t_G}}$ and $K = \frac{1}{t_G}$ with units $\Omega sec^{1/2}$ and sec^{-1} , respectively.

[1] Y. Lu, C. Kreller, and S.B. Adler, Journal of The Electrochemical Society, 156, B513-B525 (2009) doi:10.1149/1.3079337.

[2] M. González-Cuenca, W. Zipprich, B.A. Boukamp, G. Pudmich, and F. Tietz, Fuel Cells, 1, 256-264 (2001) doi:10.1016/0013-4686(93)85083-B.

impedance.models.circuits.elements.**Gs**(*p*, *f*)
defines a finite-length Gerischer Element as represented in [1]

Notes

$$Z = \frac{R_G}{\sqrt{1 + j\,2\pi f\,t_G}\,tanh(\phi\sqrt{1 + j\,2\pi f\,t_G})}$$

where R_G = p[0], t_G = p[1] and ϕ = p[2]

[1] R.D. Green, C.C Liu, and S.B. Adler, Solid State Ionics, 179, 647-660 (2008) doi:10.1016/j.ssi.2008.04.024.

impedance.models.circuits.elements.**K**(*p*, *f*)
An RC element for use in lin-KK model

Notes

$$Z = \frac{R}{1 + j\omega\tau_k}$$

impedance.models.circuits.elements.**L**(*p*, *f*)
defines an inductor

$$Z = L \times j2\pi f$$

impedance.models.circuits.elements.**La**(*p*, *f*)
defines a modified inductance element as represented in [1]

Notes

$$Z = L \times (j2\pi f)^{\alpha}$$

where L = p[0] and α = p[1]

[1] EC-Lab Application Note 42, BioLogic Instruments (2019).

impedance.models.circuits.elements.**R**(*p*, *f*)
defines a resistor

Notes

$$Z = R$$

impedance.models.circuits.elements.**T**(*p*, *f*)
A macrohomogeneous porous electrode model from Paasch et al. [1]

Notes

$$Z = A\frac{\coth\beta}{\beta} + B\frac{1}{\beta\sinh\beta}$$

where

$$A = d\frac{\rho_1^2+\rho_2^2}{\rho_1+\rho_2} \quad B = d\frac{2\rho_1\rho_2}{\rho_1+\rho_2}$$

and

$$\beta = (a+j\omega b)^{1/2} \quad a = \frac{kd^2}{K} \quad b = \frac{d^2}{K}$$

[1] G. Paasch, K. Micka, and P. Gersdorf, Electrochimica Acta, 38, 2653–2662 (1993) doi: 10.1016/0013-4686(93)85083-B.

impedance.models.circuits.elements.**TLMQ**(*p*, *f*)
Simplified transmission-line model as defined in Eq. 11 of [1]

Notes

impedance.models.circuits.elements.**W**(*p*, *f*)
defines a semi-infinite Warburg element

Notes

$$Z = \frac{A_W}{\sqrt{2\pi f}}(1-j)$$

impedance.models.circuits.elements.**Wo**(*p*, *f*)
defines an open (finite-space) Warburg element

Notes

$$Z = \frac{Z_0}{\sqrt{j\omega\tau}}\coth\sqrt{j\omega\tau}$$

where Z_0 = p[0] (Ohms) and τ = p[1] (sec) = $\frac{L^2}{D}$

impedance.models.circuits.elements.**Ws**(*p*, *f*)
defines a short (finite-length) Warburg element

Notes

$$Z = \frac{Z_0}{\sqrt{j\omega\tau}}\tanh\sqrt{j\omega\tau}$$

where Z_0 = p[0] (Ohms) and τ = p[1] (sec) = $\frac{L^2}{D}$

impedance.models.circuits.elements.**element_metadata**(*num_params*, *units*)
decorator to store metadata for a circuit element

Parameters

num_params [int] number of parameters for an element

units [list of str] list of units for the element parameters

impedance.models.circuits.elements.**p**(*parallel*)
adds elements in parallel

Notes

$$Z = \frac{1}{\frac{1}{Z_1} + \frac{1}{Z_2} + ... + \frac{1}{Z_n}}$$

impedance.models.circuits.elements.**s**(*series*)
sums elements in series

Notes

$$Z = Z_1 + Z_2 + ... + Z_n$$

2.7 Fitting

impedance.models.circuits.fitting.**buildCircuit**(*circuit*, *frequencies*, **parameters*, *constants=None*, *eval_string=''*, *index=0*)
recursive function that transforms a circuit, parameters, and frequencies into a string that can be evaluated

Parameters

circuit: str

frequencies: list/tuple/array of floats

parameters: list/tuple/array of floats

constants: dict

Returns

eval_string: str Python expression for calculating the resulting fit

index: int Tracks parameter index through recursive calling of the function

impedance.models.circuits.fitting.**calculateCircuitLength**(*circuit*)
Calculates the number of elements in the circuit.

Parameters

circuit [str] Circuit string.

Returns

length [int] Length of circuit.

impedance.models.circuits.fitting.**check_and_eval**(*element*)
Checks if an element is valid, then evaluates it.

Parameters

element [str] Circuit element.

Returns

Evaluated element.

Raises

ValueError Raised if an element is not in the list of allowed elements.

impedance.models.circuits.fitting.**circuit_fit**(*frequencies*, *impedances*, *circuit*, *initial_guess*, *constants={}*, *bounds=None*, *weight_by_modulus=False*, *global_opt=False*, ***kwargs*)

Main function for fitting an equivalent circuit to data.

