resistics

Release 1.0.0a3

Neeraj Shah

May 12, 2023

USER GUIDE:

Soon resistics will be upgrading to version 1.0.0. This will be a breaking change versus version 0.0.6. Currently, the newest version is available as a development release for those who are intersted in experimenting with its updated feature set.

Until version 1.0.0 is released as a stable version, the existing documentation for 0.0.6 will remain at resistics.io.

CHAPTER

ONE

WHY?

Resistics has been re-written from the ground up to tackle several limitations of the previous version, namely

- Processing time
- Limited traceability
- Lack of extensibility
- Difficult to maintain

The new version of resistics aims to tackle all of these issues through better coding practises, putting extensibility at the heart of its design and moving to a modern deployment pipeline.

CHAPTER

TWO

WHAT'S NEW?

The literal answer is everything as this is a from scratch rewrite, which has taken some features of the previous version but combined them with new capabilities.

For most users, notable changes are related to configuration of processing flows and the carving out of specific data format readers into a separate package.

Advanced users will be able to take advantage of opportunities to write their own solvers or processors and a greater ability to customise and extend resistics.

Other smaller changes include:

- Moving to JSON for metadata as this is a universal format
- Moving from matplotlib to plotly for plots as they are more interactive

CHAPTER

THREE

WHAT'S MISSING?

The first thing to note is that time series data reader for various formats have been removed from resistics and placed in a sister package named resistics-readers. This is to remove any coupling of data format support to core resistics releases. It is hoped that resistics-readers will receive more community support as knowledge about the various data formats in the magnetotelluric world is distributed around the community.

Statistics are another capability of resistics 0.0.6 that is missing. The intention is to re-introduce these shortly and additionally, make it easier for users to write their own features to extract.

Masks are also missing and these will be re-introduced with statistics.

CHAPTER

FOUR

NEXT STEPS

4.1 Getting started

The best way to get started with resistics is to install the package and begin with the examples.

4.1.1 Installation

Resistics can be installed using pip. For most users, it is recommended to install both resistics and the resistics-readers package which provides support for several data formats.

```
python -m pip install resistics resistics-readers
```

For those who do not need the data support in resistics-readers, it is sufficient to install only resistics

```
python -m pip install resistics
```

4.1.2 Tutorials

Reading data

The main resistics package supports two time data formats and two calibration data formats.

For time data:

- ASCII (including compressed ASCII, e.g. bz2)
- numpy .npy

Where possible, it is recommended to use the numpy data format for time data as this is quicker to read from. Whilst it is a binary format, it is portable and well supported by the numpy package.

For calibration data, resistics supports:

- Text file calibration data
- JSON calibration data

The structure of these two calibration data formats can be seen in the relevant examples.

Note: Support for other data formats is provided by the resistics-readers package. This includes support for Metronix ATS data, SPAM RAW data, Phoenix TS data, Lemi data and potentially more in the future.

Time data ASCII

This example will show how to read time data from an ASCII file using the default ASCII data reader. To do this, a metadata file is required. The example shows how an appropriate metadata file can be created and the information required to create such a metadata file.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

The dataset is KAP175. A couple of notes:

- The data has a sample every 5 seconds, meaning a 0.2 Hz sampling frequency.
- Values of 1E32 have been replaced by NaN

```
from pathlib import Path
import pandas as pd
from resistics.time import ChanMetadata, TimeMetadata, TimeReaderAscii, InterpolateNans
import plotly
```

Define the data path. This is dependent on where the data is stored. Here, the data path is being read from an environment variable.

```
time_data_path = Path("..", "..", "data", "time", "ascii")
ascii_data_path = time_data_path / "kap175as.ts"
```

The folder contains a single ascii data file. Let's have a look at the contents of the file.

```
with ascii_data_path.open("r") as f:
    for line_number, line in enumerate(f):
        print(line.strip("\n"))
        if line_number >= 130:
            break
```

```
# time series file from tssplice
# date: Mon Nov  7 05:44:13 2016
#
# Files spliced together:
# kap175a1  2003-10-31 11:00:00-2003-11-06 15:17:39
# kap175b1  2003-11-06 16:00:00-2003-11-15 09:56:39
#
# Following comment block from first file...
#
# time series file from mp2ts
# date: Mon Nov  7 05:44:07 2016
#
# input file: kap175\kap175a1.1mp
#
# Machine endian: Little
# UNIX set      : F
#
# site description: maroi
#
# Latitude   :022:11:30 S
# Longitude  :029:51:31 E
```

(continues on next page)

(continued from previous page)

```
#
# LiMS acquisition code : 10.2
# LiMS box     number   :   53
# Magnetometer number   :   53
#
# Ex line length (m):     100.00
# Ey line length (m):      94.00
#
# Azimuths relative to: MAGNETIC NORTH
# Ex azimuth;  30
# Ey azimuth; 120
# Hx azimuth;  30
# Hy azimuth; 120
#
# FIRST 20 POINTS DROPPED FROM .1mp FILE TO
# ACCOUNT FOR FILTER SETTLING
#
#F Filter block begin
#F
#F Filters applied to LiMS/LRMT data are:
#F 1: Analogue anti-alias six-pole Bessel low-pass
#F    filters on each channel with -3 dB point at nominally 5 Hz.
#F    -calibrated values given below
#F
#F 2: Digital anti-alias multi-stage Chebyshev FIR filters
#F    with final stage at 2xsampling rate
#F
#F 1: Analogue single-pole Butterworth high-pass filters on the
#F    telluric channels only with -3 dB point at nominally 30,000 s
#F    -calibrated values given below
#F
#F Chan     Calib    Low-pass   High-pass (s)
#F  1        1.00        0.00        0.00
#F  2        1.00        0.00        0.00
#F  3        1.00        0.00        0.00
#F  4        1.00        0.00        0.00
#F  5        1.00        0.00        0.00
#F
#F In the tsrestack code, these filter responses are
#F removed using bessel7.f and high17.f
#F
#F Filter block end
>INFO_START:
>STATION   :kap175
>INSTRUMENT: 53
>WINDOW    :kap175as
>LATITUDE  : -22.1916695
>LONGITUDE :  29.8586102
>ELEVATION :  0.
>UTM_ORIGIN:  27.
>UTM_NORTH : -2456678
>UTM_EAST  : 794763
```

(continues on next page)

(continued from previous page)

```
>COORD_SYS :MAGNETIC NORTH
>DECLIN    :  0.
>FORM      :ASCII
>FORMAT    :FREE
>SEQ_REC   : 1
>NCHAN     : 5
>CHAN_1    :HX
>SENSOR_1  : 53
>AZIM_1    :  30.
>UNITS_1   :nT
>GAIN_1    :  1.
>BASELINE_1:  12618.2402
>CHAN_2    :HY
>SENSOR_2  : 53
>AZIM_2    :  120.
>UNITS_2   :nT
>GAIN_2    :  1.
>BASELINE_2: -7354.87988
>CHAN_3    :HZ
>SENSOR_3  : 53
>AZIM_3    :  0.
>UNITS_3   :nT
>GAIN_3    :  1.
>BASELINE_3: -26291.1992
>CHAN_4    :EX
>SENSOR_4  : 53
>AZIM_4    :  30.
>UNITS_4   :mV/km
>GAIN_4    :  1.
>CHAN_5    :EY
>SENSOR_5  : 53
>AZIM_5    :  120.
>UNITS_5   :mV/km
>GAIN_5    :  1.
>STARTTIME :2003-10-31 11:00:00
>ENDTIME   :2003-11-15 09:56:39
>T_UNITS   :s
>DELTA_T   :  5.
>MIS_DATA  :  1.00000003E+32
>INFO_END  :
  2.39398551  1.43499565  2.21125007 -1.55760086  0.0748437345
  2.23659754  1.09759927  2.16549993 -6.5316  1.9800632
  1.6032145  0.608650982  2.02824998 -14.0248184  3.94819808
  0.724482358 -0.00434030406  1.79949999 -22.1152382  4.54121494
 -0.170995399 -0.679827273  1.54025006 -28.7814693  4.58896542
 -1.17621446 -1.34823668  1.28100002 -33.5379982  5.47701597
 -2.31609321 -1.93590927  0.991250038 -37.1378212  6.52709579
 -3.41281223 -2.44583607  0.701499999 -38.6588211  6.6605401
 -4.29263926 -2.81293082  0.442250013 -37.8415413  6.49546909
 -4.97082424 -3.01078033  0.244000003 -35.6481323  6.60037565
 -5.44532394 -3.07342243  0.106749997 -31.5662174  6.48429155
 -5.67856073 -2.94394422  0.0305000003 -26.1866817  6.25566292
```

(continues on next page)

(continued from previous page)

```
 -5.76094007 -2.70975876  0.0152500002 -22.297369  6.53417683
 -5.86769009 -2.52486253  0.0152500002 -20.8914051  7.27097702
 -6.06688833 -2.39334106  0. -20.4016094  7.70362616
 -6.25846148 -2.27502656 -0.0305000003 -20.194458  7.71082592
 -6.485569 -2.24766445 -0.0305000003 -22.052597  7.81821918
 -6.84828472 -2.35142326 -0.0457499996 -26.1871376  8.14745235
```

Note that the metadata requires the number of samples. Pandas can be useful for this purpose.

```
df = pd.read_csv(ascii_data_path, header=None, skiprows=121, delim_whitespace=True)
n_samples = len(df.index)
print(df)
```

```
                0           1           2          3         4
0       -4.292639   -2.812931    0.442250 -37.841541  6.495469
1       -4.970824   -3.010780    0.244000 -35.648132  6.600376
2       -5.445324   -3.073422    0.106750 -31.566217  6.484292
3       -5.678561   -2.943944    0.030500 -26.186682  6.255663
4       -5.760940   -2.709759    0.015250 -22.297369  6.534177
...           ...         ...         ...        ...       ...
258428 -162.422211 -738.918762 -504.817841  10.141210 -3.474090
258429 -162.422211 -738.918762 -504.817841   9.408063 -3.485243
258430 -162.422211 -738.918762 -504.817841   8.700190 -3.631670
258431 -162.422211 -738.918762 -504.817841   8.757909 -3.823143
258432 -162.422211 -738.918762 -504.817841   8.475470 -4.004943

[258433 rows x 5 columns]
```

Define other key pieces of recording information

```
fs = 0.2
chans = ["Hx", "Hy", "Hz", "Ex", "Ey"]
first_time = pd.Timestamp("2003-10-31 11:00:00")
last_time = first_time + (n_samples - 1) * pd.Timedelta(1 / fs, "s")
```

The next step is to create a TimeMetadata object. The TimeMetdata has information about the recording and channels. Let's construct the TimeMetadata and save it as a JSON along with the time series data file.

```
chans_metadata = {}
for chan in chans:
    chan_type = "electric" if chan in ["Ex", "Ey"] else "magnetic"
    chans_metadata[chan] = ChanMetadata(
        name=chan, chan_type=chan_type, data_files=[ascii_data_path.name]
    )
time_metadata = TimeMetadata(
    fs=fs,
    chans=chans,
    n_samples=n_samples,
    first_time=first_time,
    last_time=last_time,
    chans_metadata=chans_metadata,
)
```

(continues on next page)

(continued from previous page)

```
time_metadata.summary()
time_metadata.write(time_data_path / "metadata.json")
```


```
{
    'file_info': None,
    'fs': 0.2,
    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
    'n_chans': 5,
    'n_samples': 258433,
    'first_time': '2003-10-31 11:00:00.000000_000000_000000_000000',
    'last_time': '2003-11-15 09:56:00.000000_000000_000000_000000',
    'system': '',
    'serial': '',
    'wgs84_latitude': -999.0,
    'wgs84_longitude': -999.0,
    'easting': -999.0,
    'northing': -999.0,
    'elevation': -999.0,
    'chans_metadata': {
        'Hx': {
            'name': 'Hx',
            'data_files': ['kap175as.ts'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hy': {
            'name': 'Hy',
            'data_files': ['kap175as.ts'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hz': {
            'name': 'Hz',
```


(continues on next page)

(continued from previous page)

```
            'data_files': ['kap175as.ts'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ex': {
            'name': 'Ex',
            'data_files': ['kap175as.ts'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ey': {
            'name': 'Ey',
            'data_files': ['kap175as.ts'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        }
    },
    'history': {'records': []}
}
```

Now the data is ready to be read in by resistics. Read it in and print the first and last sample values.

```
reader = TimeReaderAscii(extension=".ts", n_header=121)
time_data = reader.run(time_data_path)
print(time_data.data[:, 0])
```

(continues on next page)

(continued from previous page)

```
print(time_data.data[:, -1])
```

```
[ -4.2926393  -2.8129308   0.44225   -37.84154     6.495469 ]
[-162.42221   -738.91876   -504.81784      8.47547     -4.0049434]
```

There are some invalid values in the data that have been replaced by NaN values. Interpolate the NaN values.

```
time_data = InterpolateNans().run(time_data)
```

Finally plot the data. By default, the data is downsampled using the LTTB algorithm to avoid slow and large plots.

```
fig = time_data.plot(max_pts=1_000)
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 8.907 seconds)

Time data bz2

This example will show how to read time data from a compressed ASCII file using the default ASCII data reader. In this case, the data has been compressed using bz2.

To read such a compressed ASCII data file, a metadata file is required. The example shows how an appropriate metadata file can be created and the information required to create such a metadata file.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

The dataset is KAP148. A couple of notes:

- The data has a sample every 5 seconds, meaning a 0.2 Hz sampling frequency.
- Values of 1E32 have been replaced by NaN

```
from pathlib import Path
import bz2
import pandas as pd
from resistics.time import ChanMetadata, TimeMetadata, TimeReaderAscii, InterpolateNans
import plotly
```

Define the data path. This is dependent on where the data is stored. Here, the data path is being read from an environment variable.

```
time_data_path = Path("..", "..", "data", "time", "bz2")
ascii_data_path = time_data_path / "kap148as.ts.bz2"
```

The folder contains a single ascii data file. Let's have a look at the contents of the file.

```
with bz2.open(ascii_data_path, "rt") as f:
    for line_number, line in enumerate(f):
        print(line.strip("\n"))
        if line_number >= 130:
            break
```

```
# time series file from tssplice
# date: Mon Nov  7 05:27:46 2016
#
# Files spliced together:
# kap148a1  2003-10-25 11:30:00-2003-11-02 10:52:04
# kap148b1  2003-11-02 11:30:00-2003-11-12 11:15:34
# kap148c1  2003-11-12 11:45:00-2003-11-21 13:43:14
# kap148d1  2003-11-21 14:30:00-2003-11-29 10:14:10
#
# Following comment block from first file...
#
# time series file from mp2ts
# date: Mon Nov  7 05:27:32 2016
#
# input file: kap148\kap148a1.1mp
#
# Machine endian: Little
# UNIX set      : F
#
# site description: suikerbosrand
#
# Latitude   :025:55:40 S
# Longitude  :026:27:04 E
#
# LiMS acquisition code : 10.2
# LiMS box     number   :   26
# Magnetometer number   :   26
#
# Ex line length (m):     100.00
# Ey line length (m):      98.00
#
# Azimuths relative to: MAGNETIC NORTH
# Ex azimuth;   0
# Ey azimuth;  90
# Hx azimuth;   0
# Hy azimuth;  90
#
# FIRST 20 POINTS DROPPED FROM .1mp FILE TO
# ACCOUNT FOR FILTER SETTLING
#
#F Filter block begin
#F
#F Filters applied to LiMS/LRMT data are:
#F 1: Analogue anti-alias six-pole Bessel low-pass
#F    filters on each channel with -3 dB point at nominally 5 Hz.
#F    -calibrated values given below
#F
#F 2: Digital anti-alias multi-stage Chebyshev FIR filters
#F    with final stage at 2xsampling rate
#F
#F 1: Analogue single-pole Butterworth high-pass filters on the
#F    telluric channels only with -3 dB point at nominally 30,000 s
```

(continues on next page)

(continued from previous page)

```
#F    -calibrated values given below
#F
#F Chan     Calib    Low-pass   High-pass (s)
#F  1        1.00        0.00        0.00
#F  2        1.00        0.00        0.00
#F  3        1.00        0.00        0.00
#F  4        1.00        0.00        0.00
#F  5        1.00        0.00        0.00
#F
#F In the tsrestack code, these filter responses are
#F removed using bessel7.f and high17.f
#F
#F Filter block end
>INFO_START:
>STATION   :kap148
>INSTRUMENT: 26
>WINDOW    :kap148as
>LATITUDE  : -25.9277802
>LONGITUDE :  26.4511108
>ELEVATION :  1518.
>UTM_ORIGIN:  27.
>UTM_NORTH : -2867639
>UTM_EAST  : 445033
>COORD_SYS :MAGNETIC NORTH

>DECLIN    :  0.
>FORM      :ASCII
>FORMAT    :FREE

>SEQ_REC   : 1
>NCHAN     : 5
>CHAN_1    :HX
>SENSOR_1  : 26
>AZIM_1    :  0.
>UNITS_1   :nT

>GAIN_1    :  1.
>BASELINE_1:  12410.8799
>CHAN_2    :HY
>SENSOR_2  : 26
>AZIM_2    :  90.
>UNITS_2   :nT

>GAIN_2    :  1.
>BASELINE_2: -245.759995
>CHAN_3    :HZ
>SENSOR_3  : 26
>AZIM_3    :  0.
>UNITS_3   :nT

>GAIN_3    :  1.
>BASELINE_3: -25784.3203
```

(continues on next page)

(continued from previous page)

```
>CHAN_4    :EX
>SENSOR_4  : 26
>AZIM_4    :  0.
>UNITS_4   :mV/km

>GAIN_4    :  1.
>CHAN_5    :EY
>SENSOR_5  : 26
>AZIM_5    :  90.
>UNITS_5   :mV/km

>GAIN_5    :  1.
>STARTTIME :2003-10-25 11:30:00
>ENDTIME   :2003-11-29 10:14:10
>T_UNITS   :s

>DELTA_T   :  5.
>MIS_DATA  :  1.00000003E+32
>INFO_END  :
  3.00425005  2.62300014 -0.381249994  2.5315001  2.44311237
  3.01950002  2.60774994 -0.457500011  2.62300014  2.34974504
  3.01950002  2.62300014 -0.488000005  2.51625013  2.33418369
  3.03474998  2.65350008 -0.442250013  2.45525002  2.48979592
  3.06524992  2.66875005 -0.427000016  2.50099993  2.58316326
  3.04999995  2.68400002 -0.488000005  2.54675007  2.55204082
  3.09575009  2.69924998 -0.564249992  2.63825011  2.38086748
  3.21775007  2.71449995 -0.610000014  2.60774994  2.28750014
```

Note that the metadata requires the number of samples. Pandas can be useful for this purpose.

```
df = pd.read_csv(ascii_data_path, header=None, skiprows=123, delim_whitespace=True)
n_samples = len(df.index)
print(df)
```

```
                0           1           2           3           4
0        3.004250    2.623000  -0.381250   2.531500    2.443112
1        3.019500    2.607750  -0.457500   2.623000    2.349745
2        3.019500    2.623000  -0.488000   2.516250    2.334184
3        3.034750    2.653500  -0.442250   2.455250    2.489796
4        3.065250    2.668750  -0.427000   2.501000    2.583163
...           ...         ...         ...         ...         ...
603885  67.700172 -132.892487  33.821594 -33.046749  178.160461
603886  68.035675 -132.724747  33.836845 -32.955250  398.631897
603887  67.791672 -133.105988  34.294342 -32.665501  509.894653
603888  66.952919 -134.768250  34.919594 -32.345249  508.774261
603889  66.571670 -135.332489  35.102593 -32.452000  503.001038

[603890 rows x 5 columns]
```

Define other key pieces of recording information

```
fs = 0.2
chans = ["Hx", "Hy", "Hz", "Ex", "Ey"]
first_time = pd.Timestamp("2003-10-25 11:30:00")
last_time = first_time + (n_samples - 1) * pd.Timedelta(1 / fs, "s")
```

The next step is to create a TimeMetadata object. The TimeMetdata has information about the recording and channels. Let's construct the TimeMetadata and save it as a JSON along with the time series data file.

```
chans_metadata = {}
for chan in chans:
    chan_type = "electric" if chan in ["Ex", "Ey"] else "magnetic"
    chans_metadata[chan] = ChanMetadata(
        name=chan, chan_type=chan_type, data_files=[ascii_data_path.name]
    )
time_metadata = TimeMetadata(
    fs=fs,
    chans=chans,
    n_samples=n_samples,
    first_time=first_time,
    last_time=last_time,
    chans_metadata=chans_metadata,
)
time_metadata.summary()
time_metadata.write(time_data_path / "metadata.json")
```

```
{
    'file_info': None,
    'fs': 0.2,
    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
    'n_chans': 5,
    'n_samples': 603890,
    'first_time': '2003-10-25 11:30:00.000000_000000_000000_000000',
    'last_time': '2003-11-29 10:14:05.000000_000000_000000_000000',
    'system': '',
    'serial': '',
    'wgs84_latitude': -999.0,
    'wgs84_longitude': -999.0,
    'easting': -999.0,
    'northing': -999.0,
    'elevation': -999.0,
    'chans_metadata': {
        'Hx': {
            'name': 'Hx',
            'data_files': ['kap148as.ts.bz2'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
```

(continues on next page)

(continued from previous page)

```
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hy': {
            'name': 'Hy',
            'data_files': ['kap148as.ts.bz2'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hz': {
            'name': 'Hz',
            'data_files': ['kap148as.ts.bz2'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ex': {
            'name': 'Ex',
            'data_files': ['kap148as.ts.bz2'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ey': {
            'name': 'Ey',
            'data_files': ['kap148as.ts.bz2'],
```

(continues on next page)

(continued from previous page)

```
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        }
    },
    'history': {'records': []}
}
```

Now the data is ready to be read in by resistics. Read it in and print the first and last sample values.

```
reader = TimeReaderAscii(extension=".bz2", n_header=123)
time_data = reader.run(time_data_path)
print(time_data.data[:, 0])
print(time_data.data[:, -1])
```

```
[ 3.00425    2.6230001 -0.38125    2.5315     2.4431124]
[  66.57167  -135.33249    35.102592  -32.452     503.00104 ]
```

There are some invalid values in the data that have been replaced by NaN values. Interpolate the NaN values.

```
time_data = InterpolateNans().run(time_data)
```

Finally plot the data. By default, the data is downsampled using the LTTB algorithm to avoid slow and large plots.

```
fig = time_data.plot(max_pts=1_000)
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 5.200 seconds)

Time data binary

If a data file is available in npy binary format, this can be read in using the TimeReaderNumpy reader as long as a metadata file can be made.

Information about the recording will be required to make the metadata file. In the below example, a metadata file is made and then the data is read.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

The dataset is KAP130. A couple of notes:

- The data has a sample every 5 seconds, meaning a 0.2 Hz sampling frequency.
- Values of 1E32 have been replaced by NaN

```
from pathlib import Path
import numpy as np
import pandas as pd
from resistics.time import TimeMetadata, ChanMetadata, TimeReaderNumpy
from resistics.time import InterpolateNans, LowPass
import plotly
```

Define the data path. This is dependent on where the data is stored. Here, the data path is being read from an environment variable.

```
time_data_path = Path("..", "..", "data", "time", "binary")
binary_data_path = time_data_path / "kap130as.npy"
```

Define key pieces of recording information. This is known.

```
fs = 0.2
chans = ["Hx", "Hy", "Hz", "Ex", "Ey"]
first_time = pd.Timestamp("2003-10-17 15:30:00")
```

Note that the metadata requires the number of samples. This can be found by loading the data in memory mapped mode. In most cases, it is likely that this be known.

```
data = np.load(binary_data_path, mmap_mode="r")
n_samples = data.shape[1]
last_time = first_time + (n_samples - 1) * pd.Timedelta(1 / fs, "s")
```

The next step is to create a TimeMetadata object. The TimeMetdata has information about the recording and channels. Let's construct the TimeMetadata and save it as a JSON along with the time series data file.

```
chans_metadata = {}
for chan in chans:
    chan_type = "electric" if chan in ["Ex", "Ey"] else "magnetic"
    chans_metadata[chan] = ChanMetadata(
        name=chan, chan_type=chan_type, data_files=[binary_data_path.name]
    )
time_metadata = TimeMetadata(
    fs=fs,
    chans=chans,
    n_samples=n_samples,
    first_time=first_time,
    last_time=last_time,
    chans_metadata=chans_metadata,
)
time_metadata.summary()
time_metadata.write(time_data_path / "metadata.json")
```

```
{
    'file_info': None,
    'fs': 0.2,
    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
    'n_chans': 5,
    'n_samples': 707753,
    'first_time': '2003-10-17 15:30:00.000000_000000_000000_000000',
```

(continues on next page)

(continued from previous page)

```
    'last_time': '2003-11-27 14:29:20.000000_000000_000000_000000',
    'system': '',
    'serial': '',
    'wgs84_latitude': -999.0,
    'wgs84_longitude': -999.0,
    'easting': -999.0,
    'northing': -999.0,
    'elevation': -999.0,
    'chans_metadata': {
        'Hx': {
            'name': 'Hx',
            'data_files': ['kap130as.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hy': {
            'name': 'Hy',
            'data_files': ['kap130as.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hz': {
            'name': 'Hz',
            'data_files': ['kap130as.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
```

(continues on next page)

(continued from previous page)

```
            'instrument_calibration_file': ''
        },
        'Ex': {
            'name': 'Ex',
            'data_files': ['kap130as.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ey': {
            'name': 'Ey',
            'data_files': ['kap130as.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        }
    },
    'history': {'records': []}
}
```

Read the numpy formatted time data using the appropriate time data reader.

```
time_data = TimeReaderNumpy().run(time_data_path)
time_data.metadata.summary()
```

```
{
    'file_info': {
        'created_on_local': '2023-05-12T14:40:14.108330',
        'created_on_utc': '2023-05-12T14:40:14.108335',
        'version': '1.0.0a3'
    },
    'fs': 0.2,
    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
    'n_chans': 5,
    'n_samples': 707753,
    'first_time': '2003-10-17 15:30:00.000000_000000_000000_000000',
```

(continues on next page)

(continued from previous page)

```
    'last_time': '2003-11-27 14:29:20.000000_000000_000000_000000',
    'system': '',
    'serial': '',
    'wgs84_latitude': -999.0,
    'wgs84_longitude': -999.0,
    'easting': -999.0,
    'northing': -999.0,
    'elevation': -999.0,
    'chans_metadata': {
        'Hx': {
            'name': 'Hx',
            'data_files': ['kap130as.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hy': {
            'name': 'Hy',
            'data_files': ['kap130as.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hz': {
            'name': 'Hz',
            'data_files': ['kap130as.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
```

(continues on next page)

(continued from previous page)

```
            'instrument_calibration_file': ''
        },
        'Ex': {
            'name': 'Ex',
            'data_files': ['kap130as.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ey': {
            'name': 'Ey',
            'data_files': ['kap130as.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        }
    },
    'history': {
        'records': [
            {
                'time_local': '2023-05-12T14:40:14.113421',
                'time_utc': '2023-05-12T14:40:14.113419',
                'creator': {
                    'name': 'TimeReaderNumpy',
                    'apply_scalings': True,
                    'extension': '.npy'
                },
                'messages': [
                    'Reading raw data from ../../data/time/binary',
                    'Sampling frequency 0.2 Hz',
                    'From sample, time: 0, 2003-10-17 15:30:00',
                    'To sample, time: 707752, 2003-11-27 14:29:20'
                ],
                'record_type': 'process'
            }
        ]
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Next remove any NaN values and plot the data. By default, the data is downsampled using lttb so that it is possible to plot the full timeseries. A second plot will be added with the same data filtered with a (1/(24*3600)) Hz or 1 day period low pass filter.

```
time_data = InterpolateNans().run(time_data)
fig = time_data.plot(max_pts=1_000, legend="original")
filtered_data = LowPass(cutoff=1 / (24 * 3_600)).run(time_data)
fig = filtered_data.plot(
    max_pts=1_000, fig=fig, chans=chans, legend="filtered", color="red"
)
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 2.822 seconds)

Calibration data JSON

The preferred format for calibration data is JSON file. Howeer, they are not always as easy to handwrite, so it is possible to use txt/ASCII calibration files too.

```
from pathlib import Path
import json
from resistics.time import ChanMetadata
from resistics.calibrate import SensorCalibrationJSON
import plotly
```

Define the calibration data path. This is dependent on where the data is stored.

```
cal_data_path = Path("..", "..", "data", "calibration", "example.json")
```

Inspect the contents of the calibration file

```
with cal_data_path.open("r") as f:
    file_contents = json.load(f)
print(json.dumps(file_contents, indent=4, sort_keys=True))
```

```
{
    "file_info": {
        "created_on_local": "2021-07-04T17:20:47.042892",
        "created_on_utc": "2021-07-04T16:20:47.042892",
        "version": "1.0.0a3"
    },
    "file_path": "calibration_ascii\\example.txt",
    "frequency": [
        0.00011,
        0.0011,
        0.011,
        0.021,
```

(continues on next page)

(continued from previous page)

```
        0.03177,
        0.048062,
        0.072711,
        0.11,
        0.13249,
        0.15959,
        0.19222,
        0.23153,
        0.27888,
        0.3359,
        0.40459,
        0.48733,
        0.58698,
        0.70702,
        0.8516,
        1.0257,
        1.2355,
        1.4881,
        1.7925,
        2.159,
        2.6005,
        3.1323,
        3.7728,
        4.5443,
        5.4736,
        6.5929,
        7.9411,
        9.5649,
        11.521,
        13.877,
        16.715,
        20.132,
        24.249,
        29.208,
        35.181,
        42.375,
        51.041,
        61.478,
        74.05,
        89.192,
        107.43,
        129.4,
        155.86,
        187.73,
        226.12,
        272.36,
        328.06,
        395.14,
        475.95,
        573.28,
        690.5,
        831.71,
```

(continues on next page)

(continued from previous page)

```
        1001.8
    ],
    "magnitude": [
        0.01,
        0.1,
        1.0,
        1.903,
        2.903,
        4.339,
        6.565,
        9.935,
        12.02,
        14.2,
        17.24,
        20.82,
        24.53,
        29.38,
        34.21,
        40.3,
        47.34,
        53.8,
        61.1,
        68.05,
        75.0,
        80.45,
        85.4,
        89.25,
        92.2,
        94.4,
        96.2,
        97.3,
        98.1,
        98.65,
        99.05,
        99.35,
        99.75,
        99.75,
        99.8,
        99.95,
        99.95,
        99.95,
        100.0,
        100.0,
        100.2,
        100.0,
        100.0,
        100.0,
        99.8,
        99.8,
        99.7,
        99.6,
        99.3,
```

(continues on next page)

(continued from previous page)


```
        98.8,
        98.0,
        96.0,
        92.7,
        87.55,
        80.8,
        74.5,
        70.7
    ],
    "magnitude_unit": "mV/nT",
    "n_samples": 57,
    "phase": [
        1.5707963267948966,
        1.5707963267948966,
        1.5533430342749532,
        1.546065011294137,
        1.5428361521779475,
        1.526971109277319,
        1.505398839722669,
        1.4724295701524963,
        1.4491817845159316,
        1.4238046971919343,
        1.402913106045562,
        1.3795780539463978,
        1.3431181258722362,
        1.2771446801468507,
        1.2277693156079312,
        1.1700861838295185,
        1.0909355022515757,
        1.0088177609452424,
        0.9242216521010773,
        0.827041719350033,
        0.7294952674560699,
        0.6388428661074844,
        0.5498136209632537,
        0.46968555500419407,
        0.3926641751136843,
        0.32766811376941546,
        0.270246781378802,
        0.21961477977844648,
        0.17517869702267086,
        0.13625436404469332,
        0.10112263153129945,
        0.06969099703213358,
        0.04107981460419053,
        0.013366778609323771,
        -0.01522921945412692,
        -0.04212177616763115,
        -0.07138396640656808,
        -0.10302329508672128,
        -0.1383801750736224,
        -0.1780235837034216,
```


(continues on next page)

(continued from previous page)

```
        -0.21912608758788807,
        -0.2792875869041326,
        -0.3439869422755624,
        -0.42004839107747527,
        -0.5115036438819781,
        -0.6197664173831864,
        -0.7506312046977213,
        -0.9091245540713263,
        -1.1008838789879434,
        -1.3337282544965068,
        -1.616262153809349,
        -1.9645426060448175,
        -2.384294291149454,
        -2.8906143071530086,
        2.784672821556953,
        2.030865117620602,
        0.8973959414979245
    ],
    "sensor": "lemi120",
    "serial": 710,
    "static_gain": 1.0
}
```

Read the data using the appropriate calibration data reader. As calibration data can be dependent on certain sensor parameters, channel metadata needs to be passed to the method.

```
chan_metadata = ChanMetadata(name="Hx", chan_type="magnetic", data_files=[])
cal_data = SensorCalibrationJSON().read_calibration_data(cal_data_path, chan_metadata)
```

Plot the calibration data.

```
fig = cal_data.plot(color="maroon")
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 0.312 seconds)

Calibration data TXT

An alternative to JSON calibration files is to use text/ASCII calibration files.

```
from pathlib import Path
from resistics.time import ChanMetadata
from resistics.calibrate import SensorCalibrationTXT
import plotly
```

Define the calibration data path. This is dependent on where the data is stored.

```
cal_data_path = Path("..", "..", "data", "calibration", "example.txt")
```

Inspect the contents of the calibration file

```
with cal_data_path.open("r") as f:
    for line_number, line in enumerate(f):
        print(line.strip("\n"))
```

```
Serial = 710
Sensor = LEMI120
Static gain = 1
Magnitude unit = mV/nT
Phase unit = degrees
Chopper = False

CALIBRATION DATA
1.1000E-4       1.000E-2        9.0000E1
1.1000E-3       1.000E-1        9.0000E1
1.1000E-2       1.000E0 8.9000E1
2.1000E-2       1.903E0 8.8583E1
3.1770E-2       2.903E0 8.8398E1
4.8062E-2       4.339E0 8.7489E1
7.2711E-2       6.565E0 8.6253E1
1.1000E-1       9.935E0 8.4364E1
1.3249E-1       1.202E1 8.3032E1
1.5959E-1       1.420E1 8.1578E1
1.9222E-1       1.724E1 8.0381E1
2.3153E-1       2.082E1 7.9044E1
2.7888E-1       2.453E1 7.6955E1
3.3590E-1       2.938E1 7.3175E1
4.0459E-1       3.421E1 7.0346E1
4.8733E-1       4.030E1 6.7041E1
5.8698E-1       4.734E1 6.2506E1
7.0702E-1       5.380E1 5.7801E1
8.5160E-1       6.110E1 5.2954E1
1.0257E0        6.805E1 4.7386E1
1.2355E0        7.500E1 4.1797E1
1.4881E0        8.045E1 3.6603E1
1.7925E0        8.540E1 3.1502E1
2.1590E0        8.925E1 2.6911E1
2.6005E0        9.220E1 2.2498E1
3.1323E0        9.440E1 1.8774E1
3.7728E0        9.620E1 1.5484E1
4.5443E0        9.730E1 1.2583E1
5.4736E0        9.810E1 1.0037E1
6.5929E0        9.865E1 7.8068E0
7.9411E0        9.905E1 5.7939E0
9.5649E0        9.935E1 3.9930E0
1.1521E1        9.975E1 2.3537E0
1.3877E1        9.975E1 7.6586E-1
1.6715E1        9.980E1 -8.7257E-1
2.0132E1        9.995E1 -2.4134E0
2.4249E1        9.995E1 -4.0900E0
2.9208E1        9.995E1 -5.9028E0
3.5181E1        1.000E2 -7.9286E0
4.2375E1        1.000E2 -1.0200E1
```

(continues on next page)

(continued from previous page)

```
5.1041E1        1.002E2 -1.2555E1
6.1478E1        1.000E2 -1.6002E1
7.4050E1        1.000E2 -1.9709E1
8.9192E1        1.000E2 -2.4067E1
1.0743E2        9.980E1 -2.9307E1
1.2940E2        9.980E1 -3.5510E1
1.5586E2        9.970E1 -4.3008E1
1.8773E2        9.960E1 -5.2089E1
2.2612E2        9.930E1 -6.3076E1
2.7236E2        9.880E1 -7.6417E1
3.2806E2        9.800E1 -9.2605E1
3.9514E2        9.600E1 -1.1256E2
4.7595E2        9.270E1 -1.3661E2
5.7328E2        8.755E1 -1.6562E2
6.9050E2        8.080E1 1.5955E2
8.3171E2        7.450E1 1.1636E2
1.0018E3        7.070E1 5.1417E1
```

Read the data using the appropriate calibration data reader. As calibration data can be dependent on certain sensor parameters, channel metadata needs to be passed to the method.

```
chan_metadata = ChanMetadata(name="Hx", chan_type="magnetic", data_files=[])
cal_data = SensorCalibrationTXT().read_calibration_data(cal_data_path, chan_metadata)
```

Plot the calibration data.

```
fig = cal_data.plot(color="green")
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 0.282 seconds)

Quick functions

When doing fieldwork, it's often useful to quickly assess data before dismantling a site setup. The quick functions are there to provide fast viewing and processing of data without having to set many parameters.

Reading time data

Resistics can quickly read a single continuous recording using the quick reading functionality. This can be useful for inspecting the metadata and having a look at the data when in the field.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
import resistics.letsgo as letsgo
import plotly
```

Define the data path. This is dependent on where the data is stored.

```
time_data_path = Path("..", "..", "data", "time", "quick", "kap123")
sd.seedir(str(time_data_path), style="emoji")
```

```
kap123/
├─ metadata.json
└─ data.npy
```

Quickly read the time series data and inspect the metadata

```
time_data = letsgo.quick_read(time_data_path)
time_data.metadata.summary()
```

```
{
    'file_info': {
        'created_on_local': '2021-07-07T22:25:45.320529',
        'created_on_utc': '2021-07-07T21:25:45.320529',
        'version': '1.0.0a0'
    },
    'fs': 0.2,
    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
    'n_chans': 5,
    'n_samples': 361512,
    'first_time': '2003-11-10 15:00:00.000000_000000_000000_000000',
    'last_time': '2003-12-01 13:05:55.000000_000000_000000_000000',
    'system': '',
    'serial': '',
    'wgs84_latitude': -999.0,
    'wgs84_longitude': -999.0,
    'easting': -999.0,
    'northing': -999.0,
    'elevation': -999.0,
    'chans_metadata': {
        'Hx': {
            'name': 'Hx',
            'data_files': ['data.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hy': {
            'name': 'Hy',
            'data_files': ['data.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
```

(continues on next page)

(continued from previous page)

```
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hz': {
            'name': 'Hz',
            'data_files': ['data.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ex': {
            'name': 'Ex',
            'data_files': ['data.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ey': {
            'name': 'Ey',
            'data_files': ['data.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
```

(continues on next page)

(continued from previous page)

```
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        }
    },
    'history': {
        'records': [
            {
                'time_local': '2023-05-12T14:40:18.311903',
                'time_utc': '2023-05-12T14:40:18.311902',
                'creator': {
                    'name': 'TimeReaderNumpy',
                    'apply_scalings': True,
                    'extension': '.npy'
                },
                'messages': [
                    'Reading raw data from ../../data/time/quick/kap123',
                    'Sampling frequency 0.2 Hz',
                    'From sample, time: 0, 2003-11-10 15:00:00',
                    'To sample, time: 361511, 2003-12-01 13:05:55'
                ],
                'record_type': 'process'
            }
        ]
    }
}
```

Take a subsection of the data and inspect the metadata for the subsection

```
time_data_sub = time_data.subsection("2003-11-20 12:00:00", "2003-11-21 00:00:00")
time_data_sub.metadata.summary()
```

```
{
    'file_info': {
        'created_on_local': '2021-07-07T22:25:45.320529',
        'created_on_utc': '2021-07-07T21:25:45.320529',
        'version': '1.0.0a0'
    },
    'fs': 0.2,
    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
    'n_chans': 5,
    'n_samples': 8641,
    'first_time': '2003-11-20 12:00:00.000000_000000_000000_000000',
    'last_time': '2003-11-21 00:00:00.000000_000000_000000_000000',
    'system': '',
    'serial': '',
    'wgs84_latitude': -999.0,
    'wgs84_longitude': -999.0,
    'easting': -999.0,
    'northing': -999.0,
    'elevation': -999.0,
    'chans_metadata': {
        'Hx': {
```

(continues on next page)

(continued from previous page)

```
            'name': 'Hx',
            'data_files': ['data.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hy': {
            'name': 'Hy',
            'data_files': ['data.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hz': {
            'name': 'Hz',
            'data_files': ['data.npy'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ex': {
            'name': 'Ex',
            'data_files': ['data.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
```

(continues on next page)

(continued from previous page)

```
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Ey': {
            'name': 'Ey',
            'data_files': ['data.npy'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
            'gain1': 1.0,
            'gain2': 1.0,
            'scaling': 1.0,
            'chopper': False,
            'dipole_dist': 1.0,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        }
    },
    'history': {
        'records': [
            {
                'time_local': '2023-05-12T14:40:18.311903',
                'time_utc': '2023-05-12T14:40:18.311902',
                'creator': {
                    'name': 'TimeReaderNumpy',
                    'apply_scalings': True,
                    'extension': '.npy'
                },
                'messages': [
                    'Reading raw data from ../../data/time/quick/kap123',
                    'Sampling frequency 0.2 Hz',
                    'From sample, time: 0, 2003-11-10 15:00:00',
                    'To sample, time: 361511, 2003-12-01 13:05:55'
                ],
                'record_type': 'process'
            },
            {
                'time_local': '2023-05-12T14:40:18.334664',
                'time_utc': '2023-05-12T14:40:18.334662',
                'creator': {
                    'name': 'Subsection',
                    'from_time': '2003-11-20 12:00:00',
                    'to_time': '2003-11-21 00:00:00'
                },
                'messages': [
                    'Subsection from sample 170640 to 179280',
                    'First time: 2003-11-10 15:00:00 -> 2003-11-20 12:00:00',
```

(continues on next page)

(continued from previous page)

```
                    'Last time: 2003-12-01 13:05:55 -> 2003-11-21 00:00:00'
                ],
                'record_type': 'process'
            }
        ]
    }
}
```

Plot the full time data with LTTB downsampling and a subsection without any downsampling. Comparing the downsampled and original data, there is clearly some loss but the LTTB downsampled data does a reasonable job capaturing the main features whilst showing a greater amount of data.

```
fig = time_data.plot(max_pts=1_000)
fig = time_data_sub.plot(
    fig, chans=time_data.metadata.chans, color="red", legend="Subsection", max_pts=None
)
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 7.279 seconds)

Viewing time data

With the quick viewing functionality, it is possible to view time series data without having to setup a project or explicitly read the data first. The quickview decimation option provides an easy way to see the time series at multiple sampling frequencies (decimated to lower frequencies).

> **Warning:** The time series data is downsampled for viewing using the LTTB algorithm, which tries to capture the features of the time series using a given number of data points. Setting max_pts to None will try and plot all points which can cause serious performance issues for large datasets.
>
> Those looking to view non downsampled data are advised to use the quick reading functionality and then plot specific subsections of data.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
import resistics.letsgo as letsgo
import plotly
```

Define the data path. This is dependent on where the data is stored.

```
time_data_path = Path("..", "..", "data", "time", "quick", "kap123")
sd.seedir(str(time_data_path), style="emoji")
```

```
kap123/
├─ metadata.json
└─ data.npy
```

Quickly view the time series data

```
fig = letsgo.quick_view(time_data_path, max_pts=1_000)
fig.update_layout(height=700)
plotly.io.show(fig)
```

In many cases, data plotting at its recording frequency can be quite nosiy. The quickview function has the option to plot multiple decimation levels so the data can be compared at multiple sampling frequencies.

```
fig = letsgo.quick_view(time_data_path, max_pts=1_000, decimate=True)
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 5.801 seconds)

Getting spectra data

It can often be useful to have a look at the spectral content of time data. The quick functions make it easy to get the spectra data of a single time series recording.

Note that spectra data are calculated after decimation and spectra data objects include data for multiple decimation levels.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
import resistics.letsgo as letsgo
import plotly
```

Define the data path. This is dependent on where the data is stored.

```
time_data_path = Path("..", "..", "data", "time", "quick", "kap123")
sd.seedir(str(time_data_path), style="emoji")
```

```
kap123/
├─ metadata.json
└─ data.npy
```

Get the spectra data.

```
spec_data = letsgo.quick_spectra(time_data_path)
```

Once the spectra data has been calculated, it can be plotted in a variety of ways. The default plotting function plots the spectral data for multiple decimation levels.

```
fig = spec_data.plot()
fig.update_layout(height=900)
plotly.io.show(fig)
```

It is also possible to plot spectra data for a particular decimation level. In the below example, an optional grouping is being used to stack spectra data for the decimation level into certain time groups

```
fig = spec_data.plot_level_stack(level=0, grouping="3D")
fig.update_layout(height=900)
plotly.io.show(fig)
```

It is also possible to plot spectra heatmaps for a decimation level. Here, the sphinx_gallery_defer_figures

```
fig = spec_data.plot_level_section(level=0, grouping="6H")
fig.update_layout(height=900)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 4.145 seconds)

Transfer functions

When doing field work, it can be useful to quickly estimate the transfer function from a single continuous recording. This example shows estimation of the transfer function using all default settings. The default transfer function is the impedance tensor and this will be calculated. Later, the data will be re-processed using an alternative configuration.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
import resistics.letsgo as letsgo
import plotly
```

Define the data path. This is dependent on where the data is stored.

```
time_data_path = Path("..", "..", "data", "time", "quick", "kap123")
sd.seedir(str(time_data_path), style="emoji")
```

```
kap123/
├─ metadata.json
└─ data.npy
```

Now calculate the transfer function, in this case the impedance tensor

```
soln = letsgo.quick_tf(time_data_path)
fig = soln.tf.plot(
    soln.freqs,
    soln.components,
    to_plot=["ExHy", "EyHx"],
    x_lim=[1, 5],
    res_lim=[0, 4],
    legend="128",
    symbol="circle",
)
fig.update_layout(height=900)
plotly.io.show(fig)
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
 10%|#         | 2/20 [00:00<00:01, 17.17it/s]
```

(continues on next page)

(continued from previous page)

```
 25%|##5       | 5/20 [00:00<00:00, 23.36it/s]
100%|##########| 20/20 [00:00<00:00, 72.20it/s]

  0%|          | 0/20 [00:00<?, ?it/s]
  5%|5         | 1/20 [00:01<00:23,  1.24s/it]
 10%|#         | 2/20 [00:02<00:22,  1.24s/it]
 15%|#5        | 3/20 [00:03<00:21,  1.24s/it]
 20%|##        | 4/20 [00:04<00:19,  1.24s/it]
 25%|##5       | 5/20 [00:06<00:18,  1.24s/it]
 30%|###       | 6/20 [00:06<00:13,  1.06it/s]
 35%|###5      | 7/20 [00:06<00:09,  1.34it/s]
 40%|####      | 8/20 [00:07<00:07,  1.62it/s]
 45%|####5     | 9/20 [00:07<00:05,  1.89it/s]
 50%|#####     | 10/20 [00:07<00:04,  2.13it/s]
 55%|#####5    | 11/20 [00:08<00:03,  2.70it/s]
 60%|######    | 12/20 [00:08<00:02,  3.32it/s]
 65%|######5   | 13/20 [00:08<00:01,  3.94it/s]
 70%|#######   | 14/20 [00:08<00:01,  4.54it/s]
 75%|#######5  | 15/20 [00:08<00:00,  5.05it/s]
 80%|########  | 16/20 [00:08<00:00,  5.72it/s]
 85%|########5 | 17/20 [00:08<00:00,  6.31it/s]
 90%|######### | 18/20 [00:09<00:00,  6.80it/s]
 95%|#########5| 19/20 [00:09<00:00,  7.22it/s]
100%|##########| 20/20 [00:09<00:00,  7.53it/s]
100%|##########| 20/20 [00:09<00:00,  2.16it/s]
```

Total running time of the script: (0 minutes 10.362 seconds)

Using projects

Projects in resistics are the best way to deal with multiple recordings and sites. They enable the following functionality:

- Multiple recordings at the same sampling frequency can be used to calculate transfer functions
- Processing which combines data from different sites and requires alignment of windows
- Calculation of statistics and deeper analysis of recordings
- Use of multiple configurations (useful for experimentation or mixed instrument surveys)

The use of projects is highly recommended when dealing with more than one or two sites.

Making a project

The quick reading functionality of resistics focuses on analysis of single continuous recordings. When there are multiple recordings at a site or multiple sites, it can be more convenient to use a resistics project. This is generally easier to manage and use, especially when doing remote reference or intersite processing.

The data in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the data can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
```

(continues on next page)

(continued from previous page)

```
import shutil
import resistics.letsgo as letsgo
import plotly
```

Define the path where the project will be created and any extra project metadata. The only required piece of metadata is the reference time but there are other optional fields.

```
project_path = Path("..", "..", "data", "project", "kap03")
project_info = {
    "ref_time": "2003-10-15 00:00:00",
    "year": 2003,
    "country": "South Africa",
}
```

Create the new project and look at the directory structure. There are no data files in the project yet, so there is not much to see.

```
letsgo.new(project_path, project_info)
sd.seedir(str(project_path), style="emoji")
```

```
 kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
├─ calibrate/
├─ evals/
├─ features/
└─ time/
```

Load the project and have a look. When loading a project, a resistics environment is returned. This is a combination of a resistics project and a configuration.

```
resenv = letsgo.load(project_path)
resenv.proj.summary()
```

```
{
    'dir_path': '../../data/project/kap03',
    'begin_time': '2023-05-12 14:40:46.102000_000000_000000_000000',
    'end_time': '2023-05-12 14:40:46.102002_000000_000000_000000',
    'metadata': {
        'file_info': {
            'created_on_local': '2023-05-12T14:40:46.099101',
            'created_on_utc': '2023-05-12T14:40:46.099105',
            'version': '1.0.0a3'
        },
        'ref_time': '2003-10-15 00:00:00.000000_000000_000000_000000',
        'location': '',
        'country': 'South Africa',
        'year': 2003,
        'description': '',
```

(continues on next page)

(continued from previous page)

```
        'contributors': []
    },
    'sites': {}
}
```

Now let's copy some time series data into the project and look at the directory structure. Copy the data does not have to be done using Python and users can simply copy and paste the time series data into the time folder

```
copy_from = Path("..", "..", "data", "time", "kap03")
for site in copy_from.glob("*"):
    shutil.copytree(site, project_path / "time" / site.stem)
sd.seedir(str(project_path), style="emoji")
```

```
kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
├─ calibrate/
├─ evals/
├─ features/
└─ time/
  ├─ kap172/
  │  └─ meas01/
  │     ├─ metadata.json
  │     └─ data.npy
  ├─ kap163/
  │  └─ meas01/
  │     ├─ metadata.json
  │     └─ data.npy
  └─ kap160/
     └─ meas01/
        ├─ metadata.json
        └─ data.npy
```

Reload the project and print a new summary.

```
resenv = letsgo.reload(resenv)
resenv.proj.summary()
```

```
{
    'dir_path': '../../data/project/kap03',
    'begin_time': '2003-10-28 10:00:00.000000_000000_000000_000000',
    'end_time': '2003-11-24 15:31:55.000000_000000_000000_000000',
    'metadata': {
        'file_info': {
            'created_on_local': '2023-05-12T14:40:46.099101',
            'created_on_utc': '2023-05-12T14:40:46.099105',
            'version': '1.0.0a3'
        },
```

(continues on next page)

(continued from previous page)

```
        'ref_time': '2003-10-15 00:00:00.000000_000000_000000_000000',
        'location': '',
        'country': 'South Africa',
        'year': 2003,
        'description': '',
        'contributors': []
    },
    'sites': {
        'kap172': {
            'dir_path': '../../data/project/kap03/time/kap172',
            'measurements': {
                'meas01': {
                    'site_name': 'kap172',
                    'dir_path': '../../data/project/kap03/time/kap172/meas01',
                    'metadata': {
                        'file_info': {
                            'created_on_local': '2021-07-07T22:27:00.395145',
                            'created_on_utc': '2021-07-07T21:27:00.395145',
                            'version': '1.0.0a0'
                        },
                        'fs': 0.2,
                        'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
                        'n_chans': 5,
                        'n_samples': 414498,
                        'first_time': '2003-10-30 13:00:00.000000_000000_000000_000000',
                        'last_time': '2003-11-23 12:41:25.000000_000000_000000_000000',
                        'system': '',
                        'serial': '',
                        'wgs84_latitude': -999.0,
                        'wgs84_longitude': -999.0,
                        'easting': -999.0,
                        'northing': -999.0,
                        'elevation': -999.0,
                        'chans_metadata': {
                            'Hx': {
                                'name': 'Hx',
                                'data_files': ['data.npy'],
                                'chan_type': 'magnetic',
                                'chan_source': None,
                                'sensor': '',
                                'serial': '',
                                'gain1': 1.0,
                                'gain2': 1.0,
                                'scaling': 1.0,
                                'chopper': False,
                                'dipole_dist': 1.0,
                                'sensor_calibration_file': '',
                                'instrument_calibration_file': ''
                            },
                            'Hy': {
                                'name': 'Hy',
                                'data_files': ['data.npy'],
```

(continues on next page)

(continued from previous page)

```
                'chan_type': 'magnetic',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Hz': {
                'name': 'Hz',
                'data_files': ['data.npy'],
                'chan_type': 'magnetic',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Ex': {
                'name': 'Ex',
                'data_files': ['data.npy'],
                'chan_type': 'electric',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Ey': {
                'name': 'Ey',
                'data_files': ['data.npy'],
                'chan_type': 'electric',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
```

(continues on next page)

(continued from previous page)

```
                            'chopper': False,
                            'dipole_dist': 1.0,
                            'sensor_calibration_file': '',
                            'instrument_calibration_file': ''
                        }
                    },
                    'history': {'records': []}
                },
                'reader': {
                    'name': 'TimeReaderNumpy',
                    'apply_scalings': True,
                    'extension': '.npy'
                }
            }
        },
        'begin_time': '2003-10-30 13:00:00.000000_000000_000000_000000',
        'end_time': '2003-11-23 12:41:25.000000_000000_000000_000000'
    },
    'kap163': {
        'dir_path': '../../data/project/kap03/time/kap163',
        'measurements': {
            'meas01': {
                'site_name': 'kap163',
                'dir_path': '../../data/project/kap03/time/kap163/meas01',
                'metadata': {
                    'file_info': {
                        'created_on_local': '2021-07-07T22:26:52.975361',
                        'created_on_utc': '2021-07-07T21:26:52.975361',
                        'version': '1.0.0a0'
                    },
                    'fs': 0.2,
                    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
                    'n_chans': 5,
                    'n_samples': 463807,
                    'first_time': '2003-10-28 15:30:00.000000_000000_000000_000000',
                    'last_time': '2003-11-24 11:40:30.000000_000000_000000_000000',
                    'system': '',
                    'serial': '',
                    'wgs84_latitude': -999.0,
                    'wgs84_longitude': -999.0,
                    'easting': -999.0,
                    'northing': -999.0,
                    'elevation': -999.0,
                    'chans_metadata': {
                        'Hx': {
                            'name': 'Hx',
                            'data_files': ['data.npy'],
                            'chan_type': 'magnetic',
                            'chan_source': None,
                            'sensor': '',
                            'serial': '',
                            'gain1': 1.0,
```

(continues on next page)

(continued from previous page)

```
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Hy': {
                'name': 'Hy',
                'data_files': ['data.npy'],
                'chan_type': 'magnetic',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Hz': {
                'name': 'Hz',
                'data_files': ['data.npy'],
                'chan_type': 'magnetic',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Ex': {
                'name': 'Ex',
                'data_files': ['data.npy'],
                'chan_type': 'electric',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
```

(continues on next page)

(continued from previous page)

```
                        'Ey': {
                            'name': 'Ey',
                            'data_files': ['data.npy'],
                            'chan_type': 'electric',
                            'chan_source': None,
                            'sensor': '',
                            'serial': '',
                            'gain1': 1.0,
                            'gain2': 1.0,
                            'scaling': 1.0,
                            'chopper': False,
                            'dipole_dist': 1.0,
                            'sensor_calibration_file': '',
                            'instrument_calibration_file': ''
                        }
                    },
                    'history': {'records': []}
                },
                'reader': {
                    'name': 'TimeReaderNumpy',
                    'apply_scalings': True,
                    'extension': '.npy'
                }
            }
        },
        'begin_time': '2003-10-28 15:30:00.000000_000000_000000_000000',
        'end_time': '2003-11-24 11:40:30.000000_000000_000000_000000'
    },
    'kap160': {
        'dir_path': '../../data/project/kap03/time/kap160',
        'measurements': {
            'meas01': {
                'site_name': 'kap160',
                'dir_path': '../../data/project/kap03/time/kap160/meas01',
                'metadata': {
                    'file_info': {
                        'created_on_local': '2021-07-07T22:26:48.851498',
                        'created_on_utc': '2021-07-07T21:26:48.851498',
                        'version': '1.0.0a0'
                    },
                    'fs': 0.2,
                    'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
                    'n_chans': 5,
                    'n_samples': 470544,
                    'first_time': '2003-10-28 10:00:00.000000_000000_000000_000000',
                    'last_time': '2003-11-24 15:31:55.000000_000000_000000_000000',
                    'system': '',
                    'serial': '',
                    'wgs84_latitude': -999.0,
                    'wgs84_longitude': -999.0,
                    'easting': -999.0,
                    'northing': -999.0,
```

(continues on next page)

(continued from previous page)

```
                        'elevation': -999.0,
                        'chans_metadata': {
                            'Hx': {
                                'name': 'Hx',
                                'data_files': ['data.npy'],
                                'chan_type': 'magnetic',
                                'chan_source': None,
                                'sensor': '',
                                'serial': '',
                                'gain1': 1.0,
                                'gain2': 1.0,
                                'scaling': 1.0,
                                'chopper': False,
                                'dipole_dist': 1.0,
                                'sensor_calibration_file': '',
                                'instrument_calibration_file': ''
                            },
                            'Hy': {
                                'name': 'Hy',
                                'data_files': ['data.npy'],
                                'chan_type': 'magnetic',
                                'chan_source': None,
                                'sensor': '',
                                'serial': '',
                                'gain1': 1.0,
                                'gain2': 1.0,
                                'scaling': 1.0,
                                'chopper': False,
                                'dipole_dist': 1.0,
                                'sensor_calibration_file': '',
                                'instrument_calibration_file': ''
                            },
                            'Hz': {
                                'name': 'Hz',
                                'data_files': ['data.npy'],
                                'chan_type': 'magnetic',
                                'chan_source': None,
                                'sensor': '',
                                'serial': '',
                                'gain1': 1.0,
                                'gain2': 1.0,
                                'scaling': 1.0,
                                'chopper': False,
                                'dipole_dist': 1.0,
                                'sensor_calibration_file': '',
                                'instrument_calibration_file': ''
                            },
                            'Ex': {
                                'name': 'Ex',
                                'data_files': ['data.npy'],
                                'chan_type': 'electric',
                                'chan_source': None,
```

(continues on next page)

(continued from previous page)

```
                                'sensor': '',
                                'serial': '',
                                'gain1': 1.0,
                                'gain2': 1.0,
                                'scaling': 1.0,
                                'chopper': False,
                                'dipole_dist': 1.0,
                                'sensor_calibration_file': '',
                                'instrument_calibration_file': ''
                            },
                            'Ey': {
                                'name': 'Ey',
                                'data_files': ['data.npy'],
                                'chan_type': 'electric',
                                'chan_source': None,
                                'sensor': '',
                                'serial': '',
                                'gain1': 1.0,
                                'gain2': 1.0,
                                'scaling': 1.0,
                                'chopper': False,
                                'dipole_dist': 1.0,
                                'sensor_calibration_file': '',
                                'instrument_calibration_file': ''
                            }
                        },
                        'history': {'records': []}
                    },
                    'reader': {
                        'name': 'TimeReaderNumpy',
                        'apply_scalings': True,
                        'extension': '.npy'
                    }
                }
            },
            'begin_time': '2003-10-28 10:00:00.000000_000000_000000_000000',
            'end_time': '2003-11-24 15:31:55.000000_000000_000000_000000'
        }
    }
}
```

Finally, plot the project timeline.

```
fig = resenv.proj.plot()
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 0.453 seconds)

Navigating a project

After creating a project and copying in the time data into the time folder, it is useful to be able to navigate a project. This example shows the various types of objects available in resistics that can help navigate a project and access data.

The data in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the data can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
import resistics.letsgo as letsgo
import plotly
```

Let's remind ourselves of the project contents, load the project and have a look at its contents.

```
project_path = Path("..", "..", "data", "project", "kap03")
sd.seedir(str(project_path), style="emoji")
resenv = letsgo.load(project_path)
```

```
 kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
├─ calibrate/
├─ evals/
├─ features/
└─ time/
  ├─ kap172/
  │  └─ meas01/
  │     ├─ metadata.json
  │     └─ data.npy
  ├─ kap163/
  │  └─ meas01/
  │     ├─ metadata.json
  │     └─ data.npy
  └─ kap160/
     └─ meas01/
        ├─ metadata.json
        └─ data.npy
```

Project summaries can be quite verbose. Instead, let's convert it to a pandas DataFrame and see the information in tabular form.

```
print(resenv.proj.to_dataframe())
```

```
     name   fs          first_time           last_time    site
0  meas01  0.2 2003-10-30 13:00:00 2003-11-23 12:41:25  kap172
1  meas01  0.2 2003-10-28 15:30:00 2003-11-24 11:40:30  kap163
2  meas01  0.2 2003-10-28 10:00:00 2003-11-24 15:31:55  kap160
```

The project has three sites, each with a single recording. Another way to look at the sites in the project is to make a list of them.

```
sites = [site.name for site in resenv.proj]
print(sites)
```

```
['kap172', 'kap163', 'kap160']
```

To get more information about a single site, get the corresponding Site object.

```
site = resenv.proj["kap160"]
print(type(site))
```

```
<class 'resistics.project.Site'>
```

Like most objects in resistics, the Site object has a summary method, which prints out a comprehensive summary of the site.

```
site.summary()
```

```
{
    'dir_path': '../../data/project/kap03/time/kap160',
    'measurements': {
        'meas01': {
            'site_name': 'kap160',
            'dir_path': '../../data/project/kap03/time/kap160/meas01',
            'metadata': {
                'file_info': {
                    'created_on_local': '2021-07-07T22:26:48.851498',
                    'created_on_utc': '2021-07-07T21:26:48.851498',
                    'version': '1.0.0a0'
                },
                'fs': 0.2,
                'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
                'n_chans': 5,
                'n_samples': 470544,
                'first_time': '2003-10-28 10:00:00.000000_000000_000000_000000',
                'last_time': '2003-11-24 15:31:55.000000_000000_000000_000000',
                'system': '',
                'serial': '',
                'wgs84_latitude': -999.0,
                'wgs84_longitude': -999.0,
                'easting': -999.0,
                'northing': -999.0,
                'elevation': -999.0,
                'chans_metadata': {
                    'Hx': {
                        'name': 'Hx',
                        'data_files': ['data.npy'],
                        'chan_type': 'magnetic',
                        'chan_source': None,
                        'sensor': '',
                        'serial': '',
                        'gain1': 1.0,
                        'gain2': 1.0,
```

(continues on next page)

(continued from previous page)

```
                    'scaling': 1.0,
                    'chopper': False,
                    'dipole_dist': 1.0,
                    'sensor_calibration_file': '',
                    'instrument_calibration_file': ''
                },
                'Hy': {
                    'name': 'Hy',
                    'data_files': ['data.npy'],
                    'chan_type': 'magnetic',
                    'chan_source': None,
                    'sensor': '',
                    'serial': '',
                    'gain1': 1.0,
                    'gain2': 1.0,
                    'scaling': 1.0,
                    'chopper': False,
                    'dipole_dist': 1.0,
                    'sensor_calibration_file': '',
                    'instrument_calibration_file': ''
                },
                'Hz': {
                    'name': 'Hz',
                    'data_files': ['data.npy'],
                    'chan_type': 'magnetic',
                    'chan_source': None,
                    'sensor': '',
                    'serial': '',
                    'gain1': 1.0,
                    'gain2': 1.0,
                    'scaling': 1.0,
                    'chopper': False,
                    'dipole_dist': 1.0,
                    'sensor_calibration_file': '',
                    'instrument_calibration_file': ''
                },
                'Ex': {
                    'name': 'Ex',
                    'data_files': ['data.npy'],
                    'chan_type': 'electric',
                    'chan_source': None,
                    'sensor': '',
                    'serial': '',
                    'gain1': 1.0,
                    'gain2': 1.0,
                    'scaling': 1.0,
                    'chopper': False,
                    'dipole_dist': 1.0,
                    'sensor_calibration_file': '',
                    'instrument_calibration_file': ''
                },
                'Ey': {
```

(continues on next page)

(continued from previous page)

```
                        'name': 'Ey',
                        'data_files': ['data.npy'],
                        'chan_type': 'electric',
                        'chan_source': None,
                        'sensor': '',
                        'serial': '',
                        'gain1': 1.0,
                        'gain2': 1.0,
                        'scaling': 1.0,
                        'chopper': False,
                        'dipole_dist': 1.0,
                        'sensor_calibration_file': '',
                        'instrument_calibration_file': ''
                    }
                },
                'history': {'records': []}
            },
            'reader': {
                'name': 'TimeReaderNumpy',
                'apply_scalings': True,
                'extension': '.npy'
            }
        }
    },
    'begin_time': '2003-10-28 10:00:00.000000_000000_000000_000000',
    'end_time': '2003-11-24 15:31:55.000000_000000_000000_000000'
}
```

Sometimes, it can be more convenient to access the information from the Site object directly.

```
print(site.name)
print(site.begin_time)
print(site.end_time)
measurements = [meas.name for meas in site]
print(measurements)
```

```
kap160
2003-10-28 10:00:00
2003-11-24 15:31:55
['meas01']
```

It's also possible to plot the timeline of a single site.

```
fig = site.plot()
plotly.io.show(fig)
```

There is only a single measurement in this site named "meas01". Let's get its Measurement object.

```
meas = site["meas01"]
print(type(meas))
```

```
<class 'resistics.project.Measurement'>
```

Unsurprisingly, Measurement objects also have a summary method.

```
meas.summary()
```

```
{
    'site_name': 'kap160',
    'dir_path': '../../data/project/kap03/time/kap160/meas01',
    'metadata': {
        'file_info': {
            'created_on_local': '2021-07-07T22:26:48.851498',
            'created_on_utc': '2021-07-07T21:26:48.851498',
            'version': '1.0.0a0'
        },
        'fs': 0.2,
        'chans': ['Hx', 'Hy', 'Hz', 'Ex', 'Ey'],
        'n_chans': 5,
        'n_samples': 470544,
        'first_time': '2003-10-28 10:00:00.000000_000000_000000_000000',
        'last_time': '2003-11-24 15:31:55.000000_000000_000000_000000',
        'system': '',
        'serial': '',
        'wgs84_latitude': -999.0,
        'wgs84_longitude': -999.0,
        'easting': -999.0,
        'northing': -999.0,
        'elevation': -999.0,
        'chans_metadata': {
            'Hx': {
                'name': 'Hx',
                'data_files': ['data.npy'],
                'chan_type': 'magnetic',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Hy': {
                'name': 'Hy',
                'data_files': ['data.npy'],
                'chan_type': 'magnetic',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
```

(continues on next page)

(continued from previous page)

```
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Hz': {
                'name': 'Hz',
                'data_files': ['data.npy'],
                'chan_type': 'magnetic',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Ex': {
                'name': 'Ex',
                'data_files': ['data.npy'],
                'chan_type': 'electric',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            },
            'Ey': {
                'name': 'Ey',
                'data_files': ['data.npy'],
                'chan_type': 'electric',
                'chan_source': None,
                'sensor': '',
                'serial': '',
                'gain1': 1.0,
                'gain2': 1.0,
                'scaling': 1.0,
                'chopper': False,
                'dipole_dist': 1.0,
                'sensor_calibration_file': '',
                'instrument_calibration_file': ''
            }
```

(continues on next page)

(continued from previous page)

```
        },
        'history': {'records': []}
    },
    'reader': {
        'name': 'TimeReaderNumpy',
        'apply_scalings': True,
        'extension': '.npy'
    }
}
```

Measurement objects only hold metadata to avoid loading in lots of data when projects are loaded. However, it is possible to read the data from the measurement.

```
time_data = meas.reader.run(meas.dir_path, metadata=meas.metadata)
time_data.summary()
```

```
##---Begin Summary-------------------------
<class 'resistics.time.TimeData'>
file_info:
    created_on_local: '2021-07-07T22:26:48.851498'
    created_on_utc: '2021-07-07T21:26:48.851498'
    version: 1.0.0a0
fs: 0.2
chans:
- Hx
- Hy
- Hz
- Ex
- Ey
n_chans: 5
n_samples: 470544
first_time: 2003-10-28 10:00:00.000000_000000_000000_000000
last_time: 2003-11-24 15:31:55.000000_000000_000000_000000
system: ''
serial: ''
wgs84_latitude: -999.0
wgs84_longitude: -999.0
easting: -999.0
northing: -999.0
elevation: -999.0
chans_metadata:
    Hx:
        name: Hx
        data_files:
        - data.npy
        chan_type: magnetic
        chan_source: null
        sensor: ''
        serial: ''
        gain1: 1.0
        gain2: 1.0
        scaling: 1.0
```

(continues on next page)

(continued from previous page)

```
        chopper: false
        dipole_dist: 1.0
        sensor_calibration_file: ''
        instrument_calibration_file: ''
    Hy:
        name: Hy
        data_files:
        - data.npy
        chan_type: magnetic
        chan_source: null
        sensor: ''
        serial: ''
        gain1: 1.0
        gain2: 1.0
        scaling: 1.0
        chopper: false
        dipole_dist: 1.0
        sensor_calibration_file: ''
        instrument_calibration_file: ''
    Hz:
        name: Hz
        data_files:
        - data.npy
        chan_type: magnetic
        chan_source: null
        sensor: ''
        serial: ''
        gain1: 1.0
        gain2: 1.0
        scaling: 1.0
        chopper: false
        dipole_dist: 1.0
        sensor_calibration_file: ''
        instrument_calibration_file: ''
    Ex:
        name: Ex
        data_files:
        - data.npy
        chan_type: electric
        chan_source: null
        sensor: ''
        serial: ''
        gain1: 1.0
        gain2: 1.0
        scaling: 1.0
        chopper: false
        dipole_dist: 1.0
        sensor_calibration_file: ''
        instrument_calibration_file: ''
    Ey:
        name: Ey
        data_files:
```

(continues on next page)

(continued from previous page)

```
        - data.npy
        chan_type: electric
        chan_source: null
        sensor: ''
        serial: ''
        gain1: 1.0
        gain2: 1.0
        scaling: 1.0
        chopper: false
        dipole_dist: 1.0
        sensor_calibration_file: ''
        instrument_calibration_file: ''
history:
    records:
    -   time_local: '2023-05-12T14:40:46.976075'
        time_utc: '2023-05-12T14:40:46.976074'
        creator:
            name: TimeReaderNumpy
            apply_scalings: true
            extension: .npy
        messages:
        - Reading raw data from ../../data/project/kap03/time/kap160/meas01
        - Sampling frequency 0.2 Hz
        - 'From sample, time: 0, 2003-10-28 10:00:00'
        - 'To sample, time: 470543, 2003-11-24 15:31:55'
        record_type: process

##---End summary---------------------------
```

Let's plot the time data.

```
fig = time_data.plot()
fig.update_layout(height=700)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 7.512 seconds)

Processing a project

The quick reading functionality of resistics focuses on analysis of single continuous recordings. When there are multiple recordings at a site or multiple sites, it can be more convenient to use a resistics project. This is generally easier to manage and use, especially when doing remote reference or intersite processing.

The data in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the data can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
import resistics.letsgo as letsgo
import plotly
```

Let's remind ourselves of the project contents and then load the project.

```
project_path = Path("..", "..", "data", "project", "kap03")
sd.seedir(str(project_path), style="emoji")
resenv = letsgo.load(project_path)
```

```
kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
├─ calibrate/
├─ evals/
├─ features/
└─ time/
  ├─ kap172/
  │  └─ meas01/
  │    ├─ metadata.json
  │    └─ data.npy
  ├─ kap163/
  │  └─ meas01/
  │    ├─ metadata.json
  │    └─ data.npy
  └─ kap160/
    └─ meas01/
      ├─ metadata.json
      └─ data.npy
```