By default, this function uses scipy.optimize.curve_fit to fit the equivalent circuit. This function generally works well for simple circuits. However, the final results may be sensitive to the initial conditions for more complex circuits. In these cases, the scipy.optimize.basinhopping global optimization algorithm can be used to attempt a better fit.

Parameters

frequencies [numpy array] Frequencies

impedances [numpy array of dtype 'complex128'] Impedances

circuit [string] String defining the equivalent circuit to be fit

initial_guess [list of floats] Initial guesses for the fit parameters

constants [dictionary, optional] Parameters and their values to hold constant during fitting (e.g. {"RO": 0.1}). Defaults to {}

bounds [2-tuple of array_like, optional] Lower and upper bounds on parameters. Defaults to bounds on all parameters of 0 and np.inf, except the CPE alpha which has an upper bound of 1

weight_by_modulus [bool, optional] Uses the modulus of each data (|Z|) as the weighting factor. Standard weighting scheme when experimental variances are unavailable. Only applicable when global_opt = False

global_opt [bool, optional] If global optimization should be used (uses the basinhopping algorithm). Defaults to False

kwargs Keyword arguments passed to scipy.optimize.curve_fit or scipy.optimize.basinhopping

Returns

p_values [list of floats] best fit parameters for specified equivalent circuit

p_errors [list of floats] one standard deviation error estimates for fit parameters

Notes

Need to do a better job of handling errors in fitting. Currently, an error of -1 is returned.

impedance.models.circuits.fitting.**extract_circuit_elements**(*circuit*)

Extracts circuit elements from a circuit string.

Parameters

circuit [str] Circuit string.

Returns

extracted_elements [list] list of extracted elements.

impedance.models.circuits.fitting.**rmse**(*a*, *b*)
A function which calculates the root mean squared error between two vectors.

Notes

$$RMSE = \sqrt{\frac{1}{n}(a-b)^2}$$

impedance.models.circuits.fitting.**set_default_bounds**(*circuit*, *constants={}*)
This function sets default bounds for optimization.

set_default_bounds sets bounds of 0 and np.inf for all parameters, except the CPE and La alphas which have an upper bound of 1.

Parameters

circuit [string] String defining the equivalent circuit to be fit

constants [dictionary, optional] Parameters and their values to hold constant during fitting (e.g. {"RO": 0.1}). Defaults to {}

Returns

bounds [2-tuple of array_like] Lower and upper bounds on parameters.

impedance.models.circuits.fitting.**wrapCircuit**(*circuit*, *constants*)
wraps function so we can pass the circuit string

2.8 Frequently Asked Questions

2.8.1 What method does impedance.py use for fitting equivalent circuit models?

By default, fitting is performed by non-linear least squares regression of the circuit model to impedance data via curve_fit from the *scipy.optimize* package.[1] Real and imaginary components are fit simultaneously with uniform weighting, i.e. the objective function to minimize is,

$$\chi^2 = \sum_{n=0}^{N} [Z'_{data}(\omega_n) - Z'_{model}(\omega_n)]^2 + [Z''_{data}(\omega_n) - Z''_{model}(\omega_n)]^2$$

where N is the number of frequencies and Z' and Z'' are the real and imaginary components of the impedance, respectively. The default optimization method is the Levenberg-Marquardt algorithm (`method='lm'`) for unconstrained problems and the Trust Region Reflective algorithm (`method='trf'`) if bounds are provided. See the SciPy documentation for more details and options.

While the default method converges quickly and often yields acceptable fits, the results may be sensitive to the initial conditions. EIS fitting can be prone to this issue given the high dimensionality of typical equivalent circuit models. Global optimization algorithms attempt to search the entire parameter landscape to minimize the error. By setting `global_opt=True` in `circuit_fit`, `impedance.py` will use the basinhopping global optimization algorithm (also from the *scipy.optimize* package[1]) instead of `curve_fit`. Note that the computational time may increase.

[1] Virtanen, P., Gommers, R., Oliphant, T.E. et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. Nat Methods 17, 261–272 (2020). doi: 10.1038/s41592-019-0686-2

2.8.2 How do I cite impedance.py?

If you use impedance.py in published work, please consider citing https://joss.theoj.org/papers/10.21105/joss.02349 as

```
@article{Murbach2020,
    doi = {10.21105/joss.02349},
    url = {https://doi.org/10.21105/joss.02349},
    year = {2020},
    publisher = {The Open Journal},
    volume = {5},
    number = {52},
    pages = {2349},
    author = {Matthew D. Murbach and Brian Gerwe and Neal Dawson-Elli and Lok-kun␣
↪Tsui},
    title = {impedance.py: A Python package for electrochemical impedance analysis},
    journal = {Journal of Open Source Software}
}
```

2.8.3 How can I contribute to impedance.py?

First off, thank you for your interest in contributing to the open-source electrochemical community! We're excited to welcome all contributions including suggestions for new features, bug reports/fixes, examples/documentation, and additional impedance analysis functionality.

Feature requests and bug reports

If you want to make a suggestion for a new feature, please make an issue including as much detail as possible. If you're requesting a new circuit element or data file type, there are special issue templates that you can select and use.

Contributing code

The prefered method for contributing code to impedance.py is to fork the repository on GitHub and submit a "pull request" (PR). More detailed information on how to get started developing impedance.py can be found in CONTRIBUTING.md.

Feel free to reach out via GitHub issues with any questions!

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

S

T

W