Inspect the current configuration. As no custom configuration has been specified, this will be the default configuration.

```
resenv.config.summary()
```

```
{
    'name': 'default',
    'time_readers': [
        {
            'name': 'TimeReaderAscii',
            'apply_scalings': True,
            'extension': '.txt',
            'delimiter': None,
            'n_header': 0
        },
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
        }
    ],
    'time_processors': [
        {'name': 'InterpolateNans'},
        {'name': 'RemoveMean'}
    ],
    'dec_setup': {
```

(continues on next page)

(continued from previous page)

```
        'name': 'DecimationSetup',
        'n_levels': 8,
        'per_level': 5,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
        'resample': True,
        'max_single_factor': 3
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 128,
        'min_olap': 32,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {'name': 'Windower'},
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'ImpedanceTensor',
        'variation': 'default',
        'out_chans': ['Ex', 'Ey'],
        'in_chans': ['Hx', 'Hy'],
        'cross_chans': ['Hx', 'Hy'],
        'n_out': 2,
        'n_in': 2,
        'n_cross': 2
    },
```

(continues on next page)

(continued from previous page)

```
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
}
```

And it's always useful to know what transfer function will be calculated out.

```
print(resenv.config.tf)
```

```
| Ex | = | Ex_Hx Ex_Hy | | Hx |
| Ey |   | Ey_Hx Ey_Hy | | Hy |
```

Now let's run single site processing on a site and then look at the directory structure again. Begin by transforming to frequency domain and reducing to the evaluation frequencies. Note that whilst there is only a single measurement for this site, the below is written to work when there are more measurements.

```
site = resenv.proj["kap160"]
for meas in site:
    letsgo.process_time_to_evals(resenv, "kap160", meas.name)
sd.seedir(str(project_path), style="emoji")
```

```
 kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
├─ calibrate/
├─ evals/
│  └─ kap160/
│     └─ default/
│        └─ meas01/
│           ├─ data.npz
│           └─ metadata.json
├─ features/
└─ time/
   ├─ kap172/
   │  └─ meas01/
   │     ├─ metadata.json
   │     └─ data.npy
   ├─ kap163/
   │  └─ meas01/
   │     ├─ metadata.json
   │     └─ data.npy
   └─ kap160/
```

(continues on next page)

(continued from previous page)

```
    └─ meas01/
      ├─ metadata.json
      └─ data.npy
```

Now let's run single site processing on a site and then look at the directory structure again. To run the transfer function calculation, the sampling frequency to process needs to be specified. In this case, it's 0.2 Hz.

```
letsgo.process_evals_to_tf(resenv, 0.2, "kap160")
sd.seedir(str(project_path), style="emoji")
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
 15%|#5        | 3/20 [00:00<00:00, 22.72it/s]
 30%|###       | 6/20 [00:00<00:00, 26.34it/s]
100%|##########| 20/20 [00:00<00:00, 69.56it/s]

  0%|          | 0/20 [00:00<?, ?it/s]
  5%|5         | 1/20 [00:01<00:30,  1.63s/it]
 10%|#         | 2/20 [00:03<00:29,  1.62s/it]
 15%|#5        | 3/20 [00:04<00:27,  1.62s/it]
 20%|##        | 4/20 [00:06<00:25,  1.62s/it]
 25%|##5       | 5/20 [00:08<00:24,  1.65s/it]
 30%|###       | 6/20 [00:08<00:17,  1.25s/it]
 35%|###5      | 7/20 [00:09<00:13,  1.00s/it]
 40%|####      | 8/20 [00:09<00:10,  1.19it/s]
 45%|####5     | 9/20 [00:10<00:08,  1.37it/s]
 50%|#####     | 10/20 [00:10<00:06,  1.52it/s]
 55%|#####5    | 11/20 [00:10<00:04,  1.99it/s]
 60%|######    | 12/20 [00:10<00:03,  2.52it/s]
 65%|######5   | 13/20 [00:11<00:02,  3.09it/s]
 70%|#######   | 14/20 [00:11<00:01,  3.67it/s]
 75%|#######5  | 15/20 [00:11<00:01,  4.24it/s]
 80%|########  | 16/20 [00:11<00:00,  4.95it/s]
 85%|########5 | 17/20 [00:11<00:00,  5.61it/s]
 90%|######### | 18/20 [00:11<00:00,  6.17it/s]
 95%|#########5| 19/20 [00:11<00:00,  6.66it/s]
100%|##########| 20/20 [00:12<00:00,  7.04it/s]
100%|##########| 20/20 [00:12<00:00,  1.66it/s]
 kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
│  └─ kap160/
│    └─ default/
│      └─ 0_200000_impedancetensor_default.json
├─ calibrate/
├─ evals/
│  └─ kap160/
│    └─ default/
│      └─ meas01/
│        ├─ data.npz
```

(continues on next page)

(continued from previous page)

```
│       └─ metadata.json
├─ features/
└─ time/
  ├─ kap172/
  │  └─ meas01/
  │    ├─ metadata.json
  │    └─ data.npy
  ├─ kap163/
  │  └─ meas01/
  │    ├─ metadata.json
  │    └─ data.npy
  └─ kap160/
    └─ meas01/
      ├─ metadata.json
      └─ data.npy
```

Get the transfer function

```
soln = letsgo.get_solution(
    resenv,
    "kap160",
    resenv.config.name,
    0.2,
    resenv.config.tf.name,
    resenv.config.tf.variation,
)
fig = soln.tf.plot(
    soln.freqs,
    soln.components,
    to_plot=["ExHy", "EyHx"],
    x_lim=[1, 5],
    res_lim=[1, 4],
    legend="128",
    symbol="circle",
)
fig.update_layout(height=900)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 13.387 seconds)

Calibration files

So far, none of our project examples use calibration files. Let's look at how calibration files can be used in the processing sequence.

> **Warning:** I need to find an appropriate example where calibration files are required and that can be shared online. If anyone can provide some data that can be openly shared, please get in touch.

Searching for an appropriate example

```
print("Get in touch if you have one")
```

```
Get in touch if you have one
```

Total running time of the script: (0 minutes 0.001 seconds)

Remote reference

Warning: Remote reference processing does not seem to be working yet. I need to get to the bottom of it.

There's a bug in this soup

```
print("Not working properly yet, watch this space")
```

```
Not working properly yet, watch this space
```

Total running time of the script: (0 minutes 0.001 seconds)

Intersite processing

Warning: Intersite processing does not seem to be working yet. I need to get to the bottom of it.

There's a bug in this soup

```
print("Not working properly yet, watch this space")
```

```
Not working properly yet, watch this space
```

Total running time of the script: (0 minutes 0.001 seconds)

Remote and intersite

Warning: Remote reference and intersite processing does not seem to be working yet. I need to get to the bottom of it.

There's a bug in this soup

```
print("Not working properly yet, watch this space")
```

```
Not working properly yet, watch this space
```

Total running time of the script: (0 minutes 0.001 seconds)

Configuration

There will be times when users need to customise their processing sequences. This can be achieved using resistics configuration files.

The resistics configuration files allow users to:

- Specify processing parameters
- Save processing sequencies for use later on

Default configuration

This example shows the default resistics configuration. The configuration defines the processing sequence and parameterisation that will be used to process the data.

```
from resistics.config import get_default_configuration
```

Get the default configuration and print the summary.

```
default_config = get_default_configuration()
default_config.summary()
```

```
{
    'name': 'default',
    'time_readers': [
        {
            'name': 'TimeReaderAscii',
            'apply_scalings': True,
            'extension': '.txt',
            'delimiter': None,
            'n_header': 0
        },
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
        }
    ],
    'time_processors': [
        {'name': 'InterpolateNans'},
        {'name': 'RemoveMean'}
    ],
    'dec_setup': {
        'name': 'DecimationSetup',
        'n_levels': 8,
        'per_level': 5,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
```

(continues on next page)

(continued from previous page)

```
        'resample': True,
        'max_single_factor': 3
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 128,
        'min_olap': 32,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {'name': 'Windower'},
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'ImpedanceTensor',
        'variation': 'default',
        'out_chans': ['Ex', 'Ey'],
        'in_chans': ['Hx', 'Hy'],
        'cross_chans': ['Hx', 'Hy'],
        'n_out': 2,
        'n_in': 2,
        'n_cross': 2
    },
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
```

(continues on next page)

(continued from previous page)

```
}
```

By default, the configuration includes two time data readers. These will be used to try and read any data. Each has parameters that can be altered depending on the type of data. More time readers for particular data formats are available in the resistics-readers package.

```
for time_reader in default_config.time_readers:
    time_reader.summary()
```

```
{
    'name': 'TimeReaderAscii',
    'apply_scalings': True,
    'extension': '.txt',
    'delimiter': None,
    'n_header': 0
}
{
    'name': 'TimeReaderNumpy',
    'apply_scalings': True,
    'extension': '.npy'
}
```

The default transfer function is the magnetotelluric impedance tensor. It can be printed out to help show the relationship.

```
default_config.tf.summary()
print(default_config.tf)
```

```
{
    'name': 'ImpedanceTensor',
    'variation': 'default',
    'out_chans': ['Ex', 'Ey'],
    'in_chans': ['Hx', 'Hy'],
    'cross_chans': ['Hx', 'Hy'],
    'n_out': 2,
    'n_in': 2,
    'n_cross': 2
}
| Ex | = | Ex_Hx Ex_Hy | | Hx |
| Ey |   | Ey_Hx Ey_Hy | | Hy |
```

Other important parameters include those related to decimation setup and windowing setup.

```
default_config.win_setup.summary()
default_config.dec_setup.summary()
```

```
{
    'name': 'WindowSetup',
    'min_size': 128,
    'min_olap': 32,
    'win_factor': 4,
    'olap_proportion': 0.25,
```

(continues on next page)

(continued from previous page)

```
    'min_n_wins': 5,
    'win_sizes': None,
    'olap_sizes': None
}
{
    'name': 'DecimationSetup',
    'n_levels': 8,
    'per_level': 5,
    'min_samples': 256,
    'div_factor': 2,
    'eval_freqs': None
}
```

Total running time of the script: (0 minutes 0.024 seconds)

Custom configuration

It is possible to customise the configuration and save it for use later. Configurations can either be customised at initialisation or after initialisation.

Configurations can be saved as JSON files and later reloaded. This allows users to keep a library of configurations that can be used depending on the use case or the survey.

```
from pathlib import Path
from resistics.config import Configuration
from resistics.time import Add
from resistics.transfunc import TransferFunction
```

Creating a new configuration requires only a name. In this instance, default parameters will be used for everything else.

```
custom_config = Configuration(name="example")
custom_config.summary()
```

```
{
    'name': 'example',
    'time_readers': [
        {
            'name': 'TimeReaderAscii',
            'apply_scalings': True,
            'extension': '.txt',
            'delimiter': None,
            'n_header': 0
        },
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
        }
    ],
    'time_processors': [
        {'name': 'InterpolateNans'},
```

(continues on next page)

(continued from previous page)

```
        {'name': 'RemoveMean'}
    ],
    'dec_setup': {
        'name': 'DecimationSetup',
        'n_levels': 8,
        'per_level': 5,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
        'resample': True,
        'max_single_factor': 3
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 128,
        'min_olap': 32,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {'name': 'Windower'},
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'ImpedanceTensor',
        'variation': 'default',
        'out_chans': ['Ex', 'Ey'],
        'in_chans': ['Hx', 'Hy'],
        'cross_chans': ['Hx', 'Hy'],
        'n_out': 2,
```

(continues on next page)

(continued from previous page)

```
        'n_in': 2,
        'n_cross': 2
    },
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
}
```

However, it is possible to customise more at initialisation time.

```
custom_config = Configuration(
    name="example",
    time_processors=[Add(add=5)],
    tf=TransferFunction(in_chans=["A", "B"], out_chans=["M", "N"]),
)
custom_config.summary()
```

```
{
    'name': 'example',
    'time_readers': [
        {
            'name': 'TimeReaderAscii',
            'apply_scalings': True,
            'extension': '.txt',
            'delimiter': None,
            'n_header': 0
        },
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
        }
    ],
    'time_processors': [{'name': 'Add', 'add': 5.0}],
    'dec_setup': {
        'name': 'DecimationSetup',
        'n_levels': 8,
        'per_level': 5,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
        'resample': True,
        'max_single_factor': 3
```

(continues on next page)

(continued from previous page)

```
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 128,
        'min_olap': 32,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {'name': 'Windower'},
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'TransferFunction',
        'variation': 'generic',
        'out_chans': ['M', 'N'],
        'in_chans': ['A', 'B'],
        'cross_chans': ['A', 'B'],
        'n_out': 2,
        'n_in': 2,
        'n_cross': 2
    },
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
}
```

A configuration can be updated after it has been initialised. For example, let's update a windowing parameter. First,

have a look at the summary of the windowing parameters. Then they can be updated and the summary can be inspected again.

```
custom_config.win_setup.summary()
custom_config.win_setup.min_size = 512
custom_config.win_setup.summary()
```

```
{
    'name': 'WindowSetup',
    'min_size': 128,
    'min_olap': 32,
    'win_factor': 4,
    'olap_proportion': 0.25,
    'min_n_wins': 5,
    'win_sizes': None,
    'olap_sizes': None
}
{
    'name': 'WindowSetup',
    'min_size': 512,
    'min_olap': 32,
    'win_factor': 4,
    'olap_proportion': 0.25,
    'min_n_wins': 5,
    'win_sizes': None,
    'olap_sizes': None
}
```

Configuration information can be saved to JSON files.

```
save_path = Path("..", "..", "data", "config", "custom_config.json")
with save_path.open("w") as f:
    f.write(custom_config.json())
```

Configurations can also be loaded from JSON files.

```
reloaded_config = Configuration.parse_file(save_path)
reloaded_config.summary()
```

```
{
    'name': 'example',
    'time_readers': [
        {
            'name': 'TimeReaderAscii',
            'apply_scalings': True,
            'extension': '.txt',
            'delimiter': None,
            'n_header': 0
        },
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
```

(continues on next page)

(continued from previous page)

```
        }
    ],
    'time_processors': [{'name': 'Add', 'add': 5.0}],
    'dec_setup': {
        'name': 'DecimationSetup',
        'n_levels': 8,
        'per_level': 5,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
        'resample': True,
        'max_single_factor': 3
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 512,
        'min_olap': 32,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {'name': 'Windower'},
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14.0],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'TransferFunction',
        'variation': 'generic',
        'out_chans': ['M', 'N'],
        'in_chans': ['A', 'B'],
        'cross_chans': ['A', 'B'],
```

(continues on next page)

(continued from previous page)

```
        'n_out': 2,
        'n_in': 2,
        'n_cross': 2
    },
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
}
```

Total running time of the script: (0 minutes 0.049 seconds)

Quick configuration

If no configuration is passed, the quick processing functions in resistics will use the default configuration. However, it is possible to use a different configuration if preferred.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
import resistics.letsgo as letsgo
from resistics.config import Configuration
from resistics.time import InterpolateNans, RemoveMean, Multiply
from resistics.decimate import DecimationSetup
from resistics.window import WindowerTarget
import plotly
```

Define the data path. This is dependent on where the data is stored.

```
time_data_path = Path("..", "..", "data", "time", "quick", "kap123")
sd.seedir(str(time_data_path), style="emoji")
```

```
kap123/
├─ metadata.json
└─ data.npy
```

Quick calculation of the transfer function using default parameters.

```
soln_default = letsgo.quick_tf(time_data_path)
fig = soln_default.tf.plot(
    soln_default.freqs,
    soln_default.components,
    to_plot=["ExHy", "EyHx"],
    x_lim=[1, 5],
```

(continues on next page)

(continued from previous page)

```
    res_lim=[0, 4],
    legend="Default config",
    symbol="circle",
)
fig.update_layout(height=800)
fig
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
 10%|#         | 2/20 [00:00<00:01, 16.64it/s]
 25%|##5       | 5/20 [00:00<00:00, 22.97it/s]
100%|##########| 20/20 [00:00<00:00, 70.90it/s]

  0%|          | 0/20 [00:00<?, ?it/s]
  5%|5         | 1/20 [00:01<00:23,  1.24s/it]
 10%|#         | 2/20 [00:02<00:22,  1.24s/it]
 15%|#5        | 3/20 [00:03<00:21,  1.25s/it]
 20%|##        | 4/20 [00:04<00:19,  1.25s/it]
 25%|##5       | 5/20 [00:06<00:18,  1.25s/it]
 30%|###       | 6/20 [00:06<00:13,  1.06it/s]
 35%|###5      | 7/20 [00:06<00:09,  1.34it/s]
 40%|####      | 8/20 [00:07<00:07,  1.62it/s]
 45%|####5     | 9/20 [00:07<00:05,  1.88it/s]
 50%|#####     | 10/20 [00:07<00:04,  2.12it/s]
 55%|#####5    | 11/20 [00:08<00:03,  2.68it/s]
 60%|######    | 12/20 [00:08<00:02,  3.30it/s]
 65%|######5   | 13/20 [00:08<00:01,  3.91it/s]
 70%|#######   | 14/20 [00:08<00:01,  4.51it/s]
 75%|#######5  | 15/20 [00:08<00:00,  5.05it/s]
 80%|########  | 16/20 [00:08<00:00,  5.73it/s]
 85%|########5 | 17/20 [00:08<00:00,  6.34it/s]
 90%|######### | 18/20 [00:09<00:00,  6.81it/s]
 95%|#########5| 19/20 [00:09<00:00,  7.26it/s]
100%|##########| 20/20 [00:09<00:00,  7.56it/s]
100%|##########| 20/20 [00:09<00:00,  2.16it/s]
```

Looking at the transfer function, it's clear that the phases are in the wrong quadrants. A new time process can be added to correct this by multiplying the electric channels by -1.

Further, let's use a different windower that will change the window size (subject to a minimum) to try and generate a target number of windows. The WindowTarget ignores the min_size in the WindowSetup and uses its own. This alternative windower will be combined with a modified decimation setup.

```
config = Configuration(
    name="custom",
    time_processors=[
        InterpolateNans(),
        RemoveMean(),
        Multiply(multiplier={"Ex": -1, "Ey": -1}),
    ],
    dec_setup=DecimationSetup(n_levels=3, per_level=7),
    windower=WindowerTarget(target=2_000, min_size=180),
)
```

(continues on next page)

(continued from previous page)

```
config.summary()
```

```
{
    'name': 'custom',
    'time_readers': [
        {
            'name': 'TimeReaderAscii',
            'apply_scalings': True,
            'extension': '.txt',
            'delimiter': None,
            'n_header': 0
        },
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
        }
    ],
    'time_processors': [
        {'name': 'InterpolateNans'},
        {'name': 'RemoveMean'},
        {'name': 'Multiply', 'multiplier': {'Ex': -1.0, 'Ey': -1.0}}
    ],
    'dec_setup': {
        'name': 'DecimationSetup',
        'n_levels': 3,
        'per_level': 7,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
        'resample': True,
        'max_single_factor': 3
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 128,
        'min_olap': 32,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {
        'name': 'WindowerTarget',
        'target': 2000,
        'min_size': 180,
        'olap_proportion': 0.25
```

(continues on next page)

(continued from previous page)

```
    },
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'ImpedanceTensor',
        'variation': 'default',
        'out_chans': ['Ex', 'Ey'],
        'in_chans': ['Hx', 'Hy'],
        'cross_chans': ['Hx', 'Hy'],
        'n_out': 2,
        'n_in': 2,
        'n_cross': 2
    },
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
}
```

Quick calculate the impedance tensor using the new custom configuration and plot the result.

```
soln_custom = letsgo.quick_tf(time_data_path, config)
fig = soln_custom.tf.plot(
    soln_custom.freqs,
    soln_custom.components,
    to_plot=["ExHy", "EyHx"],
    x_lim=[1, 5],
    res_lim=[0, 4],
    phs_lim=[0, 100],
    legend="Custom config",
```

(continues on next page)

(continued from previous page)

```
    symbol="diamond",
)
fig.update_layout(height=800)
plotly.io.show(fig)
```

```
  0%|          | 0/21 [00:00<?, ?it/s]
 14%|#4        | 3/21 [00:00<00:00, 29.07it/s]
 86%|########5 | 18/21 [00:00<00:00, 99.25it/s]
100%|##########| 21/21 [00:00<00:00, 102.77it/s]

  0%|          | 0/21 [00:00<?, ?it/s]
  5%|4         | 1/21 [00:00<00:13,  1.46it/s]
 10%|9         | 2/21 [00:01<00:13,  1.45it/s]
 14%|#4        | 3/21 [00:02<00:12,  1.45it/s]
 19%|#9        | 4/21 [00:02<00:11,  1.46it/s]
 24%|##3       | 5/21 [00:03<00:11,  1.45it/s]
 29%|##8       | 6/21 [00:04<00:10,  1.45it/s]
 33%|###3      | 7/21 [00:04<00:09,  1.45it/s]
 38%|###8      | 8/21 [00:05<00:06,  1.87it/s]
 43%|####2     | 9/21 [00:05<00:05,  2.33it/s]
 48%|####7     | 10/21 [00:05<00:03,  2.78it/s]
 52%|#####2    | 11/21 [00:05<00:03,  3.21it/s]
 57%|#####7    | 12/21 [00:05<00:02,  3.61it/s]
 62%|######1   | 13/21 [00:06<00:02,  3.93it/s]
 67%|######6   | 14/21 [00:06<00:01,  4.20it/s]
 71%|#######1  | 15/21 [00:06<00:01,  4.95it/s]
 76%|#######6  | 16/21 [00:06<00:00,  5.67it/s]
 81%|########  | 17/21 [00:06<00:00,  6.30it/s]
 86%|########5 | 18/21 [00:06<00:00,  6.85it/s]
 90%|######### | 19/21 [00:06<00:00,  7.28it/s]
 95%|#########5| 20/21 [00:06<00:00,  7.64it/s]
100%|##########| 21/21 [00:07<00:00,  7.92it/s]
100%|##########| 21/21 [00:07<00:00,  2.98it/s]
```

Total running time of the script: (0 minutes 17.849 seconds)

Project configuration

Alternative configurations can also be used with projects. When using cutom configurations in the project environment, the name of the configuration is key as this will determine where any data is saved. The below example shows what happens when using different configurations with a project.

The dataset in this example has been provided for use by the SAMTEX consortium. For more information, please refer to [Jones2009]. Additional details about the dataset can be found at https://www.mtnet.info/data/kap03/kap03.html.

```
from pathlib import Path
import seedir as sd
from resistics.config import Configuration
import resistics.letsgo as letsgo
from resistics.time import TimeReaderNumpy, InterpolateNans, RemoveMean, Multiply
from resistics.decimate import DecimationSetup
```

(continues on next page)

(continued from previous page)

```
from resistics.window import WindowSetup
import plotly

# The first thing to do is define the configuration to use.
myconfig = letsgo.Configuration(
    name="myconfig",
    time_readers=[TimeReaderNumpy()],
    time_processors=[
        InterpolateNans(),
        RemoveMean(),
        Multiply(multiplier={"Ex": -1, "Ey": -1}),
    ],
    dec_setup=DecimationSetup(n_levels=7, per_level=3),
    win_setup=WindowSetup(min_size=64, min_olap=16),
)
myconfig.summary()
```

```
{
    'name': 'myconfig',
    'time_readers': [
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
        }
    ],
    'time_processors': [
        {'name': 'InterpolateNans'},
        {'name': 'RemoveMean'},
        {'name': 'Multiply', 'multiplier': {'Ex': -1.0, 'Ey': -1.0}}
    ],
    'dec_setup': {
        'name': 'DecimationSetup',
        'n_levels': 7,
        'per_level': 3,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
        'resample': True,
        'max_single_factor': 3
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 64,
        'min_olap': 16,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
```

(continues on next page)

(continued from previous page)

```
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {'name': 'Windower'},
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'ImpedanceTensor',
        'variation': 'default',
        'out_chans': ['Ex', 'Ey'],
        'in_chans': ['Hx', 'Hy'],
        'cross_chans': ['Hx', 'Hy'],
        'n_out': 2,
        'n_in': 2,
        'n_cross': 2
    },
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
}
```

Save the configuration to a file. This is to imitate scenarios where users have an existing configuration file that they want to load in and use.

```
myconfig_path = Path("..", "..", "data", "config", "myconfig.json")
with myconfig_path.open("w") as f:
    f.write(myconfig.json())
```

Let's remind ourselves of the project contents. Note that some processing with default parameters has already taken place.

```
project_path = Path("..", "..", "data", "project", "kap03")
sd.seedir(str(project_path), style="emoji")
```

```
kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
│  └─ kap160/
│     └─ default/
│        └─ 0_200000_impedancetensor_default.json
├─ calibrate/
├─ evals/
│  └─ kap160/
│     └─ default/
│        └─ meas01/
│           ├─ data.npz
│           └─ metadata.json
├─ features/
└─ time/
   ├─ kap172/
   │  └─ meas01/
   │     ├─ metadata.json
   │     └─ data.npy
   ├─ kap163/
   │  └─ meas01/
   │     ├─ metadata.json
   │     └─ data.npy
   └─ kap160/
      └─ meas01/
         ├─ metadata.json
         └─ data.npy
```

Now load our configuration and the project with myconfig.

```
config = Configuration.parse_file(myconfig_path)
resenv = letsgo.load(project_path, config=config)
resenv.config.summary()
```

```
{
    'name': 'myconfig',
    'time_readers': [
        {
            'name': 'TimeReaderNumpy',
            'apply_scalings': True,
            'extension': '.npy'
        }
    ],
    'time_processors': [
        {'name': 'InterpolateNans'},
        {'name': 'RemoveMean'},
```

(continues on next page)

(continued from previous page)

```
        {'name': 'Multiply', 'multiplier': {'Ex': -1.0, 'Ey': -1.0}}
    ],
    'dec_setup': {
        'name': 'DecimationSetup',
        'n_levels': 7,
        'per_level': 3,
        'min_samples': 256,
        'div_factor': 2,
        'eval_freqs': None
    },
    'decimator': {
        'name': 'Decimator',
        'resample': True,
        'max_single_factor': 3
    },
    'win_setup': {
        'name': 'WindowSetup',
        'min_size': 64,
        'min_olap': 16,
        'win_factor': 4,
        'olap_proportion': 0.25,
        'min_n_wins': 5,
        'win_sizes': None,
        'olap_sizes': None
    },
    'windower': {'name': 'Windower'},
    'fourier': {
        'name': 'FourierTransform',
        'win_fnc': ['kaiser', 14.0],
        'detrend': 'linear',
        'workers': -2
    },
    'spectra_processors': [],
    'evals': {'name': 'EvaluationFreqs'},
    'sensor_calibrator': {
        'name': 'SensorCalibrator',
        'chans': None,
        'readers': [
            {
                'name': 'SensorCalibrationJSON',
                'extension': '.json',
                'file_str': 'IC_$sensor$extension'
            }
        ]
    },
    'tf': {
        'name': 'ImpedanceTensor',
        'variation': 'default',
        'out_chans': ['Ex', 'Ey'],
        'in_chans': ['Hx', 'Hy'],
        'cross_chans': ['Hx', 'Hy'],
        'n_out': 2,
```

(continues on next page)

(continued from previous page)

```
        'n_in': 2,
        'n_cross': 2
    },
    'regression_preparer': {'name': 'RegressionPreparerGathered'},
    'solver': {
        'name': 'SolverScikitTheilSen',
        'fit_intercept': False,
        'normalize': False,
        'n_jobs': -2,
        'max_subpopulation': 2000,
        'n_subsamples': None
    }
}
```

Now calculate the evaluation frequency spectral data and view the directory structure. This shows how resistics handles saving data for different configurations. The data is placed in a new folder with the same name as the the configuration. This is why the configuration name is important.

```
site = resenv.proj["kap160"]
for meas in site:
    letsgo.process_time_to_evals(resenv, "kap160", meas.name)
sd.seedir(str(project_path), style="emoji")
```

```
 kap03/
├─ spectra/
├─ images/
├─ resistics.json
├─ masks/
├─ results/
│  └─ kap160/
│     └─ default/
│        └─ 0_200000_impedancetensor_default.json
├─ calibrate/
├─ evals/
│  └─ kap160/
│     ├─ default/
│     │  └─ meas01/
│     │     ├─ data.npz
│     │     └─ metadata.json
│     └─ myconfig/
│        └─ meas01/
│           ├─ data.npz
│           └─ metadata.json
├─ features/
└─ time/
   ├─ kap172/
   │  └─ meas01/
   │     ├─ metadata.json
   │     └─ data.npy
   ├─ kap163/
   │  └─ meas01/
   │     ├─ metadata.json
```

(continues on next page)

(continued from previous page)

```
    └── data.npy
  └── kap160/
    └── meas01/
      ├── metadata.json
      └── data.npy
```

Let's calculate the impedance tensor with this configuration. The sampling frequency to process is 0.2 (Hz)

```
letsgo.process_evals_to_tf(resenv, 0.2, "kap160")
sd.seedir(str(project_path), style="emoji")
```

```
  0%|          | 0/21 [00:00<?, ?it/s]
 10%|9         | 2/21 [00:00<00:01, 11.35it/s]
 19%|#9        | 4/21 [00:00<00:01, 13.19it/s]
 38%|###8      | 8/21 [00:00<00:00, 21.89it/s]
100%|##########| 21/21 [00:00<00:00, 45.89it/s]

  0%|          | 0/21 [00:00<?, ?it/s]
  5%|4         | 1/21 [00:03<01:03,  3.16s/it]
 10%|9         | 2/21 [00:06<01:00,  3.16s/it]
 14%|#4        | 3/21 [00:09<00:56,  3.16s/it]
 19%|#9        | 4/21 [00:11<00:43,  2.55s/it]
 24%|##3       | 5/21 [00:12<00:35,  2.22s/it]
 29%|##8       | 6/21 [00:14<00:30,  2.01s/it]
 33%|###3      | 7/21 [00:14<00:21,  1.51s/it]
 38%|###8      | 8/21 [00:15<00:15,  1.19s/it]
 43%|####2     | 9/21 [00:15<00:11,  1.03it/s]
 48%|####7     | 10/21 [00:16<00:08,  1.32it/s]
 52%|#####2    | 11/21 [00:16<00:06,  1.64it/s]
 57%|#####7    | 12/21 [00:16<00:04,  1.98it/s]
 62%|######1   | 13/21 [00:16<00:03,  2.51it/s]
 67%|######6   | 14/21 [00:16<00:02,  3.08it/s]
 71%|#######1  | 15/21 [00:17<00:01,  3.65it/s]
 76%|#######6  | 16/21 [00:17<00:01,  4.31it/s]
 81%|########  | 17/21 [00:17<00:00,  4.93it/s]
 86%|########5 | 18/21 [00:17<00:00,  5.49it/s]
 90%|######### | 19/21 [00:17<00:00,  6.17it/s]
 95%|#########5| 20/21 [00:17<00:00,  6.74it/s]
100%|##########| 21/21 [00:17<00:00,  7.24it/s]
100%|##########| 21/21 [00:17<00:00,  1.18it/s]
 kap03/
├── spectra/
├── images/
├── resistics.json
├── masks/
├── results/
│  └── kap160/
│    ├── default/
│    │  └── 0_200000_impedancetensor_default.json
│    └── myconfig/
│      └── 0_200000_impedancetensor_default.json
├── calibrate/
```

(continues on next page)

(continued from previous page)

```
├─ evals/
│  └─ kap160/
│     ├─ default/
│     │  └─ meas01/
│     │     ├─ data.npz
│     │     └─ metadata.json
│     └─ myconfig/
│        └─ meas01/
│           ├─ data.npz
│           └─ metadata.json
├─ features/
└─ time/
   ├─ kap172/
   │  └─ meas01/
   │     ├─ metadata.json
   │     └─ data.npy
   ├─ kap163/
   │  └─ meas01/
   │     ├─ metadata.json
   │     └─ data.npy
   └─ kap160/
      └─ meas01/
         ├─ metadata.json
         └─ data.npy
```

Finally, let's plot our the impedance tensor for this configuration

```
soln = letsgo.get_solution(
    resenv,
    "kap160",
    resenv.config.name,
    0.2,
    resenv.config.tf.name,
    resenv.config.tf.variation,
)
fig = soln.tf.plot(
    soln.freqs,
    soln.components,
    to_plot=["ExHy", "EyHx"],
    x_lim=[1, 5],
    res_lim=[1, 4],
    phs_lim=[0, 100],
    legend="128",
    symbol="circle",
)
fig.update_layout(height=900)
plotly.io.show(fig)
```

Total running time of the script: (0 minutes 19.548 seconds)

Transfer functions

Transfer functions can be customised too depending on needs. There are built-in transfer functions, which have the added benefit of having plotting functions meaning the results can be visualised correctly, for example the impedance tensor.

However, if a completely custom transfer function is required, this can be done with the caveat that there will be no plotting function available. A better solution might be to write a custom transfer function if required. For more about writing custom transfer functions, see the advanced usage.

```
from resistics.transfunc import TransferFunction
```

To initialise a new transfer function, the input and channels need to be defined.

```
tf = TransferFunction(in_chans=["Cat", "Dog"], out_chans=["Tiger", "Wolf"])
print(tf)
tf.summary()
```

```
| Tiger | = | Tiger_Cat   Tiger_Dog   | | Cat   |
| Wolf  |   | Wolf_Cat    Wolf_Dog    | | Dog   |
{
    'name': 'TransferFunction',
    'variation': 'generic',
    'out_chans': ['Tiger', 'Wolf'],
    'in_chans': ['Cat', 'Dog'],
    'cross_chans': ['Cat', 'Dog'],
    'n_out': 2,
    'n_in': 2,
    'n_cross': 2
}
```

It is also possible to set the channels that will be used to calculate out the cross spectra. Note that these channels should be available in the input site, output site and cross site respectively.

```
tf = TransferFunction(
    name="Jungle",
    in_chans=["Cat", "Dog"],
    out_chans=["Tiger", "Wolf"],
    cross_chans=["Lizard", "Crocodile"],
)
print(tf)
tf.summary()
```

```
| Tiger | = | Tiger_Cat   Tiger_Dog   | | Cat   |
| Wolf  |   | Wolf_Cat    Wolf_Dog    | | Dog   |
{
    'name': 'Jungle',
    'variation': 'generic',
    'out_chans': ['Tiger', 'Wolf'],
    'in_chans': ['Cat', 'Dog'],
    'cross_chans': ['Lizard', 'Crocodile'],
    'n_out': 2,
    'n_in': 2,
```

(continues on next page)

(continued from previous page)

```
    'n_cross': 2
}
```

In scenarios where the core transfer function stays the same (input and output channels), but the cross channels will be changed, there is an additional variation property that helps separate them.

```
tf = TransferFunction(
    name="Jungle",
    variation="Birds",
    in_chans=["Cat", "Dog"],
    out_chans=["Tiger", "Wolf"],
    cross_chans=["Owl", "Eagle"],
)
print(tf)
tf.summary()
```

```
| Tiger | = | Tiger_Cat   Tiger_Dog   | | Cat   |
| Wolf  |   | Wolf_Cat    Wolf_Dog    | | Dog   |
{
    'name': 'Jungle',
    'variation': 'Birds',
    'out_chans': ['Tiger', 'Wolf'],
    'in_chans': ['Cat', 'Dog'],
    'cross_chans': ['Owl', 'Eagle'],
    'n_out': 2,
    'n_in': 2,
    'n_cross': 2
}
```

Total running time of the script: (0 minutes 0.010 seconds)

4.2 Custom processes

Writing a custom process coming soon

4.3 resistics package

4.3.1 Submodules

resistics.calibrate module

Functions and classes for instrument and sensor calibration of data

Calibration data should be given in the frequency domain and has a magnitude and phase component (in radians). Calibration data is the impulse response for an instrument or sensor and is usually deconvolved (division in frequency domain) from the time data.

Notes

Calibration data for induction coils is given in mV/nT. Because this is deconvolved from magnetic time data, which is in mV, the resultant magnetic time data is in nT.

pydantic model resistics.calibrate.**CalibrationData**

Bases: *WriteableMetadata*

Class for holding calibration data

Calibration is usually the transfer function of the instrument or sensor to be removed from the data. It is expected to be in the frequency domain.

Regarding units:

- Magnitude units are dependent on use case
- Phase is in radians


```
{
   "title": "CalibrationData",
   "description": "Class for holding calibration data\n\nCalibration is usually the␣
↪transfer function of the instrument or sensor\nto be removed from the data. It is␣
↪expected to be in the frequency domain.\n\nRegarding units:\n\n- Magnitude units␣
↪are dependent on use case\n- Phase is in radians",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "file_path": {
         "title": "File Path",
         "type": "string",
         "format": "path"
      },
      "sensor": {
         "title": "Sensor",
         "default": "",
         "type": "string"
      },
      "serial": {
         "title": "Serial",
         "anyOf": [
            {
               "type": "integer"
            },
            {
               "type": "string"
            }
         ]
      },
      "static_gain": {
         "title": "Static Gain",
         "default": 1,
         "type": "number"
      },
```


(continues on next page)

(continued from previous page)


```
      "magnitude_unit": {
         "title": "Magnitude Unit",
         "default": "mV/nT",
         "type": "string"
      },
      "frequency": {
         "title": "Frequency",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "magnitude": {
         "title": "Magnitude",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "phase": {
         "title": "Phase",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "n_samples": {
         "title": "N Samples",
         "type": "integer"
      }
   },
   "required": [
      "serial",
      "frequency",
      "magnitude",
      "phase"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
```


(continues on next page)

(continued from previous page)

```
            },
            "version": {
               "title": "Version",
               "type": "string"
            }
         }
      }
   }
}
```

field file_path: Path | None = None

Path to the calibration file

field sensor: str = ''

Sensor type

field serial: int | str [Required]

Serial number of the sensor

field static_gain: float = 1

Static gain to apply

field magnitude_unit: str = 'mV/nT'

Units of the magnitude

field frequency: List[float] [Required]

Frequencies in Hz

field magnitude: List[float] [Required]

Magnitude

field phase: List[float] [Required]

Phase

field n_samples: int | None = None

Number of data samples

Validated by

- validate_n_samples

plot(*fig: Figure | None = None*, *color: str = 'blue'*, *legend: str = 'CalibrationData'*) → Figure

Plot calibration data

Parameters

- **fig** (*Optional[go.Figure], optional*) – A figure if adding the calibration data to an existing plot, by default None
- **color** (*str, optional*) – The color for the plot, by default "blue"
- **legend** (*str, optional*) – The legend name, by default "CalibrationData"

Returns

Plotly figure with the calibration data added

Return type

go.Figure

to_dataframe() → DataFrame

Convert to pandas DataFrame

pydantic model resistics.calibrate.**CalibrationReader**

Bases: *ResisticsProcess*

Parent class for reading calibration data


```
{
   "title": "CalibrationReader",
   "description": "Parent class for reading calibration data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "extension": {
         "title": "Extension",
         "type": "string"
      }
   }
}
```


field extension: str | None = None

pydantic model resistics.calibrate.**InstrumentCalibrationReader**

Bases: *CalibrationReader*

Parent class for reading instrument calibration files


```
{
   "title": "InstrumentCalibrationReader",
   "description": "Parent class for reading instrument calibration files",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "extension": {
         "title": "Extension",
         "type": "string"
      }
   }
}
```


run(*metadata:* SpectraMetadata) → *CalibrationData*

pydantic model resistics.calibrate.**SensorCalibrationReader**

Bases: *CalibrationReader*

Parent class for reading sensor calibration files

Use this reader for induction coil calibration file readers

Examples

A short example to show how naming substitution works

```
>>> from pathlib import Path
>>> from resistics.testing import time_metadata_1chan
>>> from resistics.calibrate import SensorCalibrationReader
>>> calibration_path = Path("test")
>>> metadata = time_metadata_1chan()
>>> metadata.chans_metadata["chan1"].sensor = "example"
>>> metadata.chans_metadata["chan1"].serial = "254"
>>> calibrator = SensorCalibrationReader(extension=".json")
>>> calibrator.file_str
'IC_$sensor$extension'
>>> file_path = calibrator._get_path(calibration_path, metadata, "chan1")
>>> file_path.name
'IC_example.json'
```

If the file name has a different pattern, the file_str can be changed as required.

```
>>> calibrator = SensorCalibrationReader(file_str="$sensor_$serial$extension",␣
↪extension=".json")
>>> file_path = calibrator._get_path(calibration_path, metadata, "chan1")
>>> file_path.name
'example_254.json'
```

```
{
   "title": "SensorCalibrationReader",
   "description": "Parent class for reading sensor calibration files\n\nUse this␣
↪reader for induction coil calibration file readers\n\nExamples\n--------\nA short␣
↪example to show how naming substitution works\n\n>>> from pathlib import Path\n>>>
↪ from resistics.testing import time_metadata_1chan\n>>> from resistics.calibrate␣
↪import SensorCalibrationReader\n>>> calibration_path = Path(\"test\")\n>>>␣
↪metadata = time_metadata_1chan()\n>>> metadata.chans_metadata[\"chan1\"].sensor =␣
↪\"example\"\n>>> metadata.chans_metadata[\"chan1\"].serial = \"254\"\n>>>␣
↪calibrator = SensorCalibrationReader(extension=\".json\")\n>>> calibrator.file_
↪str\n'IC_$sensor$extension'\n>>> file_path = calibrator._get_path(calibration_
↪path, metadata, \"chan1\")\n>>> file_path.name\n'IC_example.json'\n\nIf the file␣
↪name has a different pattern, the file_str can be changed as\nrequired.\n\n>>>␣
↪calibrator = SensorCalibrationReader(file_str=\"$sensor_$serial$extension\",␣
↪extension=\".json\")\n>>> file_path = calibrator._get_path(calibration_path,␣
↪metadata, \"chan1\")\n>>> file_path.name\n'example_254.json'",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "extension": {
         "title": "Extension",
         "type": "string"
      },
      "file_str": {
```

(continues on next page)

(continued from previous page)

```
        "title": "File Str",
        "default": "IC_$sensor$extension",
        "type": "string"
    }
  }
}
```

field file_str: str = 'IC_$sensor$extension'

run(*dir_path: Path*, *metadata:* SpectraMetadata, *chan: str*) → *CalibrationData*

Run the calibration file reader

Parameters

- **dir_path** (*Path*) – The directory with calibration files
- **metadata** (SpectraMetadata) – TimeData metadata
- **chan** (*str*) – The channel for which to search for a calibration file

Returns
The calibration data

Return type
CalibrationData

Raises
CalibrationFileNotFound – If the calibration file does not exist

read_calibration_data(*file_path: Path*, *chan_metadata:* ChanMetadata) → *CalibrationData*

Read calibration data from a file

The is implemented as a separate function for anyone interested in reading a calibration file separately from the run function.

Parameters

- **file_path** (*Path*) – The file path of the calibration file
- **chan_metadata** (ChanMetadata) – The channel metadata

Raises
NotImplementedError – To be implemented in child classes

pydantic model resistics.calibrate.**SensorCalibrationJSON**

Bases: *SensorCalibrationReader*

Read in JSON formatted calibration data

```
{
   "title": "SensorCalibrationJSON",
   "description": "Read in JSON formatted calibration data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "extension": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Extension",
            "default": ".json",
            "type": "string"
        },
        "file_str": {
            "title": "File Str",
            "default": "IC_$sensor$extension",
            "type": "string"
        }
    }
}
```

field extension: str = '.json'

read_calibration_data(*file_path: Path*, *chan_metadata:* ChanMetadata) → *CalibrationData*

Read the JSON calibration data

Parameters

- **file_path** (*Path*) – The file path of the JSON calibration file
- **chan_metadata** (ChanMetadata) – The channel metadata. Note that this is not used but is kept here to ensure signature match to the parent class

Returns
The calibration data

Return type
CalibrationData

pydantic model resistics.calibrate.**SensorCalibrationTXT**

Bases: *SensorCalibrationReader*

Read in calibration data from a TXT file

In general, JSON calibration files are recommended as they are more reliable to read in. However, there are cases where it is easier to write out a text based calibration file.

The format of the calibration file should be as follows:

```
Serial = 710
Sensor = LEMI120
Static gain = 1
Magnitude unit = mV/nT
Phase unit = degrees
Chopper = False

CALIBRATION DATA
1.1000E-4       1.000E-2        9.0000E1
1.1000E-3       1.000E-1        9.0000E1
1.1000E-2       1.000E0     8.9000E1
2.1000E-2       1.903E0     8.8583E1
```

See also:

SensorCalibrationJSON
Reader for JSON calibration files

```
{
   "title": "SensorCalibrationTXT",
   "description": "Read in calibration data from a TXT file\n\nIn general, JSON␣
↪calibration files are recommended as they are more reliable\nto read in. However,␣
↪there are cases where it is easier to write out a text\nbased calibration file.\n\
↪nThe format of the calibration file should be as follows:\n\n.. code-block:: text\
↪n\n    Serial = 710\n    Sensor = LEMI120\n    Static gain = 1\n    Magnitude␣
↪unit = mV/nT\n    Phase unit = degrees\n    Chopper = False\n\n    CALIBRATION␣
↪DATA\n    1.1000E-4       1.000E-2        9.0000E1\n    1.1000E-3       1.000E-1 ␣
↪      9.0000E1\n    1.1000E-2       1.000E0     8.9000E1\n    2.1000E-2       1.
↪903E0     8.8583E1\n\nSee Also\n--------\nSensorCalibrationJSON : Reader for JSON␣
↪calibration files",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "extension": {
         "title": "Extension",
         "default": ".TXT",
         "type": "string"
      },
      "file_str": {
         "title": "File Str",
         "default": "IC_$sensor$extension",
         "type": "string"
      }
   }
}
```


read_calibration_data(*file_path: Path*, *chan_metadata:* ChanMetadata) → *CalibrationData*

Read the TXT calibration data

Parameters

- **file_path** (*Path*) – The file path of the JSON calibration file
- **chan_metadata** (ChanMetadata) – The channel metadata. Note that this is not used but is kept here to ensure signature match to the parent class

Returns

The calibration data

Return type

CalibrationData

pydantic model resistics.calibrate.**Calibrator**

Bases: *ResisticsProcess*

Parent class for a calibrator


```
{
   "title": "Calibrator",
   "description": "Parent class for a calibrator",
   "type": "object",
```


(continues on next page)

(continued from previous page)


```
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "chans": {
         "title": "Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      }
   }
}
```


field chans: List[str] | None = None

List of channels to calibrate

run(*dir_path: Path*, *spec_data:* SpectraData) → *SpectraData*

Run the instrument calibration

pydantic model resistics.calibrate.**InstrumentCalibrator**

Bases: *Calibrator*


```
{
   "title": "InstrumentCalibrator",
   "description": "Parent class for a calibrator",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "chans": {
         "title": "Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "readers": {
         "title": "Readers",
         "type": "array",
         "items": {
            "$ref": "#/definitions/InstrumentCalibrationReader"
         }
      }
   },
   "required": [
      "readers"
   ],
   "definitions": {
```


(continues on next page)

(continued from previous page)

```
      "InstrumentCalibrationReader": {
         "title": "InstrumentCalibrationReader",
         "description": "Parent class for reading instrument calibration files",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "extension": {
               "title": "Extension",
               "type": "string"
            }
         }
      }
   }
}
```

field readers: List[*InstrumentCalibrationReader*] **[Required]**

List of readers for reading in instrument calibration files

pydantic model resistics.calibrate.**SensorCalibrator**

Bases: *Calibrator*

```
{
   "title": "SensorCalibrator",
   "description": "Parent class for a calibrator",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "chans": {
         "title": "Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "readers": {
         "title": "Readers",
         "type": "array",
         "items": {
            "$ref": "#/definitions/SensorCalibrationReader"
         }
      }
   },
   "required": [
      "readers"
   ],
   "definitions": {
```

(continues on next page)

(continued from previous page)


```
      "SensorCalibrationReader": {
         "title": "SensorCalibrationReader",
         "description": "Parent class for reading sensor calibration files\n\nUse␣
↪this reader for induction coil calibration file readers\n\nExamples\n--------\nA␣
↪short example to show how naming substitution works\n\n>>> from pathlib import␣
↪Path\n>>> from resistics.testing import time_metadata_1chan\n>>> from resistics.
↪calibrate import SensorCalibrationReader\n>>> calibration_path = Path(\"test\")\n>
↪>> metadata = time_metadata_1chan()\n>>> metadata.chans_metadata[\"chan1\"].
↪sensor = \"example\"\n>>> metadata.chans_metadata[\"chan1\"].serial = \"254\"\n>>>
↪ calibrator = SensorCalibrationReader(extension=\".json\")\n>>> calibrator.file_
↪str\n'IC_$sensor$extension'\n>>> file_path = calibrator._get_path(calibration_
↪path, metadata, \"chan1\")\n>>> file_path.name\n'IC_example.json'\n\nIf the file␣
↪name has a different pattern, the file_str can be changed as\nrequired.\n\n>>>␣
↪calibrator = SensorCalibrationReader(file_str=\"$sensor_$serial$extension\",␣
↪extension=\".json\")\n>>> file_path = calibrator._get_path(calibration_path,␣
↪metadata, \"chan1\")\n>>> file_path.name\n'example_254.json'",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "extension": {
               "title": "Extension",
               "type": "string"
            },
            "file_str": {
               "title": "File Str",
               "default": "IC_$sensor$extension",
               "type": "string"
            }
         }
      }
   }
}
```


field readers: List[*SensorCalibrationReader*] [Required]

List of readers for reading in sensor calibration files

run(*dir_path: Path*, *spec_data:* SpectraData) → *SpectraData*

Calibrate Spectra data

resistics.common module

Common resistics functions and classes used throughout the package

resistics.common.**get_version**() → str

Get the version of resistics

resistics.common.**is_file**(*file_path: Path*) → bool

Check if a path exists and points to a file

Parameters
file_path (*Path*) – The path to check

Returns
True if it exists and is a file, False otherwise

Return type
bool

resistics.common.**assert_file**(*file_path: Path*) → None

Require that a file exists

Parameters
file_path (*Path*) – The path to check

Raises
- **FileNotFoundError** – If the path does not exist
- ***NotFileError*** – If the path is not a file

resistics.common.**is_dir**(*dir_path: Path*) → bool

Check if a path exists and points to a directory

Parameters
dir_path (*Path*) – The path to check

Returns
True if it exists and is a directory, False otherwise

Return type
bool

resistics.common.**assert_dir**(*dir_path: Path*) → None

Require that a path is a directory

Parameters
dir_path (*Path*) – Path to check

Raises
- **FileNotFoundError** – If the path does not exist
- ***NotDirectoryError*** – If the path is not a directory

resistics.common.**dir_contents**(*dir_path: Path*) → Tuple[List[Path], List[Path]]

Get contents of directory

Includes both files and directories

Parameters
dir_path (*Path*) – Parent directory path

Returns
- **dirs** (*list*) – List of directories
- **files** (*list*) – List of files excluding hidden files

Raises
- ***PathNotFoundError*** – Path does not exist
- ***NotDirectoryError*** – Path is not a directory

resistics.common.**dir_files**(*dir_path: Path*) → List[Path]

Get files in directory

Excludes hidden files

Parameters
dir_path (*Path*) – Parent directory path

Returns
files – List of files excluding hidden files

Return type
list

resistics.common.**dir_subdirs**(*dir_path: Path*) → List[Path]

Get subdirectories in directory

Excludes hidden files

Parameters
dir_path (*Path*) – Parent directory path

Returns
dirs – List of subdirectories

Return type
list

resistics.common.**known_chan**(*chan: str*) → bool

Check whether resistics is familiar with a channel name

Parameters
chan (*str*) – The channel name

Returns
True if it is a resistics known channel, false otherwise

Return type
bool

Examples

```
>>> from resistics.common import known_chan
>>> known_chan("Ex")
True
>>> known_chan("Hy")
True
>>> known_chan("cat")
False
```

resistics.common.**is_electric**(*chan: str*) → bool

Check if a channel is electric

Parameters
chan (*str*) – Channel name

Returns
True if channel is electric

Return type
bool

Examples

```
>>> from resistics.common import is_electric
>>> is_electric("Ex")
True
>>> is_electric("Hx")
False
```

resistics.common.**is_magnetic**(*chan: str*) → bool

Check if channel is magnetic

Parameters
chan (*str*) – Channel name

Returns
True if channel is magnetic

Return type
bool

Examples

```
>>> from resistics.common import is_magnetic
>>> is_magnetic("Ex")
False
>>> is_magnetic("Hx")
True
```

resistics.common.**get_chan_type**(*chan: str*) → str

Get the channel type from the channel name

Parameters
chan (*str*) – The name of the channel

Returns
The channel type

Return type
str

Raises
ValueError – If the channel is not known to resistics

Examples

```
>>> from resistics.common import get_chan_type
>>> get_chan_type("Ex")
'electric'
>>> get_chan_type("Hz")
'magnetic'
>>> get_chan_type("abc")
Traceback (most recent call last):
...
ValueError: Channel abc not recognised as either electric or magnetic
```

resistics.common.**check_chan**(*chan: str*, *chans: Collection[str]*) → bool

Check a channel exists and raise a KeyError if not

Parameters

- **chan** (*str*) – The channel to check
- **chans** (*Collection[str]*) – A collection of channels to check against

Returns
True if all checks passed

Return type
bool

Raises
ChannelNotFoundError – If the channel is not found in the channel list

resistics.common.**fs_to_string**(*fs: float*) → str

Convert sampling frequency into a string for filenames

Parameters
fs (*float*) – The sampling frequency

Returns
Sample frequency converted to string for the purposes of a filename

Return type
str

Examples

```
>>> from resistics.common import fs_to_string
>>> fs_to_string(512.0)
'512_000000'
```

resistics.common.**array_to_string**(*data: ndarray*, *sep: str = ', '*, *precision: int = 8*, *scientific: bool = False*) → str

Convert an array to a string for logging or printing

Parameters

- **data** (*np.ndarray*) – The array
- **sep** (*str, optional*) – The separator to use, by default ", "

- **precision** (*int, optional*) – Number of decimal places, by default 8. Ignored for integers.
- **scientific** (*bool, optional*) – Flag for formatting floats as scientific, by default False

Returns
String representation of array

Return type
str

Examples

```
>>> import numpy as np
>>> from resistics.common import array_to_string
>>> data = np.array([1,2,3,4,5])
>>> array_to_string(data)
'1, 2, 3, 4, 5'
>>> data = np.array([1,2,3,4,5], dtype=np.float32)
>>> array_to_string(data)
'1.00000000, 2.00000000, 3.00000000, 4.00000000, 5.00000000'
>>> array_to_string(data, precision=3, scientific=True)
'1.000e+00, 2.000e+00, 3.000e+00, 4.000e+00, 5.000e+00'
```

pydantic model resistics.common.**ResisticsModel**

Bases: BaseModel

Base resistics model

```
{
   "title": "ResisticsModel",
   "description": "Base resistics model",
   "type": "object",
   "properties": {}
}
```

to_string() → str

Class info as string

summary() → None

Print a summary of the class

pydantic model resistics.common.**ResisticsFile**

Bases: *ResisticsModel*

Required information for writing out a resistics file

```
{
   "title": "ResisticsFile",
   "description": "Required information for writing out a resistics file",
   "type": "object",
   "properties": {
      "created_on_local": {
         "title": "Created On Local",
         "type": "string",
```

(continues on next page)

(continued from previous page)


```
            "format": "date-time"
        },
        "created_on_utc": {
            "title": "Created On Utc",
            "type": "string",
            "format": "date-time"
        },
        "version": {
            "title": "Version",
            "type": "string"
        }
    }
}
```


field created_on_local: *datetime* **[Optional]**

field created_on_utc: *datetime* **[Optional]**

field version: *str* **|** *None* **[Optional]**

pydantic model resistics.common.**Metadata**

Bases: *ResisticsModel*

Parent class for metadata


```
{
    "title": "Metadata",
    "description": "Parent class for metadata",
    "type": "object",
    "properties": {}
}
```


pydantic model resistics.common.**WriteableMetadata**

Bases: *Metadata*

Base class for writeable metadata


```
{
    "title": "WriteableMetadata",
    "description": "Base class for writeable metadata",
    "type": "object",
    "properties": {
        "file_info": {
            "$ref": "#/definitions/ResisticsFile"
        }
    },
    "definitions": {
        "ResisticsFile": {
            "title": "ResisticsFile",
            "description": "Required information for writing out a resistics file",
            "type": "object",
            "properties": {
                "created_on_local": {
```


(continues on next page)

(continued from previous page)

```
            "title": "Created On Local",
            "type": "string",
            "format": "date-time"
         },
         "created_on_utc": {
            "title": "Created On Utc",
            "type": "string",
            "format": "date-time"
         },
         "version": {
            "title": "Version",
            "type": "string"
         }
      }
   }
}
```

field file_info: *ResisticsFile* | None = None

Information about a file, relevant if writing out or reading back in

write(*json_path: Path*)

Write out JSON metadata file

Parameters

json_path (*Path*) – Path to write JSON file

pydantic model resistics.common.**Record**

Bases: *ResisticsModel*

Class to hold a record

A record holds information about a process that was run. It is intended to track processes applied to data, allowing a process history to be saved along with any datasets.

Examples

A simple example of creating a process record

```
>>> from resistics.common import Record
>>> messages = ["message 1", "message 2"]
>>> record = Record(
...     creator={"name": "example", "parameter1": 15},
...     messages=messages,
...     record_type="example"
... )
>>> record.summary()
{
    'time_local': '...',
    'time_utc': '...',
    'creator': {'name': 'example', 'parameter1': 15},
    'messages': ['message 1', 'message 2'],
    'record_type': 'example'
}
```

```
{
   "title": "Record",
   "description": "Class to hold a record\n\nA record holds information about a␣
↪process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
   "type": "object",
   "properties": {
      "time_local": {
         "title": "Time Local",
         "type": "string",
         "format": "date-time"
      },
      "time_utc": {
         "title": "Time Utc",
         "type": "string",
         "format": "date-time"
      },
      "creator": {
         "title": "Creator",
         "type": "object"
      },
      "messages": {
         "title": "Messages",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "record_type": {
         "title": "Record Type",
         "type": "string"
      }
   },
   "required": [
      "creator",
      "messages",
      "record_type"
   ]
}
```


field time_local: datetime [Optional]

The local time when the process ran

field time_utc: datetime [Optional]

The UTC time when the process ran

field creator: Dict[str, Any] [Required]

The creator and its parameters as a dictionary

field messages: List[str] [Required]

Any messages in the record

field record_type: str [Required]

The record type

pydantic model resistics.common.**History**

Bases: *ResisticsModel*

Class for storing processing history

Parameters

records (*List[Record], optional*) – List of records, by default []

Examples

```
>>> from resistics.testing import record_example1, record_example2
>>> from resistics.common import History
>>> record1 = record_example1()
>>> record2 = record_example2()
>>> history = History(records=[record1, record2])
>>> history.summary()
{
    'records': [
        {
            'time_local': '...',
            'time_utc': '...',
            'creator': {
                'name': 'example1',
                'a': 5,
                'b': -7.0
            },
            'messages': ['Message 1', 'Message 2'],
            'record_type': 'process'
        },
        {
            'time_local': '...',
            'time_utc': '...',
            'creator': {
                'name': 'example2',
                'a': 'parzen',
                'b': -21
            },
            'messages': ['Message 5', 'Message 6'],
            'record_type': 'process'
        }
    ]
}
```

```
{
   "title": "History",
```

(continues on next page)

(continued from previous page)


```
   "description": "Class for storing processing history\n\nParameters\n----------\
↪nrecords : List[Record], optional\n    List of records, by default []\n\nExamples\
↪n--------\n>>> from resistics.testing import record_example1, record_example2\n>>>
↪ from resistics.common import History\n>>> record1 = record_example1()\n>>>␣
↪record2 = record_example2()\n>>> history = History(records=[record1, record2])\n>>
↪> history.summary()\n{\n    'records': [\n        {\n            'time_local': '..
↪.',\n            'time_utc': '...',\n            'creator': {\n
↪'name': 'example1',\n                'a': 5,\n                'b': -7.0\n        ␣
↪    },\n            'messages': ['Message 1', 'Message 2'],\n            'record_
↪type': 'process'\n        },\n        {\n            'time_local': '...',\n      ␣
↪     'time_utc': '...',\n            'creator': {\n                'name':
↪'example2',\n                'a': 'parzen',\n                'b': -21\n          ␣
↪ },\n            'messages': ['Message 5', 'Message 6'],\n            'record_type
↪': 'process'\n        }\n    ]\n}",
   "type": "object",
   "properties": {
      "records": {
         "title": "Records",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/Record"
         }
      }
   },
   "definitions": {
      "Record": {
         "title": "Record",
         "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...     ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
         "type": "object",
         "properties": {
            "time_local": {
               "title": "Time Local",
               "type": "string",
               "format": "date-time"
            },
            "time_utc": {
               "title": "Time Utc",
               "type": "string",
               "format": "date-time"
            },
            "creator": {
               "title": "Creator",
```


(continues on next page)

(continued from previous page)

```
            "type": "object"
         },
         "messages": {
            "title": "Messages",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "record_type": {
            "title": "Record Type",
            "type": "string"
         }
      },
      "required": [
         "creator",
         "messages",
         "record_type"
      ]
   }
}
```

field records: List[*Record*] = []

add_record(*record:* Record)

Add a process record to the list

Parameters

record (Record) – The record to add

resistics.common.**get_record**(*creator: Dict[str, Any]*, *messages: str | List[str]*, *record_type: str = 'process'*, *time_utc: datetime | None = None*, *time_local: datetime | None = None*) → *Record*

Get a process record

Parameters

- **creator** (*Dict[str, Any]*) – The creator and its parameters as a dictionary
- **messages** (*Union[str, List[str]]*) – The messages as either a single str or a list of strings
- **record_type** (*str, optional*) – The type of record, by default “process”
- **time_utc** (*Optional[datetime], optional*) – UTC time to attach to the record, by default None. If None, will default to UTC now
- **time_local** (*Optional[datetime], optional*) – Local time to attach to the record, by default None. If None, will defult to local now

Returns

The process record

Return type

Record

Examples

```
>>> from resistics.common import get_record
>>> record = get_record(
...     creator={"name": "example", "a": 5, "b": -7.0},
...     messages="a message"
... )
>>> record.creator
{'name': 'example', 'a': 5, 'b': -7.0}
>>> record.messages
['a message']
>>> record.record_type
'process'
>>> record.time_utc
datetime.datetime(...)
>>> record.time_local
datetime.datetime(...)
```

resistics.common.**get_history**(*record:* Record, *history:* History | *None* = *None*) → *History*

Get a new History instance or add a record to a copy of an existing one

This method always makes a deepcopy of an input history to avoid any unplanned modifications to the inputs.

Parameters

- **record** (Record) – The record
- **history** (*Optional[History], optional*) – A history to add to, by default None

Returns

History with the record added

Return type

History

Examples

Get a new History with a single Record

```
>>> from resistics.common import get_history
>>> from resistics.testing import record_example1, record_example2
>>> record1 = record_example1()
>>> history = get_history(record1)
>>> history.summary()
{
    'records': [
        {
            'time_local': '...',
            'time_utc': '...',
            'creator': {
                'name': 'example1',
                'a': 5,
                'b': -7.0
            },
            'messages': ['Message 1', 'Message 2'],
```

(continues on next page)

(continued from previous page)

```
            'record_type': 'process'
        }
    ]
}
```

Alternatively, add to an existing History. This will make a copy of the original history. If a copy is not needed, the add_record method of history can be used.

```
>>> record2 = record_example2()
>>> history = get_history(record2, history)
>>> history.summary()
{
    'records': [
        {
            'time_local': '...',
            'time_utc': '...',
            'creator': {
                'name': 'example1',
                'a': 5,
                'b': -7.0
            },
            'messages': ['Message 1', 'Message 2'],
            'record_type': 'process'
        },
        {
            'time_local': '...',
            'time_utc': '...',
            'creator': {
                'name': 'example2',
                'a': 'parzen',
                'b': -21
            },
            'messages': ['Message 5', 'Message 6'],
            'record_type': 'process'
        }
    ]
}
```

pydantic model resistics.common.**ResisticsProcess**

Bases: *ResisticsModel*

Base class for resistics processes

Resistics processes perform operations on data (including read and write operations). Each time a ResisticsProcess child class is run, it should add a process record to the dataset

```
{
   "title": "ResisticsProcess",
   "description": "Base class for resistics processes\n\nResistics processes␣
↪perform operations on data (including read and write\noperations). Each time a␣
↪ResisticsProcess child class is run, it should add\na process record to the␣
↪dataset",
   "type": "object",
```

(continues on next page)

(continued from previous page)

```
    "properties": {
        "name": {
            "title": "Name",
            "type": "string"
        }
    }
}
```

field name: `str | None = None`

Validated by

- validate_name

classmethod validate(*value:* ResisticsProcess | *Dict[str, Any]*) → *ResisticsProcess*

Validate a ResisticsProcess in another pydantic class

Parameters
value (*Union[ResisticsProcess, Dict[str, Any]]*) – A ResisticsProcess child class or a dictionary

Returns
A ResisticsProcess child class

Return type
ResisticsProcess

Raises

- **ValueError** – If the value is neither a ResisticsProcess or a dictionary
- **KeyError** – If name is not in the dictionary
- **ValueError** – If initialising from dictionary fails

Examples

The following example will show how a generic ResisticsProcess child class can be instantiated from ResisticsProcess using a dictionary, which might be read in from a JSON configuration file.

```
>>> from resistics.common import ResisticsProcess
>>> from resistics.decimate import DecimationSetup
>>> process = {"name": 'DecimationSetup', "n_levels": 8, "per_level": 5, "min_
↪samples": 256, "div_factor": 2, "eval_freqs": None}
>>> ResisticsProcess(**process)
ResisticsProcess(name='DecimationSetup')
```

This is not what was expected. To get the right result, the class validate method needs to be used. This is done automatically by pydantic.

```
>>> ResisticsProcess.validate(process)
DecimationSetup(name='DecimationSetup', n_levels=8, per_level=5, min_
↪samples=256, div_factor=2, eval_freqs=None)
```

That's better. Note that errors will be raised if the dictionary is not formatted as expected.

```
>>> process = {"n_levels": 8, "per_level": 5, "min_samples": 256, "div_factor":␣
↪2, "eval_freqs": None}
>>> ResisticsProcess.validate(process)
Traceback (most recent call last):
...
KeyError: 'No name provided for initialisation of process'
```

This functionality is most useful in the resistics configurations which can be saved as JSON files. The default configuration uses the default parameterisation of DecimationSetup.

```
>>> from resistics.letsgo import Configuration
>>> config = Configuration(name="example1")
>>> config.dec_setup
DecimationSetup(name='DecimationSetup', n_levels=8, per_level=5, min_
↪samples=256, div_factor=2, eval_freqs=None)
```

Now create another configuration with a different setup by passing a dictionary. In practise, this dictionary will most likely be read in from a configuration file.

```
>>> setup = DecimationSetup(n_levels=4, per_level=3)
>>> test_dict = setup.dict()
>>> test_dict
{'name': 'DecimationSetup', 'n_levels': 4, 'per_level': 3, 'min_samples': 256,
↪'div_factor': 2, 'eval_freqs': None}
>>> config2 = Configuration(name="example2", dec_setup=test_dict)
>>> config2.dec_setup
DecimationSetup(name='DecimationSetup', n_levels=4, per_level=3, min_
↪samples=256, div_factor=2, eval_freqs=None)
```

This method allows the saving of a configuration with custom processors in a JSON file which can be loaded and used again.

parameters() → Dict[str, Any]

Return any process parameters incuding the process name

These parameters are expected to be primatives and should be sufficient to reinitialise the process and re-run the data. The base class assumes all class variables meet this description.

Returns
Dictionary of parameters

Return type
Dict[str, Any]

class resistics.common.**ResisticsBase**

Bases: object

Resistics base class

Parent class to ensure consistency of common methods

type_to_string() → str

Get the class type as a string

to_string() → str

Class details as a string

summary(*symbol: str = '-'*) → None

Print a summary of class details

class resistics.common.**ResisticsData**

Bases: *ResisticsBase*

Base class for a resistics data object

pydantic model resistics.common.**ResisticsWriter**

Bases: *ResisticsProcess*

Parent process for data writers

Parameters

overwrite (*bool, optional*) – Boolean flag for overwriting the existing data, by default False


```
{
   "title": "ResisticsWriter",
   "description": "Parent process for data writers\n\nParameters\n----------\
↪noverwrite : bool, optional\n    Boolean flag for overwriting the existing data,␣
↪by default False",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "overwrite": {
         "title": "Overwrite",
         "default": true,
         "type": "boolean"
      }
   }
}
```


field overwrite: bool = True

run(*dir_path: Path*, *data:* ResisticsData) → None

Write out a ResisticsData child object to a directory

resistics.config module

Module containing the resistics configuration

The configuration is an essential part of a resistics environment. It defines many dependencies, such as which data readers to use for time series data or calibration data and processing options.

Configuration allows users to insert their own dependencies and processors to work with data.

Configurations can be saved to and loaded from JSON files.

pydantic model resistics.config.**Configuration**

Bases: *ResisticsModel*

The resistics configuration

Configuration can be customised by users who wish to use their own custom processes for certain steps. In most cases, customisation will be for:

- Implementing new time data readers
- Implementing readers for specific calibration formats
- Implementing time data processors
- Implementing spectra data processors
- Adding new features to extract from the data

Examples

Frequently, configuration will be used to change data readers.

```
>>> from resistics.letsgo import get_default_configuration
>>> config = get_default_configuration()
>>> config.name
'default'
>>> for tr in config.time_readers:
...     tr.summary()
{
    'name': 'TimeReaderAscii',
    'apply_scalings': True,
    'extension': '.txt',
    'delimiter': None,
    'n_header': 0
}
{
    'name': 'TimeReaderNumpy',
    'apply_scalings': True,
    'extension': '.npy'
}
>>> config.sensor_calibrator.summary()
{
    'name': 'SensorCalibrator',
    'chans': None,
    'readers': [
        {
            'name': 'SensorCalibrationJSON',
            'extension': '.json',
            'file_str': 'IC_$sensor$extension'
        }
    ]
}
```

To change these, it's best to make a new configuration with a different name

```
>>> from resistics.letsgo import Configuration
>>> from resistics.time import TimeReaderNumpy
>>> config = Configuration(name="myconfig", time_readers=[TimeReaderNumpy(apply_
↪scalings=False)])
>>> for tr in config.time_readers:
...     tr.summary()
{
    'name': 'TimeReaderNumpy',
```

(continues on next page)

(continued from previous page)

```
    'apply_scalings': False,
    'extension': '.npy'
}
```

Or for the sensor calibration

```
>>> from resistics.calibrate import SensorCalibrator, SensorCalibrationTXT
>>> calibration_reader = SensorCalibrationTXT(file_str="lemi120_IC_$serial$extension
↪")
>>> calibrator = SensorCalibrator(chans=["Hx", "Hy", "Hz"], readers=[calibration_
↪reader])
>>> config = Configuration(name="myconfig", sensor_calibrator=calibrator)
>>> config.sensor_calibrator.summary()
{
    'name': 'SensorCalibrator',
    'chans': ['Hx', 'Hy', 'Hz'],
    'readers': [
        {
            'name': 'SensorCalibrationTXT',
            'extension': '.TXT',
            'file_str': 'lemi120_IC_$serial$extension'
        }
    ]
}
```

As a final example, create a configuration which used targetted windowing instead of specified window sizes

```
>>> from resistics.letsgo import Configuration
>>> from resistics.window import WindowerTarget
>>> config = Configuration(name="window_target",␣
↪windower=WindowerTarget(target=500))
>>> config.name
'window_target'
>>> config.windower.summary()
{
    'name': 'WindowerTarget',
    'target': 500,
    'min_size': 64,
    'olap_proportion': 0.25
}
```

```
{
   "title": "Configuration",
   "description": "The resistics configuration\n\nConfiguration can be customised␣
↪by users who wish to use their own custom\nprocesses for certain steps. In most␣
↪cases, customisation will be for:\n\n- Implementing new time data readers\n-␣
↪Implementing readers for specific calibration formats\n- Implementing time data␣
↪processors\n- Implementing spectra data processors\n- Adding new features to␣
↪extract from the data\n\nExamples\n--------\nFrequently, configuration will be␣
↪used to change data readers.\n\n>>> from resistics.letsgo import get_default_
↪configuration\n>>> config = get_default_configuration()\n>>> config.name\n'default
↪'\n>>> for tr in config.time_readers:\n...     tr.summary()\n{\n    'name':
```

(continues on next page)

(continued from previous page)

```
↪'TimeReaderAscii',\n    'apply_scalings': True,\n    'extension': '.txt',\n
↪'delimiter': None,\n    'n_header': 0\n}\n{\n    'name': 'TimeReaderNumpy',\n
↪'apply_scalings': True,\n    'extension': '.npy'\n}\n>>> config.sensor_calibrator.
↪summary()\n{\n    'name': 'SensorCalibrator',\n    'chans': None,\n    'readers':␣
↪[\n        {\n            'name': 'SensorCalibrationJSON',\n            'extension
↪': '.json',\n            'file_str': 'IC_$sensor$extension'\n        }\n    ]\n}\
↪n\nTo change these, it's best to make a new configuration with a different name\n\
↪n>>> from resistics.letsgo import Configuration\n>>> from resistics.time import␣
↪TimeReaderNumpy\n>>> config = Configuration(name=\"myconfig\", time_
↪readers=[TimeReaderNumpy(apply_scalings=False)])\n>>> for tr in config.time_
↪readers:\n...     tr.summary()\n{\n    'name': 'TimeReaderNumpy',\n    'apply_
↪scalings': False,\n    'extension': '.npy'\n}\n\nOr for the sensor calibration\n\
↪n>>> from resistics.calibrate import SensorCalibrator, SensorCalibrationTXT\n>>>␣
↪calibration_reader = SensorCalibrationTXT(file_str=\"lemi120_IC_$serial$extension\
↪")\n>>> calibrator = SensorCalibrator(chans=[\"Hx\", \"Hy\", \"Hz\"],␣
↪readers=[calibration_reader])\n>>> config = Configuration(name=\"myconfig\",␣
↪sensor_calibrator=calibrator)\n>>> config.sensor_calibrator.summary()\n{\n
↪'name': 'SensorCalibrator',\n    'chans': ['Hx', 'Hy', 'Hz'],\n    'readers': [\n␣
↪       {\n            'name': 'SensorCalibrationTXT',\n            'extension': '.
↪TXT',\n            'file_str': 'lemi120_IC_$serial$extension'\n        }\n    ]\n}
↪\n\nAs a final example, create a configuration which used targetted windowing\
↪ninstead of specified window sizes\n\n>>> from resistics.letsgo import␣
↪Configuration\n>>> from resistics.window import WindowerTarget\n>>> config =␣
↪Configuration(name=\"window_target\", windower=WindowerTarget(target=500))\n>>>␣
↪config.name\n'window_target'\n>>> config.windower.summary()\n{\n    'name':
↪'WindowerTarget',\n    'target': 500,\n    'min_size': 64,\n    'olap_proportion
↪': 0.25\n}",
  "type": "object",
  "properties": {
     "name": {
        "title": "Name",
        "type": "string"
     },
     "time_readers": {
        "title": "Time Readers",
        "default": [
           {
              "name": "TimeReaderAscii",
              "apply_scalings": true,
              "extension": ".txt",
              "delimiter": null,
              "n_header": 0
           },
           {
              "name": "TimeReaderNumpy",
              "apply_scalings": true,
              "extension": ".npy"
           }
        ],
        "type": "array",
        "items": {
           "$ref": "#/definitions/TimeReader"
```

(continues on next page)

(continued from previous page)

```
            }
        },
        "time_processors": {
            "title": "Time Processors",
            "default": [
                {
                    "name": "InterpolateNans"
                },
                {
                    "name": "RemoveMean"
                }
            ],
            "type": "array",
            "items": {
                "$ref": "#/definitions/TimeProcess"
            }
        },
        "dec_setup": {
            "title": "Dec Setup",
            "default": {
                "name": "DecimationSetup",
                "n_levels": 8,
                "per_level": 5,
                "min_samples": 256,
                "div_factor": 2,
                "eval_freqs": null
            },
            "allOf": [
                {
                    "$ref": "#/definitions/DecimationSetup"
                }
            ]
        },
        "decimator": {
            "title": "Decimator",
            "default": {
                "name": "Decimator",
                "resample": true,
                "max_single_factor": 3
            },
            "allOf": [
                {
                    "$ref": "#/definitions/Decimator"
                }
            ]
        },
        "win_setup": {
            "title": "Win Setup",
            "default": {
                "name": "WindowSetup",
                "min_size": 128,
                "min_olap": 32,
```

(continues on next page)

(continued from previous page)


```
            "win_factor": 4,
            "olap_proportion": 0.25,
            "min_n_wins": 5,
            "win_sizes": null,
            "olap_sizes": null
         },
         "allOf": [
            {
               "$ref": "#/definitions/WindowSetup"
            }
         ]
      },
      "windower": {
         "title": "Windower",
         "default": {
            "name": "Windower"
         },
         "allOf": [
            {
               "$ref": "#/definitions/Windower"
            }
         ]
      },
      "fourier": {
         "title": "Fourier",
         "default": {
            "name": "FourierTransform",
            "win_fnc": [
               "kaiser",
               14
            ],
            "detrend": "linear",
            "workers": -2
         },
         "allOf": [
            {
               "$ref": "#/definitions/FourierTransform"
            }
         ]
      },
      "spectra_processors": {
         "title": "Spectra Processors",
         "default": [],
         "type": "array",
         "items": {
            "$ref": "#/definitions/SpectraProcess"
         }
      },
      "evals": {
         "title": "Evals",
         "default": {
            "name": "EvaluationFreqs"
```


(continues on next page)

(continued from previous page)

```
        },
        "allOf": [
          {
            "$ref": "#/definitions/EvaluationFreqs"
          }
        ]
      },
      "sensor_calibrator": {
        "title": "Sensor Calibrator",
        "default": {
          "name": "SensorCalibrator",
          "chans": null,
          "readers": [
            {
              "name": "SensorCalibrationJSON",
              "extension": ".json",
              "file_str": "IC_$sensor$extension"
            }
          ]
        },
        "allOf": [
          {
            "$ref": "#/definitions/Calibrator"
          }
        ]
      },
      "tf": {
        "title": "Tf",
        "default": {
          "name": "ImpedanceTensor",
          "variation": "default",
          "out_chans": [
            "Ex",
            "Ey"
          ],
          "in_chans": [
            "Hx",
            "Hy"
          ],
          "cross_chans": [
            "Hx",
            "Hy"
          ],
          "n_out": 2,
          "n_in": 2,
          "n_cross": 2
        },
        "allOf": [
          {
            "$ref": "#/definitions/TransferFunction"
          }
        ]
```

(continues on next page)

(continued from previous page)


```
      },
      "regression_preparer": {
         "title": "Regression Preparer",
         "default": {
            "name": "RegressionPreparerGathered"
         },
         "allOf": [
            {
               "$ref": "#/definitions/RegressionPreparerGathered"
            }
         ]
      },
      "solver": {
         "title": "Solver",
         "default": {
            "name": "SolverScikitTheilSen",
            "fit_intercept": false,
            "normalize": false,
            "n_jobs": -2,
            "max_subpopulation": 2000,
            "n_subsamples": null
         },
         "allOf": [
            {
               "$ref": "#/definitions/Solver"
            }
         ]
      }
   },
   "required": [
      "name"
   ],
   "definitions": {
      "TimeReader": {
         "title": "TimeReader",
         "description": "Base class for resistics processes\n\nResistics processes␣
↪perform operations on data (including read and write\noperations). Each time a␣
↪ResisticsProcess child class is run, it should add\na process record to the␣
↪dataset",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "apply_scalings": {
               "title": "Apply Scalings",
               "default": true,
               "type": "boolean"
            },
            "extension": {
               "title": "Extension",
```


(continues on next page)

(continued from previous page)


```
            "type": "string"
         }
      }
   },
   "TimeProcess": {
      "title": "TimeProcess",
      "description": "Parent class for processing time data",
      "type": "object",
      "properties": {
         "name": {
            "title": "Name",
            "type": "string"
         }
      }
   },
   "DecimationSetup": {
      "title": "DecimationSetup",
      "description": "Process to calculate decimation parameters\n\nParameters\n-
↪---------\nn_levels : int, optional\n    Number of decimation levels, by default␣
↪8\nper_level : int, optional\n    Number of frequencies per level, by default 5\
↪nmin_samples : int, optional\n    Number of samples under which to quit␣
↪decimating\ndiv_factor : int, optional\n    Minimum division factor for␣
↪decimation, by default 2.\neval_freqs : Optional[List[float]], optional\n    ␣
↪Explicit definition of evaluation frequencies as a flat list, by\n    default␣
↪None. Must be of size n_levels * per_level\n\nExamples\n--------\n>>> from␣
↪resistics.decimate import DecimationSetup\n>>> dec_setup = DecimationSetup(n_
↪levels=3, per_level=2)\n>>> dec_params = dec_setup.run(128)\n>>> print(dec_params.
↪to_dataframe())\n                     0          1     fs  factors  increments\
↪ndecimation level\n0                 32.0  22.627417  128.0        1           1\
↪n1                16.0  11.313708   64.0        2           2\n2                ␣
↪ 8.0   5.656854   32.0        4           2",
      "type": "object",
      "properties": {
         "name": {
            "title": "Name",
            "type": "string"
         },
         "n_levels": {
            "title": "N Levels",
            "default": 8,
            "type": "integer"
         },
         "per_level": {
            "title": "Per Level",
            "default": 5,
            "type": "integer"
         },
         "min_samples": {
            "title": "Min Samples",
            "default": 256,
            "type": "integer"
         },
```


(continues on next page)

(continued from previous page)

```
            "div_factor": {
               "title": "Div Factor",
               "default": 2,
               "type": "integer"
            },
            "eval_freqs": {
               "title": "Eval Freqs",
               "type": "array",
               "items": {
                  "type": "number"
               }
            }
         }
      },
      "Decimator": {
         "title": "Decimator",
         "description": "Decimate the time data into multiple levels\n\nThere are␣
↪two options for decimation, using time data Resample or using\ntime data Decimate.
↪ The default is to use Resample.",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "resample": {
               "title": "Resample",
               "default": true,
               "type": "boolean"
            },
            "max_single_factor": {
               "title": "Max Single Factor",
               "default": 3,
               "minimum": 3,
               "type": "integer"
            }
         }
      },
      "WindowSetup": {
         "title": "WindowSetup",
         "description": "Setup WindowParameters\n\nWindowSetup outputs the␣
↪WindowParameters to use for windowing decimated\ntime data.\n\nWindow parameters␣
↪are simply the window and overlap sizes for each\ndecimation level.\n\nParameters\
↪n----------\nmin_size : int, optional\n    Minimum window size, by default 128\
↪nmin_olap : int, optional\n    Minimum overlap size, by default 32\nwin_factor :␣
↪int, optional\n    Window factor, by default 4. Window sizes are calculated by␣
↪sampling\n    frequency / 4 to ensure sufficient frequency resolution. If the\n  ␣
↪ sampling frequency is small, window size will be adjusted to\n    min_size\nolap_
↪proportion : float, optional\n    The proportion of the window size to use as the␣
↪overlap, by default\n    0.25. For example, for a window size of 128, the overlap␣
↪would be\n    0.25 * 128 = 32\nmin_n_wins : int, optional\n    The minimum number␣
↪of windows needed in a decimation level, by\n    default 5\nwin_sizes :␣
```

(continues on next page)

(continued from previous page)

```
↪Optional[List[int]], optional\n    Explicit define window sizes, by default None.␣
↪Must have the same\n    length as number of decimation levels\nolap_sizes :␣
↪Optional[List[int]], optional\n    Explicitly define overlap sizes, by default␣
↪None. Must have the same\n    length as number of decimation levels\n\nExamples\n-
↪-------\nGenerate decimation and windowing parameters for data sampled at 0.05 Hz␣
↪or\n20 seconds sampling period\n\n>>> from resistics.decimate import␣
↪DecimationSetup\n>>> from resistics.window import WindowSetup\n>>> dec_params =␣
↪DecimationSetup(n_levels=3, per_level=3).run(0.05)\n>>> dec_params.summary()\n{\n␣
↪   'fs': 0.05,\n    'n_levels': 3,\n    'per_level': 3,\n    'min_samples': 256,\
↪n    'eval_freqs': [\n        0.0125,\n        0.008838834764831844,\n        0.
↪00625,\n        0.004419417382415922,\n        0.003125,\n        0.
↪002209708691207961,\n        0.0015625,\n        0.0011048543456039805,\n       ␣
↪0.00078125\n    ],\n    'dec_factors': [1, 2, 8],\n    'dec_increments': [1, 2,␣
↪4],\n    'dec_fs': [0.05, 0.025, 0.00625]\n}\n>>> win_params = WindowSetup().
↪run(dec_params.n_levels, dec_params.dec_fs)\n>>> win_params.summary()\n{\n    'n_
↪levels': 3,\n    'min_n_wins': 5,\n    'win_sizes': [128, 128, 128],\n    'olap_
↪sizes': [32, 32, 32]\n}\n\nWindow parameters can also be explicitly defined\n\n>>>
↪ from resistics.decimate import DecimationSetup\n>>> from resistics.window import␣
↪WindowSetup\n>>> dec_setup = DecimationSetup(n_levels=3, per_level=3)\n>>> dec_
↪params = dec_setup.run(0.05)\n>>> win_setup = WindowSetup(win_sizes=[1000, 578,␣
↪104])\n>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)\n>>>
↪ win_params.summary()\n{\n    'n_levels': 3,\n    'min_n_wins': 5,\n    'win_sizes
↪': [1000, 578, 104],\n    'olap_sizes': [250, 144, 32]\n}\n\nWhen providing␣
↪explicit window and overlap sizes, the size of the overlap\nis expected to be␣
↪less than the size of the window\n\n>>> from resistics.decimate import␣
↪DecimationSetup\n>>> from resistics.window import WindowSetup\n>>> dec_setup =␣
↪DecimationSetup(n_levels=3, per_level=3)\n>>> dec_params = dec_setup.run(0.05)\n>>
↪> win_setup = WindowSetup(win_sizes=[1000, 578, 104], olap_sizes=[1001, 600, 25])\
↪n>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)\
↪nTraceback (most recent call last):\n...\nValueError: Invalid overlaps found [␣
↪True  True False]",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "min_size": {
              "title": "Min Size",
              "default": 128,
              "type": "integer"
           },
           "min_olap": {
              "title": "Min Olap",
              "default": 32,
              "type": "integer"
           },
           "win_factor": {
              "title": "Win Factor",
              "default": 4,
              "type": "integer"
           },
```

(continues on next page)

(continued from previous page)

```
            "olap_proportion": {
               "title": "Olap Proportion",
               "default": 0.25,
               "type": "number"
            },
            "min_n_wins": {
               "title": "Min N Wins",
               "default": 5,
               "type": "integer"
            },
            "win_sizes": {
               "title": "Win Sizes",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "olap_sizes": {
               "title": "Olap Sizes",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            }
         }
      },
      "Windower": {
         "title": "Windower",
         "description": "Windows DecimatedData\n\nThis is the primary window making␣
↪process for resistics and should be used\nwhen alignment of windows with a site␣
↪or across sites is required.\n\nThis method uses numpy striding to produce window␣
↪views into the decimated\ndata.\n\nSee Also\n--------\nWindowerTarget : A␣
↪windower to make a target number of windows\n\nExamples\n--------\nThe Windower␣
↪windows a DecimatedData object given a reference time and some\nwindow parameters.
↪\n\nThere's quite a few imports needed for this example. Begin by doing the\
↪nimports, defining a reference time and generating random decimated data.\n\n>>>␣
↪from resistics.sampling import to_datetime\n>>> from resistics.testing import␣
↪decimated_data_linear\n>>> from resistics.window import WindowSetup, Windower\n>>>
↪ dec_data = decimated_data_linear(fs=128)\n>>> ref_time = dec_data.metadata.first_
↪time\n>>> print(dec_data.to_string())\n<class 'resistics.decimate.DecimatedData'>\
↪n           fs        dt  n_samples           first_time                        ␣
↪last_time\nlevel\n0      2048.0  0.000488      16384  2021-01-01 00:00:00  2021-
↪01-01 00:00:07.99951171875\n1       512.0  0.001953       4096  2021-01-01␣
↪00:00:00    2021-01-01 00:00:07.998046875\n2       128.0  0.007812       1024 ␣
↪2021-01-01 00:00:00      2021-01-01 00:00:07.9921875\n\nNext, initialise the␣
↪window parameters. For this example, use small windows,\nwhich will make␣
↪inspecting them easier.\n\n>>> win_params = WindowSetup(win_sizes=[16,16,16], min_
↪olap=4).run(dec_data.metadata.n_levels, dec_data.metadata.fs)\n>>> win_params.
↪summary()\n{\n    'n_levels': 3,\n    'min_n_wins': 5,\n    'win_sizes': [16, 16,␣
↪16],\n    'olap_sizes': [4, 4, 4]\n}\n\nPerform the windowing. This actually␣
↪creates views into the decimated data\nusing the numpy.lib.stride_tricks.sliding_
↪window_view function. The shape\nfor a data array at a decimation level is: n_
```

(continues on next page)

(continued from previous page)

```
↪wins x n_chans x win_size. The\ninformation about each level is also in the␣
↪levels_metadata attribute of\nWindowedMetadata.\n\n>>> win_data = Windower().
↪run(ref_time, win_params, dec_data)\n>>> win_data.data[0].shape\n(1365, 2, 16)\n>>
↪> for level_metadata in win_data.metadata.levels_metadata:\n...     level_
↪metadata.summary()\n{\n    'fs': 2048.0,\n    'n_wins': 1365,\n    'win_size': 16,
↪\n    'olap_size': 4,\n    'index_offset': 0\n}\n{\n    'fs': 512.0,\n    'n_wins
↪': 341,\n    'win_size': 16,\n    'olap_size': 4,\n    'index_offset': 0\n}\n{\n ␣
↪  'fs': 128.0,\n    'n_wins': 85,\n    'win_size': 16,\n    'olap_size': 4,\n
↪'index_offset': 0\n}\n\nLet's look at an example of data from the first␣
↪decimation level for the\nfirst channel. This is simply a linear set of data␣
↪ranging from 0...16_383.\n\n>>> dec_data.data[0][0]\narray([    0,     1,     2, .
↪.., 16381, 16382, 16383])\n\nInspecting the first few windows shows they are as␣
↪expected including the\noverlap.\n\n>>> win_data.data[0][0, 0]\narray([ 0,  1,  2,
↪  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])\n>>> win_data.data[0][1, 0]\
↪narray([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27])\n>>> win_
↪data.data[0][2, 0]\narray([24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
↪ 38, 39])",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           }
        }
     },
     "FourierTransform": {
        "title": "FourierTransform",
        "description": "Perform a Fourier transform of the windowed data\n\nThe␣
↪processor is inspired by the scipy.signal.stft function which performs\na similar␣
↪process and involves a Fourier transform along the last axis of\nthe windowed␣
↪data.\n\nParameters\n----------\nwin_fnc : Union[str, Tuple[str, float]]\n    The␣
↪window to use before performing the FFT, by default (\"kaiser\", 14)\ndetrend :␣
↪Union[str, None]\n    Type of detrending to apply before performing FFT, by␣
↪default linear\n    detrend. Setting to None will not apply any detrending to the␣
↪data prior\n    to the FFT\nworkers : int\n    The number of CPUs to use, by␣
↪default max - 2\n\nExamples\n--------\nThis example will get periodic decimated␣
↪data, perfrom windowing and run the\nFourier transform on the windowed data.\n\n..
↪ plot::\n    :width: 90%\n\n    >>> import matplotlib.pyplot as plt\n    >>>␣
↪import numpy as np\n    >>> from resistics.testing import decimated_data_periodic\
↪n    >>> from resistics.window import WindowSetup, Windower\n    >>> from␣
↪resistics.spectra import FourierTransform\n    >>> frequencies = {\"chan1\": [870,
↪ 590, 110, 32, 12], \"chan2\": [480, 375, 210, 60, 45]}\n    >>> dec_data =␣
↪decimated_data_periodic(frequencies, fs=128)\n    >>> dec_data.metadata.chans\n  ␣
↪ ['chan1', 'chan2']\n    >>> print(dec_data.to_string())\n    <class 'resistics.
↪decimate.DecimatedData'>\n                fs        dt  n_samples           first_
↪time                      last_time\n    level\n    0      2048.0  0.000488    ␣
↪ 16384  2021-01-01 00:00:00  2021-01-01 00:00:07.99951171875\n    1       512.0 ␣
↪0.001953       4096  2021-01-01 00:00:00    2021-01-01 00:00:07.998046875\n    2 ␣
↪     128.0  0.007812       1024  2021-01-01 00:00:00      2021-01-01 00:00:07.
↪9921875\n\n    Perform the windowing\n\n    >>> win_params = WindowSetup().
↪run(dec_data.metadata.n_levels, dec_data.metadata.fs)\n    >>> win_data =␣
↪Windower().run(dec_data.metadata.first_time, win_params, dec_data)\n\n    And␣
```

(continues on next page)

(continued from previous page)

```
↪then the Fourier transform. By default, the data will be (linearly)\n    ␣
↪detrended and mutliplied by a Kaiser window prior to the Fourier\n    transform\n\
↪n    >>> spec_data = FourierTransform().run(win_data)\n\n    For plotting of␣
↪magnitude, let's stack the spectra\n\n    >>> freqs_0 = spec_data.metadata.levels_
↪metadata[0].freqs\n    >>> data_0 = np.absolute(spec_data.data[0]).mean(axis=0)\n␣
↪   >>> freqs_1 = spec_data.metadata.levels_metadata[1].freqs\n    >>> data_1 = np.
↪absolute(spec_data.data[1]).mean(axis=0)\n    >>> freqs_2 = spec_data.metadata.
↪levels_metadata[2].freqs\n    >>> data_2 = np.absolute(spec_data.data[2]).
↪mean(axis=0)\n\n    Now plot\n\n    >>> plt.subplot(3,1,1) # doctest: +SKIP\n    >
↪>> plt.plot(freqs_0, data_0[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_0, data_0[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 0\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.subplot(3,1,2) # doctest: +SKIP\n    >>> plt.
↪plot(freqs_1, data_1[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_1, data_1[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 1\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.subplot(3,1,3) # doctest: +SKIP\n    >>> plt.
↪plot(freqs_2, data_2[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_2, data_2[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 2\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.xlabel(\"Frequency\") # doctest: +SKIP\n    >>> plt.
↪tight_layout() # doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "win_fnc": {
              "title": "Win Fnc",
              "default": [
                 "kaiser",
                 14
              ],
              "anyOf": [
                 {
                    "type": "string"
                 },
                 {
                    "type": "array",
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                       {
                          "type": "string"
                       },
                       {
                          "type": "number"
                       }
                    ]
                 }
              ]
```

(continues on next page)

(continued from previous page)

```
            },
            "detrend": {
               "title": "Detrend",
               "default": "linear",
               "type": "string"
            },
            "workers": {
               "title": "Workers",
               "default": -2,
               "type": "integer"
            }
         }
      },
      "SpectraProcess": {
         "title": "SpectraProcess",
         "description": "Parent class for spectra processes",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            }
         }
      },
      "EvaluationFreqs": {
         "title": "EvaluationFreqs",
         "description": "Calculate the spectra values at the evaluation frequencies\
↪n\nThis is done using linear interpolation in the complex domain\n\nExample\n-----
↪--\nThe example will show interpolation to evaluation frequencies on a very\
↪nsimple example. Begin by generating some example spectra data.\n\n>>> from␣
↪resistics.decimate import DecimationSetup\n>>> from resistics.spectra import␣
↪EvaluationFreqs\n>>> from resistics.testing import spectra_data_basic\n>>> spec_
↪data = spectra_data_basic()\n>>> spec_data.metadata.n_levels\n1\n>>> spec_data.
↪metadata.chans\n['chan1']\n>>> spec_data.metadata.levels_metadata[0].summary()\n{\
↪n    'fs': 180.0,\n    'n_wins': 2,\n    'win_size': 20,\n    'olap_size': 5,\n  ␣
↪ 'index_offset': 0,\n    'n_freqs': 10,\n    'freqs': [0.0, 10.0, 20.0, 30.0, 40.
↪0, 50.0, 60.0, 70.0, 80.0, 90.0]\n}\n\nThe spectra data has only a single channel␣
↪and a single level which has 2\nwindows. Now define our evaluation frequencies.\n\
↪n>>> eval_freqs = [1, 12, 23, 34, 45, 56, 67, 78, 89]\n>>> dec_setup =␣
↪DecimationSetup(n_levels=1, per_level=9, eval_freqs=eval_freqs)\n>>> dec_params =␣
↪dec_setup.run(spec_data.metadata.fs[0])\n>>> dec_params.summary()\n{\n    'fs':␣
↪180.0,\n    'n_levels': 1,\n    'per_level': 9,\n    'min_samples': 256,\n
↪'eval_freqs': [1.0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0],\n    'dec_
↪factors': [1],\n    'dec_increments': [1],\n    'dec_fs': [180.0]\n}\n\nNow␣
↪calculate the spectra at the evaluation frequencies\n\n>>> eval_data =␣
↪EvaluationFreqs().run(dec_params, spec_data)\n>>> eval_data.metadata.levels_
↪metadata[0].summary()\n{\n    'fs': 180.0,\n    'n_wins': 2,\n    'win_size': 20,\
↪n    'olap_size': 5,\n    'index_offset': 0,\n    'n_freqs': 9,\n    'freqs': [1.
↪0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0]\n}\n\nTo double check␣
↪everything is as expected, let's compare the data. Comparing\nwindow 1 gives\n\n>>
↪> print(spec_data.data[0][0, 0])\n[0.+0.j 1.+1.j 2.+2.j 3.+3.j 4.+4.j 5.+5.j 6.+6.
↪j 7.+7.j 8.+8.j 9.+9.j]\n>>> print(eval_data.data[0][0, 0])\n[0.1+0.1j 1.2+1.2j 2.
```

(continues on next page)

(continued from previous page)

```
↪3+2.3j 3.4+3.4j 4.5+4.5j 5.6+5.6j 6.7+6.7j 7.8+7.8j\n 8.9+8.9j]\n\nAnd window 2\n\
↪n>>> print(spec_data.data[0][1, 0])\n[-1. +1.j  0. +2.j  1. +3.j  2. +4.j  3. +5.
↪j  4. +6.j  5. +7.j  6. +8.j\n  7. +9.j  8.+10.j]\n>>> print(eval_data.data[0][1,␣
↪0])\n[-0.9+1.1j  0.2+2.2j  1.3+3.3j  2.4+4.4j  3.5+5.5j  4.6+6.6j  5.7+7.7j\n  6.
↪8+8.8j  7.9+9.9j]",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            }
         }
      },
      "Calibrator": {
         "title": "Calibrator",
         "description": "Parent class for a calibrator",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "chans": {
               "title": "Chans",
               "type": "array",
               "items": {
                  "type": "string"
               }
            }
         }
      },
      "TransferFunction": {
         "title": "TransferFunction",
         "description": "Define a generic transfer function\n\nThis class is a␣
↪describes generic transfer function, including:\n\n- The output channels for the␣
↪transfer function\n- The input channels for the transfer function\n- The cross␣
↪channels for the transfer function\n\nThe cross channels are the channels that␣
↪will be used to calculate out the\ncross powers for the regression.\n\nThis␣
↪generic parent class has no implemented plotting function. However,\nchild␣
↪classes may have a plotting function as different transfer functions\nmay need␣
↪different types of plots.\n\n.. note::\n\n    Users interested in writing a␣
↪custom transfer function should inherit\n    from this generic Transfer function\
↪n\nSee Also\n--------\nImpandanceTensor : Transfer function for the MT impedance␣
↪tensor\nTipper : Transfer function for the MT tipper\n\nExamples\n--------\nA␣
↪generic example\n\n>>> from resistics.transfunc import TransferFunction\n>>> tf =␣
↪TransferFunction(variation=\"example\", out_chans=[\"bye\", \"see you\", \"ciao\
↪"], in_chans=[\"hello\", \"hi_there\"])\n>>> print(tf.to_string())\n| bye      | ␣
↪ | bye_hello         bye_hi_there      | | hello    |\n| see you  | = | see you_
↪hello     see you_hi_there  | | hi_there |\n| ciao     |   | ciao_hello       ␣
↪ciao_hi_there     |\n\nCombining the impedance tensor and the tipper into one␣
↪TransferFunction\n\n>>> tf = TransferFunction(variation=\"combined\", out_chans=[\
↪"Ex\", \"Ey\"], in_chans=[\"Hx\", \"Hy\", \"Hz\"])\n>>> print(tf.to_string())\n|␣
```

(continues on next page)

(continued from previous page)

```
↪Ex |   | Ex_Hx Ex_Hy Ex_Hz | | Hx |\n| Ey | = | Ey_Hx Ey_Hy Ey_Hz | | Hy |\n      ␣
↪                          | Hz |",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "variation": {
              "title": "Variation",
              "default": "generic",
              "maxLength": 16,
              "type": "string"
           },
           "out_chans": {
              "title": "Out Chans",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "in_chans": {
              "title": "In Chans",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "cross_chans": {
              "title": "Cross Chans",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "n_out": {
              "title": "N Out",
              "type": "integer"
           },
           "n_in": {
              "title": "N In",
              "type": "integer"
           },
           "n_cross": {
              "title": "N Cross",
              "type": "integer"
           }
        },
        "required": [
           "out_chans",
           "in_chans"
        ]
```

(continues on next page)

(continued from previous page)


```
      },
      "RegressionPreparerGathered": {
         "title": "RegressionPreparerGathered",
         "description": "Regression preparer for gathered data\n\nIn nearly all␣
↪cases, this is the regresson preparer to use. As input, it\nrequires GatheredData.
↪",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            }
         }
      },
      "Solver": {
         "title": "Solver",
         "description": "General resistics solver",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            }
         }
      }
   }
}
```


field name: `str` **[Required]**

The name of the configuration

field time_readers: `List[TimeReader]` **= [TimeReaderAscii(name='TimeReaderAscii', apply_scalings=True, extension='.txt', delimiter=None, n_header=0), TimeReaderNumpy(name='TimeReaderNumpy', apply_scalings=True, extension='.npy')]**

Time readers in the configuration

field time_processors: `List[TimeProcess]` **= [InterpolateNans(name='InterpolateNans'), RemoveMean(name='RemoveMean')]**

List of time processors to run

field dec_setup: *DecimationSetup* **= DecimationSetup(name='DecimationSetup', n_levels=8, per_level=5, min_samples=256, div_factor=2, eval_freqs=None)**

Process to calculate decimation parameters

field decimator: *Decimator* **= Decimator(name='Decimator', resample=True, max_single_factor=3)**

Process to decimate time data

field win_setup: *WindowSetup* **= WindowSetup(name='WindowSetup', min_size=128, min_olap=32, win_factor=4, olap_proportion=0.25, min_n_wins=5, win_sizes=None, olap_sizes=None)**

Process to calculate windowing parameters

field windower: *Windower* = Windower(name='Windower')

Process to window the decimated data

field fourier: *FourierTransform* = FourierTransform(name='FourierTransform', win_fnc=('kaiser', 14), detrend='linear', workers=-2)

Process to perform the fourier transform

field spectra_processors: List[*SpectraProcess*] = []

List of processors to run on spectra data

field evals: *EvaluationFreqs* = EvaluationFreqs(name='EvaluationFreqs')

Process to get the spectra data at the evaluation frequencies

field sensor_calibrator: *Calibrator* = SensorCalibrator(name='SensorCalibrator', chans=None, readers=[SensorCalibrationJSON(name='SensorCalibrationJSON', extension='.json', file_str='IC_$sensor$extension')])

The sensor calibrator and associated calibration file readers

field tf: *TransferFunction* = ImpedanceTensor(name='ImpedanceTensor', variation='default', out_chans=['Ex', 'Ey'], in_chans=['Hx', 'Hy'], cross_chans=['Hx', 'Hy'], n_out=2, n_in=2, n_cross=2)

The transfer function to solve

field regression_preparer: *RegressionPreparerGathered* = RegressionPreparerGathered(name='RegressionPreparerGathered')

Process to prepare linear equations

field solver: *Solver* = SolverScikitTheilSen(name='SolverScikitTheilSen', fit_intercept=False, normalize=False, n_jobs=-2, max_subpopulation=2000, n_subsamples=None)

The solver to use to estimate the regression parameters

resistics.config.**get_default_configuration**() → *Configuration*

Get the default configuration

resistics.decimate module

Module for time data decimation including classes and for the following

- Definition of DecimationParameters
- Performing decimation on time data

resistics.decimate.**get_eval_freqs_min**(*fs: float*, *f_min: float*) → ndarray

Calculate evaluation frequencies with mimum allowable frequency

Highest frequency is nyquist / 4

Parameters

- **fs** (*float*) – Sampling frequency
- **f_min** (*float*) – Minimum allowable frequency

Returns

Array of evaluation frequencies

Return type

np.ndarray

Raises
ValueError – If f_min <= 0

Examples

```
>>> from resistics.decimate import get_eval_freqs_min
>>> fs = 256
>>> get_eval_freqs_min(fs, 30)
array([64.      , 45.254834, 32.      ])
>>> get_eval_freqs_min(fs, 128)
Traceback (most recent call last):
...
ValueError: Minimum frequency 128 must be > 64.0
```

resistics.decimate.**get_eval_freqs_size**(*fs: float*, *n_freqs: int*) → ndarray

Calculate evaluation frequencies with maximum size

Highest frequency is nyquist/4

Parameters

- **fs** (*float*) – Sampling frequency
- **n_freqs** (*int*) – Number of evaluation frequencies

Returns
Array of evaluation frequencies

Return type
np.ndarray

Examples

```
>>> from resistics.decimate import get_eval_freqs_size
>>> fs = 256
>>> n_freqs = 3
>>> get_eval_freqs_size(fs, n_freqs)
array([64.      , 45.254834, 32.      ])
```

resistics.decimate.**get_eval_freqs**(*fs: float*, *f_min: float | None = None*, *n_freqs: int | None = None*) → ndarray

Get evaluation frequencies either based on size or a minimum frequency

Parameters

- **fs** (*float*) – Sampling frequency Hz
- **f_min** (*Optional[float], optional*) – Minimum cutoff for evaluation frequencies, by default None
- **n_freqs** (*Optional[int], optional*) – Number of evaluation frequencies, by default None

Returns
Evaluation frequencies array

Return type
np.ndarray

Raises
ValueError – ValueError if both f_min and n_freqs are None

Examples

```
>>> from resistics.decimate import get_eval_freqs
>>> get_eval_freqs(256, f_min=30)
array([64.      , 45.254834, 32.      ])
>>> get_eval_freqs(256, n_freqs=3)
array([64.      , 45.254834, 32.      ])
```

pydantic model resistics.decimate.**DecimationParameters**

Bases: *ResisticsModel*

Decimation parameters

Parameters

- **fs** (*float*) – Sampling frequency Hz
- **n_levels** (*int*) – Number of levels
- **per_level** (*int*) – Evaluation frequencies per level
- **min_samples** (*int*) – Number of samples to under which to quit decimating
- **eval_df** (*pd.DataFrame*) – The DataFrame with the decimation information

Examples

```
>>> from resistics.decimate import DecimationSetup
>>> dec_setup = DecimationSetup(n_levels=3, per_level=2)
>>> dec_params = dec_setup.run(128)
>>> type(dec_params)
<class 'resistics.decimate.DecimationParameters'>
>>> print(dec_params.to_dataframe())
                     0          1     fs  factors  increments
decimation level
0                 32.0  22.627417  128.0        1           1
1                 16.0  11.313708   64.0        2           2
2                  8.0   5.656854   32.0        4           2
>>> dec_params[2]
[8.0, 5.65685424949238]
>>> dec_params[2,1]
5.65685424949238
>>> dec_params.get_total_factor(2)
4
>>> dec_params.get_incremental_factor(2)
2
```

```
{
   "title": "DecimationParameters",
   "description": "Decimation parameters\n\nParameters\n----------\nfs : float\n    Sampling frequency Hz\nn_levels : int\n    Number of levels\nper_level : int\n    Evaluation frequencies per level\nmin_samples : int\n    Number of samples to under which to quit decimating\neval_df : pd.DataFrame\n    The DataFrame with the decimation information\n\nExamples\n--------\n>>> from resistics.decimate import DecimationSetup\n>>> dec_setup = DecimationSetup(n_levels=3, per_level=2)\n>>> dec_params = dec_setup.run(128)\n>>> type(dec_params)\n<class 'resistics.decimate.DecimationParameters'>\n>>> print(dec_params.to_dataframe())\n                0          1     fs  factors  increments\ndecimation level\n0           32.0  22.627417  128.0        1           1\n1           16.0  11.313708   64.0        2           2\n2            8.0   5.656854   32.0        4           2\n>>> dec_params[2]\n[8.0, 5.65685424949238]\n>>> dec_params[2,1]\n5.65685424949238\n>>> dec_params.get_total_factor(2)\n4\n>>> dec_params.get_incremental_factor(2)\n2",
   "type": "object",
   "properties": {
      "fs": {
         "title": "Fs",
         "type": "number"
      },
      "n_levels": {
         "title": "N Levels",
         "type": "integer"
      },
      "per_level": {
         "title": "Per Level",
         "type": "integer"
      },
      "min_samples": {
         "title": "Min Samples",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "eval_freqs": {
         "title": "Eval Freqs",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "dec_factors": {
         "title": "Dec Factors",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
      "dec_increments": {
         "title": "Dec Increments",
         "type": "array",
```


(continues on next page)

(continued from previous page)

```
            "items": {
               "type": "integer"
            }
         },
         "dec_fs": {
            "title": "Dec Fs",
            "type": "array",
            "items": {
               "type": "number"
            }
         }
      },
      "required": [
         "fs",
         "n_levels",
         "per_level",
         "min_samples",
         "eval_freqs",
         "dec_factors"
      ]
   }
```

field fs: *float* **[Required]**

field n_levels: *int* **[Required]**

field per_level: *int* **[Required]**

field min_samples: *PositiveInt* **[Required]**

Constraints

- **exclusiveMinimum** = 0

field eval_freqs: *List[float]* **[Required]**

field dec_factors: *List[int]* **[Required]**

field dec_increments: *List[int] | None* **= None**

Validated by

- `set_dec_increments`

field dec_fs: *List[float] | None* **= None**

Validated by

- `set_dec_fs`

check_level(*level: int*)

Check level

check_eval_idx(*idx: int*)

Check evaluation frequency index

get_eval_freqs(*level: int*) → List[float]

Get the evaluation frequencies for a level

Parameters
level (*int*) – The decimation level

Returns
List of evaluation frequencies

Return type
List[float]

get_eval_freq(*level: int*, *idx: int*) → float

Get an evaluation frequency

Parameters

- **level** (*int*) – The level
- **idx** (*int*) – Evaluation frequency index

Returns
The evaluation frequency

Return type
float

get_fs(*level: int*) → float

Get sampling frequency for level

Parameters
level (*int*) – The decimation level

Returns
Sampling frequency Hz

Return type
float

get_total_factor(*level: int*) → int

Get total decimation factor for a level

Parameters
level (*int*) – The level

Returns
The decimation factor

Return type
int

get_incremental_factor(*level: int*) → int

Get incremental decimation factor

Parameters
level (*int*) – The level

Returns
The incremental decimation factor

Return type
int

to_numpy() → ndarray

Get evaluation frequencies as a 2-D array

to_dataframe() → DataFrame

Provide decimation parameters as DataFrame

pydantic model resistics.decimate.**DecimationSetup**

Bases: *ResisticsProcess*

Process to calculate decimation parameters

Parameters

- **n_levels** (*int, optional*) – Number of decimation levels, by default 8
- **per_level** (*int, optional*) – Number of frequencies per level, by default 5
- **min_samples** (*int, optional*) – Number of samples under which to quit decimating
- **div_factor** (*int, optional*) – Minimum division factor for decimation, by default 2.
- **eval_freqs** (*Optional[List[float]], optional*) – Explicit definition of evaluation frequencies as a flat list, by default None. Must be of size n_levels * per_level

Examples

```
>>> from resistics.decimate import DecimationSetup
>>> dec_setup = DecimationSetup(n_levels=3, per_level=2)
>>> dec_params = dec_setup.run(128)
>>> print(dec_params.to_dataframe())
                     0          1     fs  factors  increments
decimation level
0                 32.0  22.627417  128.0        1           1
1                 16.0  11.313708   64.0        2           2
2                  8.0   5.656854   32.0        4           2
```

```
{
   "title": "DecimationSetup",
   "description": "Process to calculate decimation parameters\n\nParameters\n-------
↪---\nn_levels : int, optional\n    Number of decimation levels, by default 8\nper_
↪level : int, optional\n    Number of frequencies per level, by default 5\nmin_
↪samples : int, optional\n    Number of samples under which to quit decimating\
↪ndiv_factor : int, optional\n    Minimum division factor for decimation, by␣
↪default 2.\neval_freqs : Optional[List[float]], optional\n    Explicit definition␣
↪of evaluation frequencies as a flat list, by\n    default None. Must be of size n_
↪levels * per_level\n\nExamples\n--------\n>>> from resistics.decimate import␣
↪DecimationSetup\n>>> dec_setup = DecimationSetup(n_levels=3, per_level=2)\n>>>␣
↪dec_params = dec_setup.run(128)\n>>> print(dec_params.to_dataframe())\n          ␣
↪          0          1     fs  factors  increments\ndecimation level\n0         ␣
↪      32.0  22.627417  128.0        1           1\n1                 16.0  11.
↪313708   64.0        2           2\n2                  8.0   5.656854   32.0     ␣
↪  4           2",
   "type": "object",
   "properties": {
      "name": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Name",
            "type": "string"
        },
        "n_levels": {
            "title": "N Levels",
            "default": 8,
            "type": "integer"
        },
        "per_level": {
            "title": "Per Level",
            "default": 5,
            "type": "integer"
        },
        "min_samples": {
            "title": "Min Samples",
            "default": 256,
            "type": "integer"
        },
        "div_factor": {
            "title": "Div Factor",
            "default": 2,
            "type": "integer"
        },
        "eval_freqs": {
            "title": "Eval Freqs",
            "type": "array",
            "items": {
                "type": "number"
            }
        }
    }
}
```

field n_levels: *int* = 8

field per_level: *int* = 5

field min_samples: *int* = 256

field div_factor: *int* = 2

field eval_freqs: *List[float] | None* = None

run(*fs: float*) → *DecimationParameters*

Run DecimationSetup

Parameters
fs (*float*) – Sampling frequency, Hz

Returns
Decimation parameterisation

Return type
DecimationParameters

pydantic model resistics.decimate.**DecimatedLevelMetadata**

Bases: *Metadata*

Metadata for a decimation level


```
{
   "title": "DecimatedLevelMetadata",
   "description": "Metadata for a decimation level",
   "type": "object",
   "properties": {
      "fs": {
         "title": "Fs",
         "type": "number"
      },
      "n_samples": {
         "title": "N Samples",
         "type": "integer"
      },
      "first_time": {
         "title": "First Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "last_time": {
         "title": "Last Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      }
   },
   "required": [
      "fs",
      "n_samples",
      "first_time",
      "last_time"
   ]
}
```


field fs: *float* **[Required]**

The sampling frequency of the decimation level

field n_samples: *int* **[Required]**

The number of samples in the decimation level

field first_time: *HighResDateTime* **[Required]**

The first time in the decimation level

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = [‘2021-01-01 00:00:00.000061_035156_250000_000000’]

field last_time: *HighResDateTime* **[Required]**

The last time in the decimation level

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

property dt

pydantic model resistics.decimate.**DecimatedMetadata**

Bases: *WriteableMetadata*

Metadata for DecimatedData


```
{
   "title": "DecimatedMetadata",
   "description": "Metadata for DecimatedData",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "fs": {
         "title": "Fs",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "chans": {
         "title": "Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "n_chans": {
         "title": "N Chans",
         "type": "integer"
      },
      "n_levels": {
         "title": "N Levels",
         "type": "integer"
      },
      "first_time": {
         "title": "First Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "last_time": {
         "title": "Last Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
```


(continues on next page)

(continued from previous page)

```
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "system": {
            "title": "System",
            "default": "",
            "type": "string"
         },
         "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
         },
         "wgs84_latitude": {
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
         },
         "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
         },
         "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
         },
         "northing": {
            "title": "Northing",
            "default": -999.0,
            "type": "number"
         },
         "elevation": {
            "title": "Elevation",
            "default": -999.0,
            "type": "number"
         },
         "chans_metadata": {
            "title": "Chans Metadata",
            "type": "object",
            "additionalProperties": {
               "$ref": "#/definitions/ChanMetadata"
            }
         },
         "levels_metadata": {
            "title": "Levels Metadata",
            "type": "array",
            "items": {
               "$ref": "#/definitions/DecimatedLevelMetadata"
            }
```

(continues on next page)

(continued from previous page)

```
      },
      "history": {
         "title": "History",
         "default": {
            "records": []
         },
         "allOf": [
            {
               "$ref": "#/definitions/History"
            }
         ]
      }
   },
   "required": [
      "fs",
      "chans",
      "n_levels",
      "first_time",
      "last_time",
      "chans_metadata",
      "levels_metadata"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
               "title": "Version",
               "type": "string"
            }
         }
      },
      "ChanMetadata": {
         "title": "ChanMetadata",
         "description": "Channel metadata",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
```

(continues on next page)

(continued from previous page)

```
            },
            "data_files": {
               "title": "Data Files",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "chan_type": {
               "title": "Chan Type",
               "type": "string"
            },
            "chan_source": {
               "title": "Chan Source",
               "type": "string"
            },
            "sensor": {
               "title": "Sensor",
               "default": "",
               "type": "string"
            },
            "serial": {
               "title": "Serial",
               "default": "",
               "type": "string"
            },
            "gain1": {
               "title": "Gain1",
               "default": 1,
               "type": "number"
            },
            "gain2": {
               "title": "Gain2",
               "default": 1,
               "type": "number"
            },
            "scaling": {
               "title": "Scaling",
               "default": 1,
               "type": "number"
            },
            "chopper": {
               "title": "Chopper",
               "default": false,
               "type": "boolean"
            },
            "dipole_dist": {
               "title": "Dipole Dist",
               "default": 1,
               "type": "number"
            },
            "sensor_calibration_file": {
```

(continues on next page)

(continued from previous page)


```
            "title": "Sensor Calibration File",
            "default": "",
            "type": "string"
         },
         "instrument_calibration_file": {
            "title": "Instrument Calibration File",
            "default": "",
            "type": "string"
         }
      },
      "required": [
         "name"
      ]
   },
   "DecimatedLevelMetadata": {
      "title": "DecimatedLevelMetadata",
      "description": "Metadata for a decimation level",
      "type": "object",
      "properties": {
         "fs": {
            "title": "Fs",
            "type": "number"
         },
         "n_samples": {
            "title": "N Samples",
            "type": "integer"
         },
         "first_time": {
            "title": "First Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "last_time": {
            "title": "Last Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         }
      },
      "required": [
         "fs",
         "n_samples",
         "first_time",
         "last_time"
      ]
   },
   "Record": {
      "title": "Record",
      "description": "Class to hold a record\n\nA record holds information about␣
```


(continues on next page)

(continued from previous page)

```
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
        "type": "object",
        "properties": {
           "time_local": {
              "title": "Time Local",
              "type": "string",
              "format": "date-time"
           },
           "time_utc": {
              "title": "Time Utc",
              "type": "string",
              "format": "date-time"
           },
           "creator": {
              "title": "Creator",
              "type": "object"
           },
           "messages": {
              "title": "Messages",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "record_type": {
              "title": "Record Type",
              "type": "string"
           }
        },
        "required": [
           "creator",
           "messages",
           "record_type"
        ]
     },
     "History": {
        "title": "History",
        "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n    ␣
```

(continues on next page)

(continued from previous page)

```
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n               'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
         "type": "object",
         "properties": {
            "records": {
               "title": "Records",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/Record"
               }
            }
         }
      }
   }
}
```

field fs: List[float] **[Required]**

field chans: List[str] **[Required]**

field n_chans: int | None = None

Validated by

- validate_n_chans

field n_levels: int **[Required]**

field first_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field last_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field system: str = ''

field serial: str = ''

field wgs84_latitude: float = -999.0

field wgs84_longitude: float = -999.0

field easting: *float* = -999.0

field northing: *float* = -999.0

field elevation: *float* = -999.0

field chans_metadata: *Dict[str, ChanMetadata]* [Required]

field levels_metadata: *List[DecimatedLevelMetadata]* [Required]

field history: *History* = History(records=[])

class resistics.decimate.**DecimatedData**(*metadata:* DecimatedMetadata, *data: Dict[int, ndarray]*)

Bases: *ResisticsData*

Data class for storing decimated data

The data for is stored in a dictionary attribute named data. The indices are integers representing the decimation level. Each decimation level is a numpy array of shape:

n_chans x n_samples

Parameters

- **metadata** (DecimatedMetadata) – The metadata
- **data** (*Dict[int, TimeData]*) – The decimated time data

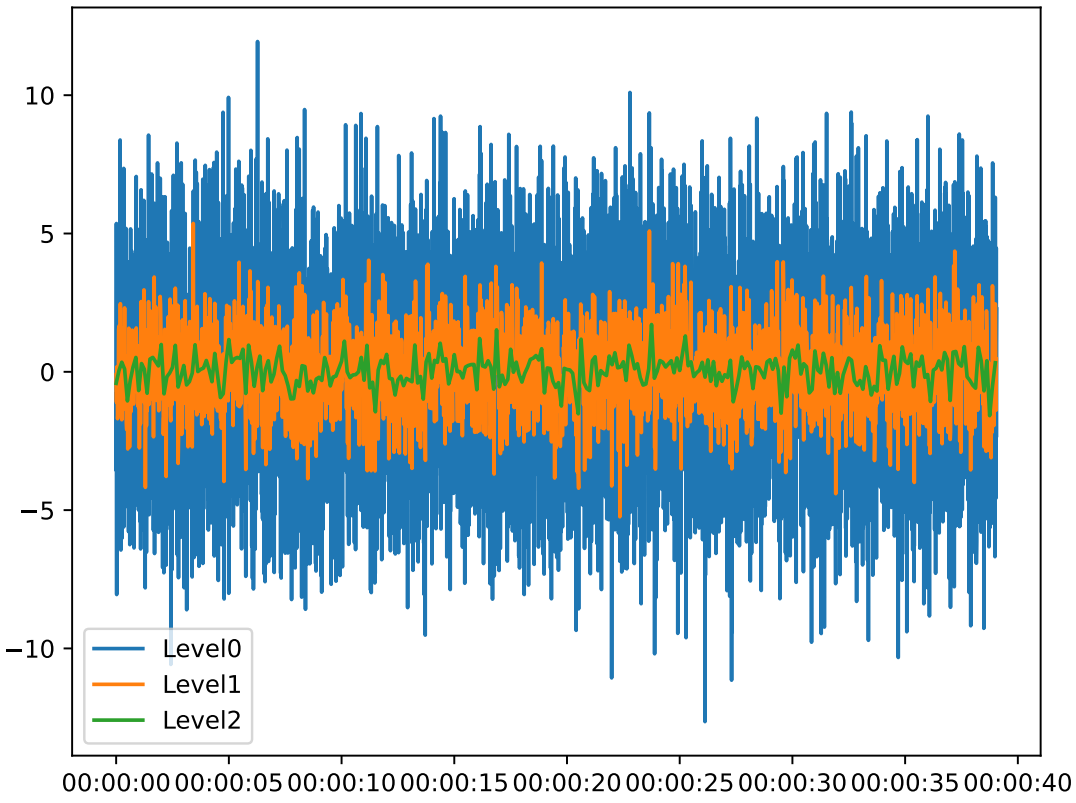
Examples

```
>>> import matplotlib.pyplot as plt
>>> from resistics.testing import time_data_random
>>> from resistics.decimate import DecimationSetup, Decimator
>>> time_data = time_data_random(fs=256, n_samples=10_000)
>>> dec_params = DecimationSetup(n_levels=4, per_freq=4).run(time_data.metadata.fs)
>>> dec_data = Decimator().run(dec_params, time_data)
>>> for level_metadata in dec_data.metadata.levels_metadata:
...     level_metadata.summary()
{
    'fs': 256.0,
    'n_samples': 10000,
    'first_time': '2020-01-01 00:00:00.000000_000000_000000_000000',
    'last_time': '2020-01-01 00:00:39.058593_750000_000000_000000'
}
{
    'fs': 64.0,
    'n_samples': 2500,
    'first_time': '2020-01-01 00:00:00.000000_000000_000000_000000',
    'last_time': '2020-01-01 00:00:39.046875_000000_000000_000000'
}
{
    'fs': 8.0,
    'n_samples': 313,
    'first_time': '2020-01-01 00:00:00.000000_000000_000000_000000',
    'last_time': '2020-01-01 00:00:39.000000_000000_000000_000000'
}
```

(continues on next page)

(continued from previous page)

```
>>> for ilevel in range(0, dec_data.metadata.n_levels):
...     data = dec_data.get_level(ilevel)
...     plt.plot(dec_data.get_timestamps(ilevel), data[0], label=f"Level{ilevel}")
>>> plt.legend(loc=3)
>>> plt.tight_layout()
>>> plt.show()
```



get_level(*level: int*) → ndarray

Get data for a decimation level

Parameters
level (*int*) – The level

Returns
The data for the decimation level

Return type
np.ndarary

Raises
ValueError – If level > max_level

get_timestamps(*level: int*, *samples: ndarray | None = None*, *estimate: bool = True*) → ndarray | DatetimeIndex

Get an array of timestamps

Parameters

- **level** (*int*) – The decimation level
- **samples** (*Optional[np.ndarray], optional*) – If provided, timestamps are only returned for the specified samples, by default None
- **estimate** (*bool, optional*) – Flag for using estimates instead of high precision datetimes, by default True

Returns
The return dates. This will be a numpy array of RSDateTime objects if estimate is False, else it will be a pandas DatetimeIndex

Return type
Union[np.ndarray, pd.DatetimeIndex]

Raises
ValueError – If the level is not within bounds

plot(*max_pts: int | None = 10000*) → Figure

Plot the decimated data

Parameters
max_pts (*Optional[int], optional*) – The maximum number of points in any individual plot before applying lttbc downsampling, by default 10_000. If set to None, no downsampling will be applied.

Returns
Plotly Figure

Return type
go.Figure

to_string() → str

Class details as a string

pydantic model resistics.decimate.**Decimator**

Bases: *ResisticsProcess*

Decimate the time data into multiple levels

There are two options for decimation, using time data Resample or using time data Decimate. The default is to use Resample.


```
{
   "title": "Decimator",
   "description": "Decimate the time data into multiple levels\n\nThere are two␣
↪options for decimation, using time data Resample or using\ntime data Decimate.␣
↪The default is to use Resample.",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "resample": {
         "title": "Resample",
         "default": true,
         "type": "boolean"
      },
```


(continues on next page)

(continued from previous page)


```
      "max_single_factor": {
         "title": "Max Single Factor",
         "default": 3,
         "minimum": 3,
         "type": "integer"
      }
   }
}
```


field resample: bool = True

Boolean flag for using resampling instead of decimation

field max_single_factor: ConstrainedIntValue = 3

Maximum single decimation factor, only used if resample is False

Constraints

- **minimum** = 3

run(*dec_params:* DecimationParameters, *time_data:* TimeData) → *DecimatedData*

Decimate the TimeData

Parameters

- **dec_params** (DecimationParameters) – The decimation parameters
- **time_data** (TimeData) – TimeData to decimate

Returns

DecimatedData instance with all the decimated data

Return type

DecimatedData

pydantic model resistics.decimate.**DecimatedDataWriter**

Bases: *ResisticsWriter*

Writer of resistics decimated data


```
{
   "title": "DecimatedDataWriter",
   "description": "Writer of resistics decimated data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "overwrite": {
         "title": "Overwrite",
         "default": true,
         "type": "boolean"
      }
   }
}
```

run(*dir_path: Path*, *dec_data:* DecimatedData) → None

Write out DecimatedData

Parameters

- **dir_path** (*Path*) – The directory path to write to
- **dec_data** (DecimatedData) – Decimated data to write out

Raises

WriteError – If unable to write to the directory

pydantic model resistics.decimate.**DecimatedDataReader**

Bases: *ResisticsProcess*

Reader of resistics decimated data

```
{
   "title": "DecimatedDataReader",
   "description": "Reader of resistics decimated data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```

run(*dir_path: Path*, *metadata_only: bool = False*) → *DecimatedMetadata* | *DecimatedData*

Read DecimatedData

Parameters

- **dir_path** (*Path*) – The directory path to read from
- **metadata_only** (*bool, optional*) – Flag for getting metadata only, by default False

Returns

DecimatedData or DecimatedMetadata if metadata_only

Return type

Union[*DecimatedMetadata*, *DecimatedData*]

Raises

ReadError – If the directory does not exist

resistics.errors module

Module for custom resistics errors

resistics.errors.**path_to_string**(*path: Path*) → str

Convert a path to a string in a OS agnostic way

Parameters

path (*Path*) – The path to convert

Returns

A string for the path

Return type
str

exception resistics.errors.**PathError**(*path: Path*)
Bases: Exception

Use for a general error with paths

exception resistics.errors.**PathNotFoundError**(*path: Path*)
Bases: *PathError*

Use if path does not exist

exception resistics.errors.**NotFileError**(*path: Path*)
Bases: *PathError*

Use if expected a file and got a directory

exception resistics.errors.**NotDirectoryError**(*path: Path*)
Bases: *PathError*

Use if expected a directory and got a file

exception resistics.errors.**WriteError**(*path: Path*, *message: str = ''*)
Bases: Exception

exception resistics.errors.**ReadError**(*path: Path*, *message: str = ''*)
Bases: Exception

exception resistics.errors.**MetadataReadError**(*path: Path*, *message: str | None = None*)
Bases: Exception

Use when failed to read a metadata

exception resistics.errors.**ProjectPathError**(*project_dir: Path*, *message: str*)
Bases: Exception

Use for a general error with a project path

exception resistics.errors.**ProjectCreateError**(*project_dir: Path*, *message: str*)
Bases: *ProjectPathError*

Use if encounter an error creating a project

exception resistics.errors.**ProjectLoadError**(*project_dir: Path*, *message: str*)
Bases: *ProjectPathError*

Use if error on project load

exception resistics.errors.**MeasurementNotFoundError**(*site_name: str*, *meas_name: str*)
Bases: Exception

Use if unable to find a measurement

exception resistics.errors.**SiteNotFoundError**(*site_name: str*)
Bases: Exception

Use if unable to find a site

exception resistics.errors.**TimeDataReadError**(*dir_path: Path*, *message: str*)
Bases: Exception

Use when encounter an error reading time series data

exception resistics.errors.**ChannelNotFoundError**(*chan: str*, *chans: Collection[str]*)

Bases: Exception

Use when a channel is not found

exception resistics.errors.**CalibrationFileNotFound**(*dir_path: Path*, *file_paths: Path | List[Path]*, *message: str = ''*)

Bases: Exception

Use when calibration files are not found

exception resistics.errors.**CalibrationFileReadError**(*calibration_path: Path*, *message: str = ''*)

Bases: Exception

Use if encounter an error reading a calibration file

exception resistics.errors.**ProcessRunError**(*process: str*, *message: str*)

Bases: Exception

Use when a error is encountered during a process run

resistics.gather module

Module for gathering data that will be combined to calculate transfer functions

There are two scenarios considered here. The first is the simplest, which is quick processing outside the project environment. In this case data gathering is not complicated. This workflow does not involve a data selector, meaining only a single step is required.

- QuickGather to put together the out_data, in_data and cross_data

When inside the project environment, regardless of whether it is single site or multi site processing, the workflow follows:

- Selector to select shared windows across all sites for a sampling frequency
- Gather to gather the combined evaluation frequency data

Warning: There may be some confusion in the code with many references to spectra data and evaluation frequency data. Evaluation frequency data, referred to below as eval_data is actually an instance of Spectra data. However, it is named differently to highlight the fact that it is not the complete spectra data, but is actually spectra data at a reduced set of frequencies corresponding to the evaluation frequncies.

Within a project, there are separate folders for users who want to save both the full spectra data with all the frequencies as well as the evaluation frequency spectra data with the smaller subset of frequencies. Only the evaluation frequency data is required to calculate the transfer function, but the full spectral data might be useful for visualisation and analysis reasons.

resistics.gather.**get_site_evals_metadata**(*config_name: str*, *proj:* Project, *site_name: str*, *fs: float*) → Dict[str, *SpectraMetadata*]

Get spectra metadata for a given site and sampling frequency

Parameters

- **config_name** (*str*) – The configuration name to get the right data
- **proj** (Project) – The project instance to get the measurements
- **site_name** (*str*) – The name of the site for which to gather the SpectraMetadata

- **fs** (*float*) – The original recording sampling frequency

Returns
Dictionary of measurement name to SpectraMetadata

Return type
Dict[str, *SpectraMetadata*]

resistics.gather.**get_site_level_wins**(*meas_metadata: Dict[str, SpectraMetadata]*, *level: int*) → Series

Get site windows for a decimation level given a sampling frequency

Parameters

- **meas_metadata** (*Dict[str, SpectraMetadata]*) – The measurement spectra metadata for a site
- **level** (*int*) – The decimation level

Returns
A series with an index of global windows for the site and values the measurements which have that global window. This is for a single decimation level

Return type
pd.Series

See also:

get_site_wins
Get windows for all decimation levels

Examples

An example getting the site windows for decimation level 0 when there are three measurements in the site.

```
>>> from resistics.testing import spectra_metadata_multilevel
>>> from resistics.gather import get_site_level_wins
>>> meas_metadata = {}
>>> meas_metadata["meas1"] = spectra_metadata_multilevel(n_wins=[3, 2, 2], index_
↪offset=[3, 2, 1])
>>> meas_metadata["meas2"] = spectra_metadata_multilevel(n_wins=[4, 3, 2], index_
↪offset=[28, 25, 22])
>>> meas_metadata["meas3"] = spectra_metadata_multilevel(n_wins=[2, 2, 1], index_
↪offset=[108, 104, 102])
>>> get_site_level_wins(meas_metadata, 0)
3      meas1
4      meas1
5      meas1
28     meas2
29     meas2
30     meas2
31     meas2
108    meas3
109    meas3
dtype: object
>>> get_site_level_wins(meas_metadata, 1)
2      meas1
3      meas1
```

(continues on next page)

(continued from previous page)

```
25     meas2
26     meas2
27     meas2
104    meas3
105    meas3
dtype: object
>>> get_site_level_wins(meas_metadata, 2)
1      meas1
2      meas1
22     meas2
23     meas2
102    meas3
dtype: object
```

resistics.gather.**get_site_wins**(*config_name: str*, *proj:* Project, *site_name: str*, *fs: float*) → Dict[int, Series]

Get site windows for all levels given a sampling frequency

Parameters

- **config_name** (*str*) – The configuration name to get the right data
- **proj** (Project) – The project instance to get the measurements
- **site_name** (*str*) – The site name
- **fs** (*float*) – The recording sampling frequency

Returns
Dictionary of integer to levels, with one entry for each decimation level

Return type
Dict[int, pd.Series]

Raises
ValueError – If no matching spectra metadata is found

class resistics.gather.**Selection**(*sites: List[*Site*]*, *dec_params:* DecimationParameters, *tables: Dict[int, DataFrame]*)

Bases: *ResisticsData*

Selections are output by the Selector. They hold information about the data that should be gathered for the regression.

get_n_evals() → int

Get the total number of evaluation frequnecies

Returns
The total number of evaluation frequencies that can be calculated

Return type
int

get_n_wins(*level: int*, *eval_idx: int*) → int

Get the number of windows for an evaluation frequency

Parameters

- **level** (*int*) – The decimation level
- **eval_idx** (*int*) – The evaluation frequency index in the decimation level

Returns
The number of windows

Return type
int

Raises
ValueError – If the level is greater than the maximum level available

get_measurements(*site:* Site) → List[str]

Get the measurement names to read from a Site

Parameters
site (Site) – The site for which to get the measurements

Returns
The measurements to read from

Return type
List[str]

get_eval_freqs() → List[float]

Get the evaluation frequencies

Returns
The evaluation frequencies as a flat list of floats

Return type
List[float]

get_eval_wins(*level: int*, *eval_idx: int*) → DataFrame

Limit the level windows to the evaluation frequency

Parameters

- **level** (*int*) – The decimation level
- **eval_idx** (*int*) – The evalution frequency index in the decimation level

Returns
pandas DataFrame of the windows and the measurements from each site the window can be read from

Return type
pd.DataFrame

pydantic model resistics.gather.**Selector**

Bases: *ResisticsProcess*

The Selector takes Sites and tries to find shared windows across them. A project instance is required for the Selector to be able to find shared windows.

The Selector should be used for remote reference and intersite processing and single site processing when masks are involved.

```
{
   "title": "Selector",
   "description": "The Selector takes Sites and tries to find shared windows across␣
↪them. A\nproject instance is required for the Selector to be able to find shared\
↪nwindows.\n\nThe Selector should be used for remote reference and intersite␣
↪processing\nand single site processing when masks are involved.",
```

(continues on next page)

(continued from previous page)


```
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*config_name: str*, *proj:* Project, *site_names: List[str]*, *dec_params:* DecimationParameters, *masks: Dict[str, str] | None = None*) → *Selection*

Run the selector

If a site repeats, the selector only considers it once. This might be the case when performing intersite or other cross site style processing.

Parameters

- **config_name** (*str*) – The configuration name
- **proj** (Project) – The project instance
- **site_names** (*List[str]*) – The names of the sites to get data from
- **dec_params** (DecimationParameters) – The decimation parameters with number of levels etc.
- **masks** (*Optional[Dict[str, str]], optional*) – Any masks to add, by default None

Returns

The Selection information defining the measurements and windows to read for each site

Return type

Selection

pydantic model resistics.gather.**SiteCombinedMetadata**

Bases: *WriteableMetadata*

Metadata for combined data

Combined metadata stores metadata for measurements that are combined from a single site.


```
{
   "title": "SiteCombinedMetadata",
   "description": "Metadata for combined data\n\nCombined metadata stores metadata␣
↪for measurements that are combined from\na single site.",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "site_name": {
         "title": "Site Name",
         "type": "string"
      },
      "fs": {
```


(continues on next page)

(continued from previous page)

```
            "title": "Fs",
            "type": "number"
        },
        "system": {
            "title": "System",
            "default": "",
            "type": "string"
        },
        "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
        },
        "wgs84_latitude": {
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
        },
        "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
        },
        "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
        },
        "northing": {
            "title": "Northing",
            "default": -999.0,
            "type": "number"
        },
        "elevation": {
            "title": "Elevation",
            "default": -999.0,
            "type": "number"
        },
        "measurements": {
            "title": "Measurements",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "chans": {
            "title": "Chans",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
```

(continues on next page)

(continued from previous page)


```
      "n_evals": {
         "title": "N Evals",
         "type": "integer"
      },
      "eval_freqs": {
         "title": "Eval Freqs",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "histories": {
         "title": "Histories",
         "type": "object",
         "additionalProperties": {
            "$ref": "#/definitions/History"
         }
      }
   },
   "required": [
      "site_name",
      "fs",
      "chans",
      "n_evals",
      "eval_freqs",
      "histories"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
               "title": "Version",
               "type": "string"
            }
         }
      },
      "Record": {
         "title": "Record",
         "description": "Class to hold a record\n\nA record holds information about␣
```


(continues on next page)

(continued from previous page)

```
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
        "type": "object",
        "properties": {
           "time_local": {
              "title": "Time Local",
              "type": "string",
              "format": "date-time"
           },
           "time_utc": {
              "title": "Time Utc",
              "type": "string",
              "format": "date-time"
           },
           "creator": {
              "title": "Creator",
              "type": "object"
           },
           "messages": {
              "title": "Messages",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "record_type": {
              "title": "Record Type",
              "type": "string"
           }
        },
        "required": [
           "creator",
           "messages",
           "record_type"
        ]
     },
     "History": {
        "title": "History",
        "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n    ␣
```

(continues on next page)

(continued from previous page)

```
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n               'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
         "type": "object",
         "properties": {
            "records": {
               "title": "Records",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/Record"
               }
            }
         }
      }
   }
}
```

field site_name: `str` **[Required]**

The name of the site

field fs: `float` **[Required]**

Recording sampling frequency

field system: `str` = ''

The system used for recording

field serial: `str` = ''

Serial number of the system

field wgs84_latitude: `float` = -999.0

Latitude in WGS84

field wgs84_longitude: `float` = -999.0

Longitude in WGS84

field easting: `float` = -999.0

The easting of the site in local cartersian coordinates

field northing: `float` = -999.0

The northing of the site in local cartersian coordinates

field elevation: `float` = -999.0

The elevation of the site

field measurements: `List[str] | None` = None

List of measurement names that were included in the combined data

field chans: `List[str]` **[Required]**

List of channels, these are common amongst all the measurements

field n_evals: `int` **[Required]**

The number of evaluation frequencies

field eval_freqs: `List[float]` **[Required]**

The evaluation frequencies

field histories: `Dict[str, History]` **[Required]**

Dictionary mapping measurement name to measurement processing history

class resistics.gather.**SiteCombinedData**(*metadata:* SiteCombinedMetadata, *data: Dict[int, ndarray]*)

Bases: *ResisticsData*

Combined data is data that is combined from a single site for the purposes of regression.

All of the data that is combined should have the same sampling frequency, same evaluation frequencies and some shared channels.

Data is stored in the data attribute of the class. This is a dictionary mapping evaluation frequency index to data for the evaluation frequency from all windows in the site. The shape of data for a single evaluation frequency is:

n_wins x n_chans

The data is complex valued.

class resistics.gather.**GatheredData**(*out_data:* SiteCombinedData, *in_data:* SiteCombinedData, *cross_data:* SiteCombinedData)

Bases: *ResisticsData*

Class to hold data to be used in by Regression preparers

Gathered data has an out_data, in_data and cross_data. The important thing here is that the data is all aligned with regards to windows

pydantic model resistics.gather.**ProjectGather**

Bases: *ResisticsProcess*

Gather aligned data from a single or multiple sites in the project

Aligned data means that the same index of data across multiple sites points to data covering the same global window (i.e. the same time window). This is essential for calculating intersite or remote reference transfer functions.


```
{
   "title": "ProjectGather",
   "description": "Gather aligned data from a single or multiple sites in the␣
↪project\n\nAligned data means that the same index of data across multiple sites␣
↪points\nto data covering the same global window (i.e. the same time window). This\
↪nis essential for calculating intersite or remote reference transfer\nfunctions.",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*config_name: str*, *proj:* Project, *selection:* Selection, *tf:* TransferFunction, *out_name: str*, *in_name: str | None = None*, *cross_name: str | None = None*) → *GatheredData*

Gather data for input into the regression preparer

Parameters

- **config_name** (*str*) – The config name for getting the correct evals data
- **proj** (Project) – The project instance
- **selection** (Selection) – The selection
- **tf** (TransferFunction) – The transfer function
- **out_name** (*str*) – The name of the output site
- **in_name** (*Optional[str], optional*) – The name of the input site, by default None
- **cross_name** (*Optional[str], optional*) – The name of the cross site, by default None

Returns

The data gathered for the regression preparer

Return type

GatheredData

pydantic model resistics.gather.**QuickGather**

Bases: *ResisticsProcess*

Processor to gather data outside of a resistics environment

This is intended for use when quickly calculating out a transfer function for a single measurement and only a single spectra data instance is accepted as input.

Remote reference or intersite processing is not possible using QuickGather

See also:

ProjectGather

For more advanced gathering of data in a project


```
{
   "title": "QuickGather",
   "description": "Processor to gather data outside of a resistics environment\n\
↪nThis is intended for use when quickly calculating out a transfer function\nfor a␣
↪single measurement and only a single spectra data instance is accepted\nas input.\
↪n\nRemote reference or intersite processing is not possible using QuickGather\n\
↪nSee Also\n--------\nProjectGather : For more advanced gathering of data in a␣
↪project",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*dir_path: Path*, *dec_params:* DecimationParameters, *tf:* TransferFunction, *eval_data:* SpectraData) → *GatheredData*

Generate the GatheredData object for input into regression preparation

The input is a single spectra data instance and is used to populate the in_data, out_data and cross_data.

Parameters

- **dir_path** (*Path*) – The directory path to the measurement
- **dec_params** (DecimationParameters) – The decimation parameters
- **tf** (TransferFunction) – The transfer function
- **eval_data** (SpectraData) – The spectra data at the evaluation frequencies

Returns
GatheredData for regression preparer

Return type
GatheredData

resistics.letsgo module

This module is the main interface to resistics and includes:

- Classes and functions for making, loading and using resistics projects
- Functions for processing data

pydantic model resistics.letsgo.**ProjectCreator**

Bases: *ResisticsProcess*

Process to create a project


```
{
   "title": "ProjectCreator",
   "description": "Process to create a project",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "dir_path": {
         "title": "Dir Path",
         "type": "string",
         "format": "path"
      },
      "metadata": {
         "$ref": "#/definitions/ProjectMetadata"
      }
   },
   "required": [
      "dir_path",
      "metadata"
   ],
   "definitions": {
      "ResisticsFile": {
```


(continues on next page)

(continued from previous page)


```
        "title": "ResisticsFile",
        "description": "Required information for writing out a resistics file",
        "type": "object",
        "properties": {
          "created_on_local": {
            "title": "Created On Local",
            "type": "string",
            "format": "date-time"
          },
          "created_on_utc": {
            "title": "Created On Utc",
            "type": "string",
            "format": "date-time"
          },
          "version": {
            "title": "Version",
            "type": "string"
          }
        }
      },
      "ProjectMetadata": {
        "title": "ProjectMetadata",
        "description": "Project metadata",
        "type": "object",
        "properties": {
          "file_info": {
            "$ref": "#/definitions/ResisticsFile"
          },
          "ref_time": {
            "title": "Ref Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
              "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
          },
          "location": {
            "title": "Location",
            "default": "",
            "type": "string"
          },
          "country": {
            "title": "Country",
            "default": "",
            "type": "string"
          },
          "year": {
            "title": "Year",
            "default": -999,
            "type": "integer"
          },
          "description": {
            "title": "Description",
```


(continues on next page)

(continued from previous page)


```
                "default": "",
                "type": "string"
            },
            "contributors": {
                "title": "Contributors",
                "default": [],
                "type": "array",
                "items": {
                    "type": "string"
                }
            }
        },
        "required": [
            "ref_time"
        ]
    }
}
}
```


field dir_path: Path [Required]

field metadata: *ProjectMetadata* [Required]

run()

Create the project

Raises

ProjectCreateError – If an existing project found

resistics.letsgo.**new**(*dir_path: Path | str*, *proj_info: Dict[str, Any]*) → bool

Create a new project

Parameters

- **dir_path** (*Union[Path, str]*) – The directory to create the project in
- **proj_info** (*Dict[str, Any]*) – Any project details

Returns

True if the creator was successful

Return type

bool

pydantic model resistics.letsgo.**ProjectLoader**

Bases: *ResisticsProcess*

Project loader


```
{
   "title": "ProjectLoader",
   "description": "Project loader",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
```


(continues on next page)

(continued from previous page)

```
            "type": "string"
        },
        "dir_path": {
            "title": "Dir Path",
            "type": "string",
            "format": "path"
        }
    },
    "required": [
        "dir_path"
    ]
}
```

field dir_path: Path [Required]

run(*config:* Configuration) → *Project*

Load a project

Parameters
config (Configuration) – The configuration for the purposes of getting the time readers

Returns
Project instance

Return type
Project

Raises
ProjectLoadError – If the resistcs project metadata is not found

pydantic model resistics.letsgo.**ResisticsEnvironment**

Bases: *ResisticsModel*

A Resistics environment which combines a project and a configuration

```
{
    "title": "ResisticsEnvironment",
    "description": "A Resistics environment which combines a project and a␣
↪configuration",
    "type": "object",
    "properties": {
        "proj": {
            "$ref": "#/definitions/Project"
        },
        "config": {
            "$ref": "#/definitions/Configuration"
        }
    },
    "required": [
        "proj",
        "config"
    ],
    "definitions": {
        "ResisticsFile": {
            "title": "ResisticsFile",
```

(continues on next page)

(continued from previous page)


```
        "description": "Required information for writing out a resistics file",
        "type": "object",
        "properties": {
          "created_on_local": {
            "title": "Created On Local",
            "type": "string",
            "format": "date-time"
          },
          "created_on_utc": {
            "title": "Created On Utc",
            "type": "string",
            "format": "date-time"
          },
          "version": {
            "title": "Version",
            "type": "string"
          }
        }
      },
      "ProjectMetadata": {
        "title": "ProjectMetadata",
        "description": "Project metadata",
        "type": "object",
        "properties": {
          "file_info": {
            "$ref": "#/definitions/ResisticsFile"
          },
          "ref_time": {
            "title": "Ref Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
              "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
          },
          "location": {
            "title": "Location",
            "default": "",
            "type": "string"
          },
          "country": {
            "title": "Country",
            "default": "",
            "type": "string"
          },
          "year": {
            "title": "Year",
            "default": -999,
            "type": "integer"
          },
          "description": {
            "title": "Description",
            "default": "",
```


(continues on next page)

(continued from previous page)

```
            "type": "string"
         },
         "contributors": {
            "title": "Contributors",
            "default": [],
            "type": "array",
            "items": {
               "type": "string"
            }
         }
      },
      "required": [
         "ref_time"
      ]
   },
   "ChanMetadata": {
      "title": "ChanMetadata",
      "description": "Channel metadata",
      "type": "object",
      "properties": {
         "name": {
            "title": "Name",
            "type": "string"
         },
         "data_files": {
            "title": "Data Files",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "chan_type": {
            "title": "Chan Type",
            "type": "string"
         },
         "chan_source": {
            "title": "Chan Source",
            "type": "string"
         },
         "sensor": {
            "title": "Sensor",
            "default": "",
            "type": "string"
         },
         "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
         },
         "gain1": {
            "title": "Gain1",
            "default": 1,
```

(continues on next page)

(continued from previous page)


```
                "type": "number"
            },
            "gain2": {
                "title": "Gain2",
                "default": 1,
                "type": "number"
            },
            "scaling": {
                "title": "Scaling",
                "default": 1,
                "type": "number"
            },
            "chopper": {
                "title": "Chopper",
                "default": false,
                "type": "boolean"
            },
            "dipole_dist": {
                "title": "Dipole Dist",
                "default": 1,
                "type": "number"
            },
            "sensor_calibration_file": {
                "title": "Sensor Calibration File",
                "default": "",
                "type": "string"
            },
            "instrument_calibration_file": {
                "title": "Instrument Calibration File",
                "default": "",
                "type": "string"
            }
        },
        "required": [
            "name"
        ]
    },
    "Record": {
        "title": "Record",
        "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
        "type": "object",
        "properties": {
            "time_local": {
```


(continues on next page)

(continued from previous page)


```
            "title": "Time Local",
            "type": "string",
            "format": "date-time"
         },
         "time_utc": {
            "title": "Time Utc",
            "type": "string",
            "format": "date-time"
         },
         "creator": {
            "title": "Creator",
            "type": "object"
         },
         "messages": {
            "title": "Messages",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "record_type": {
            "title": "Record Type",
            "type": "string"
         }
      },
      "required": [
         "creator",
         "messages",
         "record_type"
      ]
   },
   "History": {
      "title": "History",
      "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n    ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n    ␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n    ␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
      "type": "object",
      "properties": {
         "records": {
            "title": "Records",
            "default": [],
```


(continues on next page)

(continued from previous page)

```
            "type": "array",
            "items": {
               "$ref": "#/definitions/Record"
            }
         }
      }
   },
   "TimeMetadata": {
      "title": "TimeMetadata",
      "description": "Time metadata",
      "type": "object",
      "properties": {
         "file_info": {
            "$ref": "#/definitions/ResisticsFile"
         },
         "fs": {
            "title": "Fs",
            "type": "number"
         },
         "chans": {
            "title": "Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "n_chans": {
            "title": "N Chans",
            "type": "integer"
         },
         "n_samples": {
            "title": "N Samples",
            "type": "integer"
         },
         "first_time": {
            "title": "First Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "last_time": {
            "title": "Last Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "system": {
            "title": "System",
            "default": "",
            "type": "string"
```

(continues on next page)

(continued from previous page)

```
            },
            "serial": {
               "title": "Serial",
               "default": "",
               "type": "string"
            },
            "wgs84_latitude": {
               "title": "Wgs84 Latitude",
               "default": -999.0,
               "type": "number"
            },
            "wgs84_longitude": {
               "title": "Wgs84 Longitude",
               "default": -999.0,
               "type": "number"
            },
            "easting": {
               "title": "Easting",
               "default": -999.0,
               "type": "number"
            },
            "northing": {
               "title": "Northing",
               "default": -999.0,
               "type": "number"
            },
            "elevation": {
               "title": "Elevation",
               "default": -999.0,
               "type": "number"
            },
            "chans_metadata": {
               "title": "Chans Metadata",
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/ChanMetadata"
               }
            },
            "history": {
               "title": "History",
               "default": {
                  "records": []
               },
               "allOf": [
                  {
                     "$ref": "#/definitions/History"
                  }
               ]
            }
         },
         "required": [
            "fs",
```

(continues on next page)

(continued from previous page)


```
            "chans",
            "n_samples",
            "first_time",
            "last_time",
            "chans_metadata"
         ]
      },
      "TimeReader": {
         "title": "TimeReader",
         "description": "Base class for resistics processes\n\nResistics processes␣
↪perform operations on data (including read and write\noperations). Each time a␣
↪ResisticsProcess child class is run, it should add\na process record to the␣
↪dataset",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "apply_scalings": {
               "title": "Apply Scalings",
               "default": true,
               "type": "boolean"
            },
            "extension": {
               "title": "Extension",
               "type": "string"
            }
         }
      },
      "Measurement": {
         "title": "Measurement",
         "description": "Class for interfacing with a measurement\n\nThe class␣
↪holds the original time series metadata and can provide\ninformation about other␣
↪types of data",
         "type": "object",
         "properties": {
            "site_name": {
               "title": "Site Name",
               "type": "string"
            },
            "dir_path": {
               "title": "Dir Path",
               "type": "string",
               "format": "path"
            },
            "metadata": {
               "$ref": "#/definitions/TimeMetadata"
            },
            "reader": {
               "$ref": "#/definitions/TimeReader"
            }
```


(continues on next page)

(continued from previous page)

```
        },
        "required": [
           "site_name",
           "dir_path",
           "metadata",
           "reader"
        ]
     },
     "Site": {
        "title": "Site",
        "description": "Class for describing Sites\n\n.. note::\n\n    This should␣
↪essentially describe a single instrument setup. If the same\n    site is re-
↪occupied later with a different instrument setup, it is\n    suggested to split␣
↪this into a different site.",
        "type": "object",
        "properties": {
           "dir_path": {
              "title": "Dir Path",
              "type": "string",
              "format": "path"
           },
           "measurements": {
              "title": "Measurements",
              "type": "object",
              "additionalProperties": {
                 "$ref": "#/definitions/Measurement"
              }
           },
           "begin_time": {
              "title": "Begin Time",
              "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
              "examples": [
                 "2021-01-01 00:00:00.000061_035156_250000_000000"
              ]
           },
           "end_time": {
              "title": "End Time",
              "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
              "examples": [
                 "2021-01-01 00:00:00.000061_035156_250000_000000"
              ]
           }
        },
        "required": [
           "dir_path",
           "measurements",
           "begin_time",
           "end_time"
        ]
     },
     "Project": {
        "title": "Project",
```

(continues on next page)

(continued from previous page)

```
        "description": "Class to describe a resistics project\n\nThe resistics␣
↪Project Class connects all resistics data. It is an essential\npart of processing␣
↪data with resistics.\n\nResistics projects are in directory with several sub-
↪directories. Project\nmetadata is saved in the resistics.json file at the top␣
↪level directory.",
        "type": "object",
        "properties": {
           "dir_path": {
              "title": "Dir Path",
              "type": "string",
              "format": "path"
           },
           "begin_time": {
              "title": "Begin Time",
              "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
              "examples": [
                 "2021-01-01 00:00:00.000061_035156_250000_000000"
              ]
           },
           "end_time": {
              "title": "End Time",
              "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
              "examples": [
                 "2021-01-01 00:00:00.000061_035156_250000_000000"
              ]
           },
           "metadata": {
              "$ref": "#/definitions/ProjectMetadata"
           },
           "sites": {
              "title": "Sites",
              "default": {},
              "type": "object",
              "additionalProperties": {
                 "$ref": "#/definitions/Site"
              }
           }
        },
        "required": [
           "dir_path",
           "begin_time",
           "end_time",
           "metadata"
        ]
     },
     "TimeProcess": {
        "title": "TimeProcess",
        "description": "Parent class for processing time data",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
```

(continues on next page)

(continued from previous page)


```
            "type": "string"
         }
      }
   },
   "DecimationSetup": {
      "title": "DecimationSetup",
      "description": "Process to calculate decimation parameters\n\nParameters\n-
↪---------\nn_levels : int, optional\n    Number of decimation levels, by default␣
↪8\nper_level : int, optional\n    Number of frequencies per level, by default 5\
↪nmin_samples : int, optional\n    Number of samples under which to quit␣
↪decimating\ndiv_factor : int, optional\n    Minimum division factor for␣
↪decimation, by default 2.\neval_freqs : Optional[List[float]], optional\n   ␣
↪Explicit definition of evaluation frequencies as a flat list, by\n    default␣
↪None. Must be of size n_levels * per_level\n\nExamples\n--------\n>>> from␣
↪resistics.decimate import DecimationSetup\n>>> dec_setup = DecimationSetup(n_
↪levels=3, per_level=2)\n>>> dec_params = dec_setup.run(128)\n>>> print(dec_params.
↪to_dataframe())\n                     0          1     fs  factors  increments\
↪ndecimation level\n0                 32.0  22.627417  128.0        1           1\
↪n1                16.0  11.313708   64.0        2           2\n2                ␣
↪ 8.0   5.656854   32.0        4           2",
      "type": "object",
      "properties": {
         "name": {
            "title": "Name",
            "type": "string"
         },
         "n_levels": {
            "title": "N Levels",
            "default": 8,
            "type": "integer"
         },
         "per_level": {
            "title": "Per Level",
            "default": 5,
            "type": "integer"
         },
         "min_samples": {
            "title": "Min Samples",
            "default": 256,
            "type": "integer"
         },
         "div_factor": {
            "title": "Div Factor",
            "default": 2,
            "type": "integer"
         },
         "eval_freqs": {
            "title": "Eval Freqs",
            "type": "array",
            "items": {
               "type": "number"
            }
```


(continues on next page)

(continued from previous page)

```
            }
         }
      },
      "Decimator": {
         "title": "Decimator",
         "description": "Decimate the time data into multiple levels\n\nThere are␣
↪two options for decimation, using time data Resample or using\ntime data Decimate.
↪ The default is to use Resample.",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "resample": {
               "title": "Resample",
               "default": true,
               "type": "boolean"
            },
            "max_single_factor": {
               "title": "Max Single Factor",
               "default": 3,
               "minimum": 3,
               "type": "integer"
            }
         }
      },
      "WindowSetup": {
         "title": "WindowSetup",
         "description": "Setup WindowParameters\n\nWindowSetup outputs the␣
↪WindowParameters to use for windowing decimated\ntime data.\n\nWindow parameters␣
↪are simply the window and overlap sizes for each\ndecimation level.\n\nParameters\
↪n----------\nmin_size : int, optional\n    Minimum window size, by default 128\
↪nmin_olap : int, optional\n    Minimum overlap size, by default 32\nwin_factor :␣
↪int, optional\n    Window factor, by default 4. Window sizes are calculated by␣
↪sampling\n    frequency / 4 to ensure sufficient frequency resolution. If the\n  ␣
↪ sampling frequency is small, window size will be adjusted to\n    min_size\nolap_
↪proportion : float, optional\n    The proportion of the window size to use as the␣
↪overlap, by default\n    0.25. For example, for a window size of 128, the overlap␣
↪would be\n    0.25 * 128 = 32\nmin_n_wins : int, optional\n    The minimum number␣
↪of windows needed in a decimation level, by\n    default 5\nwin_sizes :␣
↪Optional[List[int]], optional\n    Explicit define window sizes, by default None.␣
↪Must have the same\n    length as number of decimation levels\nolap_sizes :␣
↪Optional[List[int]], optional\n    Explicitly define overlap sizes, by default␣
↪None. Must have the same\n    length as number of decimation levels\n\nExamples\n-
↪-------\nGenerate decimation and windowing parameters for data sampled at 0.05 Hz␣
↪or\n20 seconds sampling period\n\n>>> from resistics.decimate import␣
↪DecimationSetup\n>>> from resistics.window import WindowSetup\n>>> dec_params =␣
↪DecimationSetup(n_levels=3, per_level=3).run(0.05)\n>>> dec_params.summary()\n{\n␣
↪   'fs': 0.05,\n    'n_levels': 3,\n    'per_level': 3,\n    'min_samples': 256,\
↪n    'eval_freqs': [\n        0.0125,\n        0.008838834764831844,\n        0.
↪00625,\n        0.004419417382415922,\n        0.003125,\n        0.
```

(continues on next page)

(continued from previous page)

```
↪002209708691207961,\n        0.0015625,\n        0.0011048543456039805,\n       ␣
↪0.00078125\n    ],\n    'dec_factors': [1, 2, 8],\n    'dec_increments': [1, 2,␣
↪4],\n    'dec_fs': [0.05, 0.025, 0.00625]\n}\n>>> win_params = WindowSetup().
↪run(dec_params.n_levels, dec_params.dec_fs)\n>>> win_params.summary()\n{\n    'n_
↪levels': 3,\n    'min_n_wins': 5,\n    'win_sizes': [128, 128, 128],\n    'olap_
↪sizes': [32, 32, 32]\n}\n\nWindow parameters can also be explicitly defined\n\n>>>
↪ from resistics.decimate import DecimationSetup\n>>> from resistics.window import␣
↪WindowSetup\n>>> dec_setup = DecimationSetup(n_levels=3, per_level=3)\n>>> dec_
↪params = dec_setup.run(0.05)\n>>> win_setup = WindowSetup(win_sizes=[1000, 578,␣
↪104])\n>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)\n>>>
↪ win_params.summary()\n{\n    'n_levels': 3,\n    'min_n_wins': 5,\n    'win_sizes
↪': [1000, 578, 104],\n    'olap_sizes': [250, 144, 32]\n}\n\nWhen providing␣
↪explicit window and overlap sizes, the size of the overlap\nis expected to be␣
↪less than the size of the window\n\n>>> from resistics.decimate import␣
↪DecimationSetup\n>>> from resistics.window import WindowSetup\n>>> dec_setup =␣
↪DecimationSetup(n_levels=3, per_level=3)\n>>> dec_params = dec_setup.run(0.05)\n>>
↪> win_setup = WindowSetup(win_sizes=[1000, 578, 104], olap_sizes=[1001, 600, 25])\
↪n>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)\
↪nTraceback (most recent call last):\n...\nValueError: Invalid overlaps found [␣
↪True  True False]",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "min_size": {
              "title": "Min Size",
              "default": 128,
              "type": "integer"
           },
           "min_olap": {
              "title": "Min Olap",
              "default": 32,
              "type": "integer"
           },
           "win_factor": {
              "title": "Win Factor",
              "default": 4,
              "type": "integer"
           },
           "olap_proportion": {
              "title": "Olap Proportion",
              "default": 0.25,
              "type": "number"
           },
           "min_n_wins": {
              "title": "Min N Wins",
              "default": 5,
              "type": "integer"
           },
           "win_sizes": {
```

(continues on next page)

(continued from previous page)


```
            "title": "Win Sizes",
            "type": "array",
            "items": {
               "type": "integer"
            }
         },
         "olap_sizes": {
            "title": "Olap Sizes",
            "type": "array",
            "items": {
               "type": "integer"
            }
         }
      }
   },
   "Windower": {
      "title": "Windower",
      "description": "Windows DecimatedData\n\nThis is the primary window making␣
↪process for resistics and should be used\nwhen alignment of windows with a site␣
↪or across sites is required.\n\nThis method uses numpy striding to produce window␣
↪views into the decimated\ndata.\n\nSee Also\n--------\nWindowerTarget : A␣
↪windower to make a target number of windows\n\nExamples\n--------\nThe Windower␣
↪windows a DecimatedData object given a reference time and some\nwindow parameters.
↪\n\nThere's quite a few imports needed for this example. Begin by doing the\
↪nimports, defining a reference time and generating random decimated data.\n\n>>>␣
↪from resistics.sampling import to_datetime\n>>> from resistics.testing import␣
↪decimated_data_linear\n>>> from resistics.window import WindowSetup, Windower\n>>>
↪ dec_data = decimated_data_linear(fs=128)\n>>> ref_time = dec_data.metadata.first_
↪time\n>>> print(dec_data.to_string())\n<class 'resistics.decimate.DecimatedData'>\
↪n           fs        dt  n_samples           first_time                    ␣
↪last_time\nlevel\n0      2048.0  0.000488      16384  2021-01-01 00:00:00  2021-
↪01-01 00:00:07.99951171875\n1       512.0  0.001953       4096  2021-01-01␣
↪00:00:00    2021-01-01 00:00:07.998046875\n2       128.0  0.007812       1024 ␣
↪2021-01-01 00:00:00      2021-01-01 00:00:07.9921875\n\nNext, initialise the␣
↪window parameters. For this example, use small windows,\nwhich will make␣
↪inspecting them easier.\n\n>>> win_params = WindowSetup(win_sizes=[16,16,16], min_
↪olap=4).run(dec_data.metadata.n_levels, dec_data.metadata.fs)\n>>> win_params.
↪summary()\n{\n    'n_levels': 3,\n    'min_n_wins': 5,\n    'win_sizes': [16, 16,␣
↪16],\n    'olap_sizes': [4, 4, 4]\n}\n\nPerform the windowing. This actually␣
↪creates views into the decimated data\nusing the numpy.lib.stride_tricks.sliding_
↪window_view function. The shape\nfor a data array at a decimation level is: n_
↪wins x n_chans x win_size. The\ninformation about each level is also in the␣
↪levels_metadata attribute of\nWindowedMetadata.\n\n>>> win_data = Windower().
↪run(ref_time, win_params, dec_data)\n>>> win_data.data[0].shape\n(1365, 2, 16)\n>>
↪> for level_metadata in win_data.metadata.levels_metadata:\n...     level_
↪metadata.summary()\n{\n    'fs': 2048.0,\n    'n_wins': 1365,\n    'win_size': 16,
↪\n    'olap_size': 4,\n    'index_offset': 0\n}\n{\n    'fs': 512.0,\n    'n_wins
↪': 341,\n    'win_size': 16,\n    'olap_size': 4,\n    'index_offset': 0\n}\n{\n ␣
↪  'fs': 128.0,\n    'n_wins': 85,\n    'win_size': 16,\n    'olap_size': 4,\n
↪'index_offset': 0\n}\n\nLet's look at an example of data from the first␣
↪decimation level for the\nfirst channel. This is simply a linear set of data␣
↪ranging from 0...16_383.\n\n>>> dec_data.data[0][0]\narray([    0,     1,     2, .
```


(continues on next page)

(continued from previous page)

```
↪.., 16381, 16382, 16383])\n\nInspecting the first few windows shows they are as␣
↪expected including the\noverlap.\n\n>>> win_data.data[0][0, 0]\narray([ 0,  1,  2,
↪  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])\n>>> win_data.data[0][1, 0]\
↪narray([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27])\n>>> win_
↪data.data[0][2, 0]\narray([24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
↪ 38, 39])",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           }
        }
     },
     "FourierTransform": {
        "title": "FourierTransform",
        "description": "Perform a Fourier transform of the windowed data\n\nThe␣
↪processor is inspired by the scipy.signal.stft function which performs\na similar␣
↪process and involves a Fourier transform along the last axis of\nthe windowed␣
↪data.\n\nParameters\n----------\nwin_fnc : Union[str, Tuple[str, float]]\n    The␣
↪window to use before performing the FFT, by default (\"kaiser\", 14)\ndetrend :␣
↪Union[str, None]\n    Type of detrending to apply before performing FFT, by␣
↪default linear\n    detrend. Setting to None will not apply any detrending to the␣
↪data prior\n    to the FFT\nworkers : int\n    The number of CPUs to use, by␣
↪default max - 2\n\nExamples\n--------\nThis example will get periodic decimated␣
↪data, perfrom windowing and run the\nFourier transform on the windowed data.\n\n..
↪ plot::\n    :width: 90%\n\n    >>> import matplotlib.pyplot as plt\n    >>>␣
↪import numpy as np\n    >>> from resistics.testing import decimated_data_periodic\
↪n    >>> from resistics.window import WindowSetup, Windower\n    >>> from␣
↪resistics.spectra import FourierTransform\n    >>> frequencies = {\"chan1\": [870,
↪ 590, 110, 32, 12], \"chan2\": [480, 375, 210, 60, 45]}\n    >>> dec_data =␣
↪decimated_data_periodic(frequencies, fs=128)\n    >>> dec_data.metadata.chans\n  ␣
↪ ['chan1', 'chan2']\n    >>> print(dec_data.to_string())\n    <class 'resistics.
↪decimate.DecimatedData'>\n               fs        dt  n_samples           first_
↪time                      last_time\n    level\n    0      2048.0  0.000488    ␣
↪ 16384  2021-01-01 00:00:00  2021-01-01 00:00:07.99951171875\n    1       512.0 ␣
↪0.001953       4096  2021-01-01 00:00:00    2021-01-01 00:00:07.998046875\n    2 ␣
↪     128.0  0.007812       1024  2021-01-01 00:00:00      2021-01-01 00:00:07.
↪9921875\n\n    Perform the windowing\n\n    >>> win_params = WindowSetup().
↪run(dec_data.metadata.n_levels, dec_data.metadata.fs)\n    >>> win_data =␣
↪Windower().run(dec_data.metadata.first_time, win_params, dec_data)\n\n    And␣
↪then the Fourier transform. By default, the data will be (linearly)\n   ␣
↪detrended and mutliplied by a Kaiser window prior to the Fourier\n    transform\n\
↪n    >>> spec_data = FourierTransform().run(win_data)\n\n    For plotting of␣
↪magnitude, let's stack the spectra\n\n    >>> freqs_0 = spec_data.metadata.levels_
↪metadata[0].freqs\n    >>> data_0 = np.absolute(spec_data.data[0]).mean(axis=0)\n␣
↪   >>> freqs_1 = spec_data.metadata.levels_metadata[1].freqs\n    >>> data_1 = np.
↪absolute(spec_data.data[1]).mean(axis=0)\n    >>> freqs_2 = spec_data.metadata.
↪levels_metadata[2].freqs\n    >>> data_2 = np.absolute(spec_data.data[2]).
↪mean(axis=0)\n\n    Now plot\n\n    >>> plt.subplot(3,1,1) # doctest: +SKIP\n    >
↪>> plt.plot(freqs_0, data_0[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_0, data_0[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
```

(continues on next page)

(continued from previous page)

```
↪  >>> plt.title(\"Decimation level 0\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.subplot(3,1,2) # doctest: +SKIP\n    >>> plt.
↪plot(freqs_1, data_1[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_1, data_1[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 1\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.subplot(3,1,3) # doctest: +SKIP\n    >>> plt.
↪plot(freqs_2, data_2[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_2, data_2[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 2\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.xlabel(\"Frequency\") # doctest: +SKIP\n    >>> plt.
↪tight_layout() # doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "win_fnc": {
              "title": "Win Fnc",
              "default": [
                 "kaiser",
                 14
              ],
              "anyOf": [
                 {
                    "type": "string"
                 },
                 {
                    "type": "array",
                    "minItems": 2,
                    "maxItems": 2,
                    "items": [
                       {
                          "type": "string"
                       },
                       {
                          "type": "number"
                       }
                    ]
                 }
              ]
           },
           "detrend": {
              "title": "Detrend",
              "default": "linear",
              "type": "string"
           },
           "workers": {
              "title": "Workers",
              "default": -2,
              "type": "integer"
           }
```

(continues on next page)

(continued from previous page)


```
        }
     },
     "SpectraProcess": {
        "title": "SpectraProcess",
        "description": "Parent class for spectra processes",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           }
        }
     },
     "EvaluationFreqs": {
        "title": "EvaluationFreqs",
        "description": "Calculate the spectra values at the evaluation frequencies\
↪n\nThis is done using linear interpolation in the complex domain\n\nExample\n-----
↪--\nThe example will show interpolation to evaluation frequencies on a very\
↪nsimple example. Begin by generating some example spectra data.\n\n>>> from␣
↪resistics.decimate import DecimationSetup\n>>> from resistics.spectra import␣
↪EvaluationFreqs\n>>> from resistics.testing import spectra_data_basic\n>>> spec_
↪data = spectra_data_basic()\n>>> spec_data.metadata.n_levels\n1\n>>> spec_data.
↪metadata.chans\n['chan1']\n>>> spec_data.metadata.levels_metadata[0].summary()\n{\
↪n    'fs': 180.0,\n    'n_wins': 2,\n    'win_size': 20,\n    'olap_size': 5,\n  ␣
↪ 'index_offset': 0,\n    'n_freqs': 10,\n    'freqs': [0.0, 10.0, 20.0, 30.0, 40.
↪0, 50.0, 60.0, 70.0, 80.0, 90.0]\n}\n\nThe spectra data has only a single channel␣
↪and a single level which has 2\nwindows. Now define our evaluation frequencies.\n\
↪n>>> eval_freqs = [1, 12, 23, 34, 45, 56, 67, 78, 89]\n>>> dec_setup =␣
↪DecimationSetup(n_levels=1, per_level=9, eval_freqs=eval_freqs)\n>>> dec_params =␣
↪dec_setup.run(spec_data.metadata.fs[0])\n>>> dec_params.summary()\n{\n    'fs':␣
↪180.0,\n    'n_levels': 1,\n    'per_level': 9,\n    'min_samples': 256,\n
↪'eval_freqs': [1.0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0],\n    'dec_
↪factors': [1],\n    'dec_increments': [1],\n    'dec_fs': [180.0]\n}\n\nNow␣
↪calculate the spectra at the evaluation frequencies\n\n>>> eval_data =␣
↪EvaluationFreqs().run(dec_params, spec_data)\n>>> eval_data.metadata.levels_
↪metadata[0].summary()\n{\n    'fs': 180.0,\n    'n_wins': 2,\n    'win_size': 20,\
↪n    'olap_size': 5,\n    'index_offset': 0,\n    'n_freqs': 9,\n    'freqs': [1.
↪0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0]\n}\n\nTo double check␣
↪everything is as expected, let's compare the data. Comparing\nwindow 1 gives\n\n>>
↪> print(spec_data.data[0][0, 0])\n[0.+0.j 1.+1.j 2.+2.j 3.+3.j 4.+4.j 5.+5.j 6.+6.
↪j 7.+7.j 8.+8.j 9.+9.j]\n>>> print(eval_data.data[0][0, 0])\n[0.1+0.1j 1.2+1.2j 2.
↪3+2.3j 3.4+3.4j 4.5+4.5j 5.6+5.6j 6.7+6.7j 7.8+7.8j\n 8.9+8.9j]\n\nAnd window 2\n\
↪n>>> print(spec_data.data[0][1, 0])\n[-1. +1.j  0. +2.j  1. +3.j  2. +4.j  3. +5.
↪j  4. +6.j  5. +7.j  6. +8.j\n  7. +9.j  8.+10.j]\n>>> print(eval_data.data[0][1,␣
↪0])\n[-0.9+1.1j  0.2+2.2j  1.3+3.3j  2.4+4.4j  3.5+5.5j  4.6+6.6j  5.7+7.7j\n  6.
↪8+8.8j  7.9+9.9j]",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           }
```


(continues on next page)

(continued from previous page)

```
        }
     },
     "Calibrator": {
        "title": "Calibrator",
        "description": "Parent class for a calibrator",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "chans": {
              "title": "Chans",
              "type": "array",
              "items": {
                 "type": "string"
              }
           }
        }
     },
     "TransferFunction": {
        "title": "TransferFunction",
        "description": "Define a generic transfer function\n\nThis class is a␣
↪describes generic transfer function, including:\n\n- The output channels for the␣
↪transfer function\n- The input channels for the transfer function\n- The cross␣
↪channels for the transfer function\n\nThe cross channels are the channels that␣
↪will be used to calculate out the\ncross powers for the regression.\n\nThis␣
↪generic parent class has no implemented plotting function. However,\nchild␣
↪classes may have a plotting function as different transfer functions\nmay need␣
↪different types of plots.\n\n.. note::\n\n    Users interested in writing a␣
↪custom transfer function should inherit\n    from this generic Transfer function\
↪n\nSee Also\n--------\nImpandanceTensor : Transfer function for the MT impedance␣
↪tensor\nTipper : Transfer function for the MT tipper\n\nExamples\n--------\nA␣
↪generic example\n\n>>> from resistics.transfunc import TransferFunction\n>>> tf =␣
↪TransferFunction(variation=\"example\", out_chans=[\"bye\", \"see you\", \"ciao\
↪"], in_chans=[\"hello\", \"hi_there\"])\n>>> print(tf.to_string())\n| bye      | ␣
↪ | bye_hello         bye_hi_there      | | hello    |\n| see you  | = | see you_
↪hello     see you_hi_there  | | hi_there |\n| ciao     |   | ciao_hello       ␣
↪ciao_hi_there     |\n\nCombining the impedance tensor and the tipper into one␣
↪TransferFunction\n\n>>> tf = TransferFunction(variation=\"combined\", out_chans=[\
↪"Ex\", \"Ey\"], in_chans=[\"Hx\", \"Hy\", \"Hz\"])\n>>> print(tf.to_string())\n|␣
↪Ex |   | Ex_Hx Ex_Hy Ex_Hz | | Hx |\n| Ey | = | Ey_Hx Ey_Hy Ey_Hz | | Hy |\n     ␣
↪                          | Hz |",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "variation": {
              "title": "Variation",
              "default": "generic",
```

(continues on next page)

(continued from previous page)

```
                "maxLength": 16,
                "type": "string"
            },
            "out_chans": {
                "title": "Out Chans",
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "in_chans": {
                "title": "In Chans",
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "cross_chans": {
                "title": "Cross Chans",
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            "n_out": {
                "title": "N Out",
                "type": "integer"
            },
            "n_in": {
                "title": "N In",
                "type": "integer"
            },
            "n_cross": {
                "title": "N Cross",
                "type": "integer"
            }
        },
        "required": [
            "out_chans",
            "in_chans"
        ]
    },
    "RegressionPreparerGathered": {
        "title": "RegressionPreparerGathered",
        "description": "Regression preparer for gathered data\n\nIn nearly all␣
↪cases, this is the regresson preparer to use. As input, it\nrequires GatheredData.
↪",
        "type": "object",
        "properties": {
            "name": {
                "title": "Name",
                "type": "string"
```

(continues on next page)

(continued from previous page)

```
            }
         }
      },
      "Solver": {
         "title": "Solver",
         "description": "General resistics solver",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            }
         }
      },
      "Configuration": {
         "title": "Configuration",
         "description": "The resistics configuration\n\nConfiguration can be␣
↪customised by users who wish to use their own custom\nprocesses for certain steps.
↪ In most cases, customisation will be for:\n\n- Implementing new time data␣
↪readers\n- Implementing readers for specific calibration formats\n- Implementing␣
↪time data processors\n- Implementing spectra data processors\n- Adding new␣
↪features to extract from the data\n\nExamples\n--------\nFrequently,␣
↪configuration will be used to change data readers.\n\n>>> from resistics.letsgo␣
↪import get_default_configuration\n>>> config = get_default_configuration()\n>>>␣
↪config.name\n'default'\n>>> for tr in config.time_readers:\n...     tr.summary()\n
↪{\n    'name': 'TimeReaderAscii',\n    'apply_scalings': True,\n    'extension':
↪'.txt',\n    'delimiter': None,\n    'n_header': 0\n}\n{\n    'name':
↪'TimeReaderNumpy',\n    'apply_scalings': True,\n    'extension': '.npy'\n}\n>>>␣
↪config.sensor_calibrator.summary()\n{\n    'name': 'SensorCalibrator',\n    'chans
↪': None,\n    'readers': [\n        {\n            'name': 'SensorCalibrationJSON
↪',\n            'extension': '.json',\n            'file_str': 'IC_$sensor
↪$extension'\n        }\n    ]\n}\n\nTo change these, it's best to make a new␣
↪configuration with a different name\n\n>>> from resistics.letsgo import␣
↪Configuration\n>>> from resistics.time import TimeReaderNumpy\n>>> config =␣
↪Configuration(name=\"myconfig\", time_readers=[TimeReaderNumpy(apply_
↪scalings=False)])\n>>> for tr in config.time_readers:\n...     tr.summary()\n{\n ␣
↪  'name': 'TimeReaderNumpy',\n    'apply_scalings': False,\n    'extension': '.npy
↪'\n}\n\nOr for the sensor calibration\n\n>>> from resistics.calibrate import␣
↪SensorCalibrator, SensorCalibrationTXT\n>>> calibration_reader =␣
↪SensorCalibrationTXT(file_str=\"lemi120_IC_$serial$extension\")\n>>> calibrator =␣
↪SensorCalibrator(chans=[\"Hx\", \"Hy\", \"Hz\"], readers=[calibration_reader])\n>>
↪> config = Configuration(name=\"myconfig\", sensor_calibrator=calibrator)\n>>>␣
↪config.sensor_calibrator.summary()\n{\n    'name': 'SensorCalibrator',\n    'chans
↪': ['Hx', 'Hy', 'Hz'],\n    'readers': [\n        {\n            'name':
↪'SensorCalibrationTXT',\n            'extension': '.TXT',\n            'file_str
↪': 'lemi120_IC_$serial$extension'\n        }\n    ]\n}\n\nAs a final example,␣
↪create a configuration which used targetted windowing\ninstead of specified␣
↪window sizes\n\n>>> from resistics.letsgo import Configuration\n>>> from␣
↪resistics.window import WindowerTarget\n>>> config = Configuration(name=\"window_
↪target\", windower=WindowerTarget(target=500))\n>>> config.name\n'window_target'\
↪n>>> config.windower.summary()\n{\n    'name': 'WindowerTarget',\n    'target':␣
↪500,\n    'min_size': 64,\n    'olap_proportion': 0.25\n}",
```

(continues on next page)

(continued from previous page)


```
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "time_readers": {
              "title": "Time Readers",
              "default": [
                 {
                    "name": "TimeReaderAscii",
                    "apply_scalings": true,
                    "extension": ".txt",
                    "delimiter": null,
                    "n_header": 0
                 },
                 {
                    "name": "TimeReaderNumpy",
                    "apply_scalings": true,
                    "extension": ".npy"
                 }
              ],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/TimeReader"
              }
           },
           "time_processors": {
              "title": "Time Processors",
              "default": [
                 {
                    "name": "InterpolateNans"
                 },
                 {
                    "name": "RemoveMean"
                 }
              ],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/TimeProcess"
              }
           },
           "dec_setup": {
              "title": "Dec Setup",
              "default": {
                 "name": "DecimationSetup",
                 "n_levels": 8,
                 "per_level": 5,
                 "min_samples": 256,
                 "div_factor": 2,
                 "eval_freqs": null
              },
```


(continues on next page)

(continued from previous page)


```
            "allOf": [
               {
                  "$ref": "#/definitions/DecimationSetup"
               }
            ]
         },
         "decimator": {
            "title": "Decimator",
            "default": {
               "name": "Decimator",
               "resample": true,
               "max_single_factor": 3
            },
            "allOf": [
               {
                  "$ref": "#/definitions/Decimator"
               }
            ]
         },
         "win_setup": {
            "title": "Win Setup",
            "default": {
               "name": "WindowSetup",
               "min_size": 128,
               "min_olap": 32,
               "win_factor": 4,
               "olap_proportion": 0.25,
               "min_n_wins": 5,
               "win_sizes": null,
               "olap_sizes": null
            },
            "allOf": [
               {
                  "$ref": "#/definitions/WindowSetup"
               }
            ]
         },
         "windower": {
            "title": "Windower",
            "default": {
               "name": "Windower"
            },
            "allOf": [
               {
                  "$ref": "#/definitions/Windower"
               }
            ]
         },
         "fourier": {
            "title": "Fourier",
            "default": {
               "name": "FourierTransform",
```


(continues on next page)

(continued from previous page)


```
                "win_fnc": [
                    "kaiser",
                    14
                ],
                "detrend": "linear",
                "workers": -2
            },
            "allOf": [
                {
                    "$ref": "#/definitions/FourierTransform"
                }
            ]
        },
        "spectra_processors": {
            "title": "Spectra Processors",
            "default": [],
            "type": "array",
            "items": {
                "$ref": "#/definitions/SpectraProcess"
            }
        },
        "evals": {
            "title": "Evals",
            "default": {
                "name": "EvaluationFreqs"
            },
            "allOf": [
                {
                    "$ref": "#/definitions/EvaluationFreqs"
                }
            ]
        },
        "sensor_calibrator": {
            "title": "Sensor Calibrator",
            "default": {
                "name": "SensorCalibrator",
                "chans": null,
                "readers": [
                    {
                        "name": "SensorCalibrationJSON",
                        "extension": ".json",
                        "file_str": "IC_$sensor$extension"
                    }
                ]
            },
            "allOf": [
                {
                    "$ref": "#/definitions/Calibrator"
                }
            ]
        },
        "tf": {
```


(continues on next page)

(continued from previous page)


```
            "title": "Tf",
            "default": {
               "name": "ImpedanceTensor",
               "variation": "default",
               "out_chans": [
                  "Ex",
                  "Ey"
               ],
               "in_chans": [
                  "Hx",
                  "Hy"
               ],
               "cross_chans": [
                  "Hx",
                  "Hy"
               ],
               "n_out": 2,
               "n_in": 2,
               "n_cross": 2
            },
            "allOf": [
               {
                  "$ref": "#/definitions/TransferFunction"
               }
            ]
         },
         "regression_preparer": {
            "title": "Regression Preparer",
            "default": {
               "name": "RegressionPreparerGathered"
            },
            "allOf": [
               {
                  "$ref": "#/definitions/RegressionPreparerGathered"
               }
            ]
         },
         "solver": {
            "title": "Solver",
            "default": {
               "name": "SolverScikitTheilSen",
               "fit_intercept": false,
               "normalize": false,
               "n_jobs": -2,
               "max_subpopulation": 2000,
               "n_subsamples": null
            },
            "allOf": [
               {
                  "$ref": "#/definitions/Solver"
               }
            ]
```


(continues on next page)

(continued from previous page)

```
            }
         },
         "required": [
            "name"
         ]
      }
   }
}
```

field proj: *Project* [Required]

The project

field config: *Configuration* [Required]

The configuration for processing

resistics.letsgo.**load**(*dir_path: Path | str*, *config:* Configuration | *None* = *None*) → *ResisticsEnvironment*

Load an existing project into a ResisticsEnvironment

Parameters

- **dir_path** (*Union[Path, str]*) – The project directory
- **config** (*Optional[Configuration], optional*) – A configuration of parameters to use

Returns

The ResisticsEnvironment combining a project and a configuration

Return type

ResisticsEnvironment

Raises

ProjectLoadError – If the loading failed

resistics.letsgo.**reload**(*resenv:* ResisticsEnvironment) → *ResisticsEnvironment*

Reload the project in the ResisticsEnvironment

Parameters

resenv (ResisticsEnvironment) – The current resistics environment

Returns

The resistics environment with the project reloaded

Return type

ResisticsEnvironment

resistics.letsgo.**run_time_processors**(*config:* Configuration, *time_data:* TimeData) → *TimeData*

Process time data

Parameters

- **config** (Configuration) – The configuration
- **time_data** (TimeData) – Time data to process

Returns

Process time data

Return type

TimeData

resistics.letsgo.**run_decimation**(*config:* Configuration, *time_data:* TimeData, *dec_params:* DecimationParameters | *None* = *None*) → *DecimatedData*

Decimate TimeData

Parameters

- **config** (Configuration) – The configuration
- **time_data** (TimeData) – Time data to decimate
- **dec_params** (DecimationParameters) – Number of levels, decimation factors etc.

Returns
Decimated time data

Return type
DecimatedData

resistics.letsgo.**run_windowing**(*config:* Configuration, *ref_time:* HighResDateTime, *dec_data:* DecimatedData) → *WindowedData*

Window time data

Parameters

- **config** (Configuration) – The configuration
- **ref_time** (HighResDateTime) – The reference time
- **dec_data** (DecimatedData) – Decimated data to window

Returns
The windowed data

Return type
WindowedData

resistics.letsgo.**run_fft**(*config:* Configuration, *win_data:* WindowedData) → *SpectraData*

Run Fourier transform

Parameters

- **config** (Configuration) – The configuration
- **win_data** (WindowedData) – Windowed data

Returns
Fourier transformed windowed data

Return type
SpectraData

resistics.letsgo.**run_spectra_processors**(*config:* Configuration, *spec_data:* SpectraData) → *SpectraData*

Run any spectra processors

Parameters

- **config** (Configuration) – The configuration
- **spec_data** (SpectraData) – Spectra data

Returns
Processed spectra data

Return type
SpectraData

resistics.letsgo.**run_evals**(*config:* Configuration, *dec_params:* DecimationParameters, *spec_data:* SpectraData) → *SpectraData*

Run evaluation frequency data calculator

Parameters

- **config** (Configuration) – The configuration
- **dec_params** (DecimationParameters) – Decimation parameters with the evaluation frequencies
- **spec_data** (SpectraData) – The spectra data

Returns
Spectra data at evaluation frequencies

Return type
SpectraData

resistics.letsgo.**run_sensor_calibration**(*config:* Configuration, *calibration_path:* *Path*, *spec_data:* SpectraData) → *SpectraData*

Run calibration

Parameters

- **config** (Configuration) – The configuration
- **calibration_path** (*Path*) – Path to calibration data
- **spec_data** (SpectraData) – Spectra data to calibrate

Returns
Calibrated spectra data

Return type
SpectraData

resistics.letsgo.**run_regression_preparer**(*config:* Configuration, *gathered_data:* GatheredData) → *RegressionInputData*

Prepare linear regression data

Parameters

- **config** (Configuration) – The configuration
- **gathered_data** (GatheredData) – Gathered data to input into the regression

Returns
Regression inputs for all evaluation frequencies

Return type
RegressionInputData

resistics.letsgo.**run_solver**(*config:* Configuration, *reg_data:* RegressionInputData) → *Solution*

Run the regression solver

Parameters

- **config** (Configuration) – The configuration
- **reg_data** (RegressionInputData) – The regression input data

Returns
Transfer function estimate

Return type
Solution

resistics.letsgo.**quick_read**(*dir_path: Path*, *config:* Configuration | *None* = *None*, *from_time: str* | *Timestamp* | *datetime* | *None* = *None*, *to_time: str* | *Timestamp* | *datetime* | *None* = *None*, *from_sample: None* = *None*, *to_sample: None* = *None*) → *TimeData*

Read time data folder

Parameters

- **dir_path** (*Path*) – The directory path to read
- **config** (*Optional[Configuration], optional*) – Configuration with appropriate readers, by default None.
- **from_time** (*Union[DateTimeLike, None], optional*) – Timestamp to read from, by default None
- **to_time** (*Union[DateTimeLike, None], optional*) – Timestamp to read to, by default None
- **from_sample** (*Union[int, None], optional*) – Sample to read from, by default None
- **to_sample** (*Union[int, None], optional*) – Sample to read to, by default None

Returns
The read TimeData

Return type
TimeData

Raises
TimeDataReadError – If unable to read data

resistics.letsgo.**quick_view**(*dir_path: Path*, *config:* Configuration | *None* = *None*, *decimate: bool* = *False*, *max_pts: int* = *10000*)

Quick plotting of time data

Parameters

- **dir_path** (*Path*) – The directory path
- **config** (*Optional[Configuration], optional*) – The configuration with the required time readers, by default None
- **decimate** (*bool, optional*) – Boolean flag for decimating, by default False
- **max_pts** (*Optional[int], optional*) – Max points in lttb decimation, by default 10_000

Returns
Plotly figure

Return type
go.Figure

Raises
ValueError – If time data fails reading

resistics.letsgo.**quick_spectra**(*dir_path: Path*, *config:* Configuration | *None* = *None*) → *SpectraData*

Quick plotting of time data

Parameters

- **dir_path** (*Path*) – The directory path

- **config** (*Optional[Configuration], optional*) – The configuration with the required time readers, by default None

Returns
The spectra data

Return type
SpectraData

Raises
ValueError – If time data fails reading

resistics.letsgo.**quick_tf**(*dir_path: Path*, *config:* Configuration | *None* = *None*, *calibration_path: Path* | *None* = *None*) → *Solution*

Quickly calculate out a transfer function for time data in its own directory

Parameters

- **dir_path** (*Path*) – The directory path
- **config** (*Optional[Configuration], optional*) – A configuration instance, by default None
- **calibration_path** (*Optional[Path], optional*) – The path to the calibration data, by default None

Returns
Transfer function estimate

Return type
Solution

resistics.letsgo.**profile_windowing**(*dir_path: Path*, *config:* Configuration | *None* = *None*, *ref_time: str* | *Timestamp* | *datetime* | *None* = *None*) → Dict[int, DataFrame]

Profile windowing for a measurement

This function will return window tables for each decimation level. Note that any time processses are run first in case these changes the start or end time of the data.

Parameters

- **dir_path** (*Path*) – Directory path of the time data
- **config** (*Optional[Configuration], optional*) – Configuration to use, by default None. If not provided, the default configuration will be used.
- **ref_time** (*Optional[DateTimeLike], optional*) – A reference time to perform windowing against, by default None. If not provided, the start time of the recording will be used as the reference time.

Returns
Mapping from decimation level to a pandas DataFrame of the windows for the decimation level

Return type
Dict[int, pd.DataFrame]

resistics.letsgo.**process_time**(*resenv:* ResisticsEnvironment, *site_name: str*, *meas_name: str*, *out_site: str*, *out_meas: str*, *input_from_time: str* | *Timestamp* | *datetime* | *None* = *None*, *input_to_time: str* | *Timestamp* | *datetime* | *None* = *None*, *output_from_time: str* | *Timestamp* | *datetime* | *None* = *None*, *output_to_time: str* | *Timestamp* | *datetime* | *None* = *None*) → None

Process time data and save as a new measurement

This is useful when resampling data to use with other measurements

Parameters

- **resenv** (ResisticsEnvironment) – The resistics environment
- **site_name** (*str*) – The name of the site with the data to process
- **meas_name** (*str*) – The name of the measurement to process
- **out_site** (*str*) – The site to output the data to
- **out_meas** (*str*) – The name of the measurement to output the data to
- **input_from_time** (*Optional[DateTimeLike], optional*) – Time to read data from for the input data, by default None. If None, the first time of the time series data is used.
- **input_to_time** (*Optional[DateTimeLike], optional*) – Time to read data to for the input data, by default None. If None, the last time of the input time series data is used.
- **output_from_time** (*Optional[DateTimeLike], optional*) – Time to output data from, by default None. If None, the first time of input data is used.
- **output_to_time** (*Optional[DateTimeLike], optional*) – Time to output data to, by default None. If None, the last time of the input data is used.

resistics.letsgo.**process_time_to_evals**(*resenv: ResisticsEnvironment*, *site_name: str*, *meas_name: str*) → None

Process from time data to Fourier spectra

Parameters

- **resenv** (ResisticsEnvironment) – The resistics environment containing the project and configuration
- **site_name** (*str*) – The name of the site
- **meas_name** (*str*) – The name of the measurement to process

resistics.letsgo.**process_evals_to_tf**(*resenv: ResisticsEnvironment*, *fs: float*, *out_site: str*, *in_site: str | None = None*, *cross_site: str | None = None*, *masks: Dict[str, str] | None = None*, *postfix: str | None = None*) → *Solution*

Process spectra to transfer functions

Parameters

- **resenv** (ResisticsEnvironment) – The resistics environment
- **fs** (*float*) – The sampling frequency to process
- **out_site** (*str*) – The name of the output site
- **in_site** (*Optional[str], optional*) – The name of the input site, by default None. This should be used for intersite processing
- **cross_site** (*Optional[str], optional*) – The name of the cross site, by default None. This is usually the site to use as the remote reference.
- **masks** (*Optional[Dict[str, str]], optional*) – Any masks to apply, by default None
- **postfix** (*Optional[str]*) – String to add to the end of solution, by default None

Returns

Transfer function estimate

Return type
Solution

resistics.letsgo.**get_solution**(*resenv: ResisticsEnvironment*, *site_name: str*, *config_name: str*, *fs: float*, *tf_name: str*, *tf_var: str*, *postfix: str | None = None*) → *Solution*

Get a solution

Parameters

- **resenv** (*ResisticsEnvironment*) – The resistics environment
- **site_name** (*str*) – The site for which to get the solution
- **config_name** (*str*) – The configuration that was used
- **fs** (*float*) – The sampling frequency
- **tf_name** (*str*) – The transfer function name
- **tf_var** (*str*) – The transfer function variation
- **postfix** (*Optional[str], optional*) – Any postfix on the solution, by default None

Returns
The solution

Return type
Solution

resistics.plot module

Module to help plotting various data

resistics.plot.**lttb_downsample**(*x: ndarray*, *y: ndarray*, *max_pts: int = 5000*) → Tuple[ndarray, ndarray]

Downsample x, y for visualisation

Parameters

- **x** (*np.ndarray*) – x array
- **y** (*np.ndarray*) – y array
- **max_pts** (*int, optional*) – Maximum number of points after downsampling, by default 5000

Returns
(new_x, new_y), the downsampled x and y arrays

Return type
Tuple[np.ndarray, np.ndarray]

Raises
ValueError – If the size of x does not match the size of y

resistics.plot.**apply_lttb**(*data: ndarray*, *max_pts: int | None*) → Tuple[ndarray, ndarray]

A helper function for applying lttb downsampling if max_pts is not None

Parameters

- **data** (*np.ndarray*) – The data to downsample
- **max_pts** (*Union[int, None]*) – The maximum number of points or None. If None, no downsamping is performed

Returns
Indices and data selected for plotting

Return type
Tuple[np.ndarray, np.ndarray]

resistics.plot.**plot_timeline**(*df: DataFrame*, *y_col: str*, *title: str = 'Timeline'*, *ref_time: Timestamp | None = None*) → Figure

Plot a timeline

Parameters

- **df** (*pd.DataFrame*) – DataFrame with the first and last times of the horizontal bars
- **y_col** (*str*) – The column to use for the y axis
- **title** (*str, optional*) – The title for the plot, by default "Timeline"
- **ref_time** (*Optional[pd.Timestamp], optional*) – The reference time, by default None

Returns
Plotly figure

Return type
go.Figure

resistics.plot.**get_calibration_fig**() → Figure

Get a figure for plotting calibration data

Returns
Plotly figure

Return type
go.Figure

resistics.plot.**get_time_fig**(*chans: List[str]*, *y_axis_label: Dict[str, str]*) → Figure

Get a figure for plotting time data

Parameters

- **chans** (*List[str]*) – The channels to plot
- **y_axis_label** (*Dict[str, str]*) – The labels to use for the y axis

Returns
Plotly figure

Return type
go.Figure

resistics.plot.**get_spectra_stack_fig**(*chans: List[str]*, *y_axis_label: Dict[str, str]*) → Figure

Get a figure for plotting spectra stack data

Parameters

- **chans** (*List[str]*) – The channels to plot
- **y_axis_label** (*Dict[str, str]*) – The y axis labels

Returns
Plotly figure

Return type
go.Figure

resistics.plot.**get_spectra_section_fig**(*chans: List[str]*) → Figure

Get figure for plotting spectra sections

Parameters
chans (*List[str]*) – The channels to plot

Returns
Plotly figure

Return type
go.Figure

resistics.project module

Classes and methods to enable a resistics project

A project is an essential element of a resistics environment together with a configuration.

In particular, this module includes the core Project, Site and Measurement clasess and some supporting functions.

resistics.project.**get_calibration_path**(*proj_dir: Path*) → Path

Get the path to the calibration data

resistics.project.**get_meas_time_path**(*proj_dir: Path*, *site_name: str*, *meas_name: str*) → Path

Get path to measurement time data

resistics.project.**get_meas_spectra_path**(*proj_dir: Path*, *site_name: str*, *meas_name: str*, *config_name: str*) → Path

Get path to measurement spectra data

resistics.project.**get_meas_evals_path**(*proj_dir: Path*, *site_name: str*, *meas_name: str*, *config_name: str*) → Path

Get path to measurement evaluation frequency spectra data

resistics.project.**get_meas_features_path**(*proj_dir: Path*, *site_name: str*, *meas_name: str*, *config_name: str*) → Path

Get path to measurement features data

resistics.project.**get_mask_path**(*proj_dir: Path*, *site_name: str*, *config_name: str*) → Path

Get path to mask data

resistics.project.**get_mask_name**(*fs: float*, *mask_name: str*) → str

Get the name of a mask file

resistics.project.**get_results_path**(*proj_dir: Path*, *site_name: str*, *config_name: str*) → Path

Get path to solutions

resistics.project.**get_solution_name**(*fs: float*, *tf_name: str*, *tf_var: str*, *postfix: str | None = None*) → str

Get the name of a solution file

pydantic model resistics.project.**Measurement**

Bases: *ResisticsModel*

Class for interfacing with a measurement

The class holds the original time series metadata and can provide information about other types of data

```
{
   "title": "Measurement",
   "description": "Class for interfacing with a measurement\n\nThe class holds the␣
↪original time series metadata and can provide\ninformation about other types of␣
↪data",
   "type": "object",
   "properties": {
      "site_name": {
         "title": "Site Name",
         "type": "string"
      },
      "dir_path": {
         "title": "Dir Path",
         "type": "string",
         "format": "path"
      },
      "metadata": {
         "$ref": "#/definitions/TimeMetadata"
      },
      "reader": {
         "$ref": "#/definitions/TimeReader"
      }
   },
   "required": [
      "site_name",
      "dir_path",
      "metadata",
      "reader"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
               "title": "Version",
               "type": "string"
            }
         }
      },
      "ChanMetadata": {
```

(continues on next page)

(continued from previous page)

```
        "title": "ChanMetadata",
        "description": "Channel metadata",
        "type": "object",
        "properties": {
           "name": {
              "title": "Name",
              "type": "string"
           },
           "data_files": {
              "title": "Data Files",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "chan_type": {
              "title": "Chan Type",
              "type": "string"
           },
           "chan_source": {
              "title": "Chan Source",
              "type": "string"
           },
           "sensor": {
              "title": "Sensor",
              "default": "",
              "type": "string"
           },
           "serial": {
              "title": "Serial",
              "default": "",
              "type": "string"
           },
           "gain1": {
              "title": "Gain1",
              "default": 1,
              "type": "number"
           },
           "gain2": {
              "title": "Gain2",
              "default": 1,
              "type": "number"
           },
           "scaling": {
              "title": "Scaling",
              "default": 1,
              "type": "number"
           },
           "chopper": {
              "title": "Chopper",
              "default": false,
              "type": "boolean"
```

(continues on next page)

(continued from previous page)


```
            },
            "dipole_dist": {
               "title": "Dipole Dist",
               "default": 1,
               "type": "number"
            },
            "sensor_calibration_file": {
               "title": "Sensor Calibration File",
               "default": "",
               "type": "string"
            },
            "instrument_calibration_file": {
               "title": "Instrument Calibration File",
               "default": "",
               "type": "string"
            }
         },
         "required": [
            "name"
         ]
      },
      "Record": {
         "title": "Record",
         "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
         "type": "object",
         "properties": {
            "time_local": {
               "title": "Time Local",
               "type": "string",
               "format": "date-time"
            },
            "time_utc": {
               "title": "Time Utc",
               "type": "string",
               "format": "date-time"
            },
            "creator": {
               "title": "Creator",
               "type": "object"
            },
            "messages": {
               "title": "Messages",
               "type": "array",
```


(continues on next page)

(continued from previous page)

```
                "items": {
                   "type": "string"
                }
             },
             "record_type": {
                "title": "Record Type",
                "type": "string"
             }
          },
          "required": [
             "creator",
             "messages",
             "record_type"
          ]
       },
       "History": {
          "title": "History",
          "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n     ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n     ␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n      ␣
↪          'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
          "type": "object",
          "properties": {
             "records": {
                "title": "Records",
                "default": [],
                "type": "array",
                "items": {
                   "$ref": "#/definitions/Record"
                }
             }
          }
       },
       "TimeMetadata": {
          "title": "TimeMetadata",
          "description": "Time metadata",
          "type": "object",
          "properties": {
             "file_info": {
                "$ref": "#/definitions/ResisticsFile"
             },
             "fs": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Fs",
            "type": "number"
        },
        "chans": {
            "title": "Chans",
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "n_chans": {
            "title": "N Chans",
            "type": "integer"
        },
        "n_samples": {
            "title": "N Samples",
            "type": "integer"
        },
        "first_time": {
            "title": "First Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
                "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
        },
        "last_time": {
            "title": "Last Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
                "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
        },
        "system": {
            "title": "System",
            "default": "",
            "type": "string"
        },
        "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
        },
        "wgs84_latitude": {
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
        },
        "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
        },
```

(continues on next page)

(continued from previous page)


```
            "easting": {
               "title": "Easting",
               "default": -999.0,
               "type": "number"
            },
            "northing": {
               "title": "Northing",
               "default": -999.0,
               "type": "number"
            },
            "elevation": {
               "title": "Elevation",
               "default": -999.0,
               "type": "number"
            },
            "chans_metadata": {
               "title": "Chans Metadata",
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/ChanMetadata"
               }
            },
            "history": {
               "title": "History",
               "default": {
                  "records": []
               },
               "allOf": [
                  {
                     "$ref": "#/definitions/History"
                  }
               ]
            }
         },
         "required": [
            "fs",
            "chans",
            "n_samples",
            "first_time",
            "last_time",
            "chans_metadata"
         ]
      },
      "TimeReader": {
         "title": "TimeReader",
         "description": "Base class for resistics processes\n\nResistics processes␣
↪perform operations on data (including read and write\noperations). Each time a␣
↪ResisticsProcess child class is run, it should add\na process record to the␣
↪dataset",
         "type": "object",
         "properties": {
            "name": {
```


(continues on next page)

(continued from previous page)


```
                "title": "Name",
                "type": "string"
            },
            "apply_scalings": {
                "title": "Apply Scalings",
                "default": true,
                "type": "boolean"
            },
            "extension": {
                "title": "Extension",
                "type": "string"
            }
        }
    }
}
```


field site_name: *str* **[Required]**

field dir_path: *Path* **[Required]**

field metadata: *TimeMetadata* **[Required]**

field reader: *TimeReader* **[Required]**

property name: *str*

Get the name of the measurement

pydantic model resistics.project.**Site**

Bases: *ResisticsModel*

Class for describing Sites

Note: This should essentially describe a single instrument setup. If the same site is re-occupied later with a different instrument setup, it is suggested to split this into a different site.

```
{
   "title": "Site",
   "description": "Class for describing Sites\n\n.. note::\n\n    This should␣
↪essentially describe a single instrument setup. If the same\n    site is re-
↪occupied later with a different instrument setup, it is\n    suggested to split␣
↪this into a different site.",
   "type": "object",
   "properties": {
      "dir_path": {
         "title": "Dir Path",
         "type": "string",
         "format": "path"
      },
      "measurements": {
         "title": "Measurements",
         "type": "object",
```


(continues on next page)

(continued from previous page)

```
        "additionalProperties": {
           "$ref": "#/definitions/Measurement"
        }
     },
     "begin_time": {
        "title": "Begin Time",
        "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
        "examples": [
           "2021-01-01 00:00:00.000061_035156_250000_000000"
        ]
     },
     "end_time": {
        "title": "End Time",
        "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
        "examples": [
           "2021-01-01 00:00:00.000061_035156_250000_000000"
        ]
     }
  },
  "required": [
     "dir_path",
     "measurements",
     "begin_time",
     "end_time"
  ],
  "definitions": {
     "ResisticsFile": {
        "title": "ResisticsFile",
        "description": "Required information for writing out a resistics file",
        "type": "object",
        "properties": {
           "created_on_local": {
              "title": "Created On Local",
              "type": "string",
              "format": "date-time"
           },
           "created_on_utc": {
              "title": "Created On Utc",
              "type": "string",
              "format": "date-time"
           },
           "version": {
              "title": "Version",
              "type": "string"
           }
        }
     },
     "ChanMetadata": {
        "title": "ChanMetadata",
        "description": "Channel metadata",
        "type": "object",
        "properties": {
```

(continues on next page)

(continued from previous page)

```
            "name": {
               "title": "Name",
               "type": "string"
            },
            "data_files": {
               "title": "Data Files",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "chan_type": {
               "title": "Chan Type",
               "type": "string"
            },
            "chan_source": {
               "title": "Chan Source",
               "type": "string"
            },
            "sensor": {
               "title": "Sensor",
               "default": "",
               "type": "string"
            },
            "serial": {
               "title": "Serial",
               "default": "",
               "type": "string"
            },
            "gain1": {
               "title": "Gain1",
               "default": 1,
               "type": "number"
            },
            "gain2": {
               "title": "Gain2",
               "default": 1,
               "type": "number"
            },
            "scaling": {
               "title": "Scaling",
               "default": 1,
               "type": "number"
            },
            "chopper": {
               "title": "Chopper",
               "default": false,
               "type": "boolean"
            },
            "dipole_dist": {
               "title": "Dipole Dist",
               "default": 1,
```

(continues on next page)

(continued from previous page)

```
            "type": "number"
         },
         "sensor_calibration_file": {
            "title": "Sensor Calibration File",
            "default": "",
            "type": "string"
         },
         "instrument_calibration_file": {
            "title": "Instrument Calibration File",
            "default": "",
            "type": "string"
         }
      },
      "required": [
         "name"
      ]
   },
   "Record": {
      "title": "Record",
      "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...     ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
      "type": "object",
      "properties": {
         "time_local": {
            "title": "Time Local",
            "type": "string",
            "format": "date-time"
         },
         "time_utc": {
            "title": "Time Utc",
            "type": "string",
            "format": "date-time"
         },
         "creator": {
            "title": "Creator",
            "type": "object"
         },
         "messages": {
            "title": "Messages",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
```

(continues on next page)

(continued from previous page)


```
            "record_type": {
               "title": "Record Type",
               "type": "string"
            }
         },
         "required": [
            "creator",
            "messages",
            "record_type"
         ]
      },
      "History": {
         "title": "History",
         "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n     ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n     ␣
↪       'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n      ␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
         "type": "object",
         "properties": {
            "records": {
               "title": "Records",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/Record"
               }
            }
         }
      },
      "TimeMetadata": {
         "title": "TimeMetadata",
         "description": "Time metadata",
         "type": "object",
         "properties": {
            "file_info": {
               "$ref": "#/definitions/ResisticsFile"
            },
            "fs": {
               "title": "Fs",
               "type": "number"
            },
            "chans": {
```


(continues on next page)

(continued from previous page)


```
            "title": "Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "n_chans": {
            "title": "N Chans",
            "type": "integer"
         },
         "n_samples": {
            "title": "N Samples",
            "type": "integer"
         },
         "first_time": {
            "title": "First Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "last_time": {
            "title": "Last Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "system": {
            "title": "System",
            "default": "",
            "type": "string"
         },
         "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
         },
         "wgs84_latitude": {
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
         },
         "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
         },
         "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
```


(continues on next page)

(continued from previous page)


```
            },
            "northing": {
               "title": "Northing",
               "default": -999.0,
               "type": "number"
            },
            "elevation": {
               "title": "Elevation",
               "default": -999.0,
               "type": "number"
            },
            "chans_metadata": {
               "title": "Chans Metadata",
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/ChanMetadata"
               }
            },
            "history": {
               "title": "History",
               "default": {
                  "records": []
               },
               "allOf": [
                  {
                     "$ref": "#/definitions/History"
                  }
               ]
            }
         },
         "required": [
            "fs",
            "chans",
            "n_samples",
            "first_time",
            "last_time",
            "chans_metadata"
         ]
      },
      "TimeReader": {
         "title": "TimeReader",
         "description": "Base class for resistics processes\n\nResistics processes␣
↪perform operations on data (including read and write\noperations). Each time a␣
↪ResisticsProcess child class is run, it should add\na process record to the␣
↪dataset",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "apply_scalings": {
```


(continues on next page)

(continued from previous page)

```
               "title": "Apply Scalings",
               "default": true,
               "type": "boolean"
            },
            "extension": {
               "title": "Extension",
               "type": "string"
            }
         }
      },
      "Measurement": {
         "title": "Measurement",
         "description": "Class for interfacing with a measurement\n\nThe class␣
↪holds the original time series metadata and can provide\ninformation about other␣
↪types of data",
         "type": "object",
         "properties": {
            "site_name": {
               "title": "Site Name",
               "type": "string"
            },
            "dir_path": {
               "title": "Dir Path",
               "type": "string",
               "format": "path"
            },
            "metadata": {
               "$ref": "#/definitions/TimeMetadata"
            },
            "reader": {
               "$ref": "#/definitions/TimeReader"
            }
         },
         "required": [
            "site_name",
            "dir_path",
            "metadata",
            "reader"
         ]
      }
   }
}
```

field dir_path: `Path` **[Required]**

field measurements: `Dict[str, Measurement]` **[Required]**

field begin_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field end_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

property name: str

The Site name

property n_meas: int

Get the number of measurements

fs() → List[float]

Get the sampling frequencies in the Site

get_measurement(*meas_name: str*) → *Measurement*

Get a measurement

get_measurements(*fs: float | None = None*) → Dict[str, *Measurement*]

Get dictionary of measurements with optional filter by sampling frequency

plot() → Figure

Plot the site timeline

to_dataframe() → DataFrame

Get measurements list in a pandas DataFrame

Note: Measurement first and last times are converted to pandas Timestamps as these are more universally useful in a pandas DataFrame. However, this may result in a loss of precision, especially at high sampling frequencies.

Returns

Site measurement DataFrame

Return type

pd.DataFrame

pydantic model resistics.project.**ProjectMetadata**

Bases: *WriteableMetadata*

Project metadata


```
{
   "title": "ProjectMetadata",
   "description": "Project metadata",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "ref_time": {
         "title": "Ref Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
```


(continues on next page)

(continued from previous page)

```
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "location": {
         "title": "Location",
         "default": "",
         "type": "string"
      },
      "country": {
         "title": "Country",
         "default": "",
         "type": "string"
      },
      "year": {
         "title": "Year",
         "default": -999,
         "type": "integer"
      },
      "description": {
         "title": "Description",
         "default": "",
         "type": "string"
      },
      "contributors": {
         "title": "Contributors",
         "default": [],
         "type": "array",
         "items": {
            "type": "string"
         }
      }
   },
   "required": [
      "ref_time"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
```

(continues on next page)

(continued from previous page)

```
                "title": "Version",
                "type": "string"
            }
        }
    }
}
```

field ref_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field location: `str = ''`

field country: `str = ''`

field year: `int = -999`

field description: `str = ''`

field contributors: `List[str] = []`

pydantic model resistics.project.**Project**

Bases: *ResisticsModel*

Class to describe a resistics project

The resistics Project Class connects all resistics data. It is an essential part of processing data with resistics.

Resistics projects are in directory with several sub-directories. Project metadata is saved in the resistics.json file at the top level directory.

```
{
   "title": "Project",
   "description": "Class to describe a resistics project\n\nThe resistics Project␣
↪Class connects all resistics data. It is an essential\npart of processing data␣
↪with resistics.\n\nResistics projects are in directory with several sub-
↪directories. Project\nmetadata is saved in the resistics.json file at the top␣
↪level directory.",
   "type": "object",
   "properties": {
      "dir_path": {
         "title": "Dir Path",
         "type": "string",
         "format": "path"
      },
      "begin_time": {
         "title": "Begin Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
```

(continues on next page)

(continued from previous page)


```
      },
      "end_time": {
         "title": "End Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "metadata": {
         "$ref": "#/definitions/ProjectMetadata"
      },
      "sites": {
         "title": "Sites",
         "default": {},
         "type": "object",
         "additionalProperties": {
            "$ref": "#/definitions/Site"
         }
      }
   },
   "required": [
      "dir_path",
      "begin_time",
      "end_time",
      "metadata"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
               "title": "Version",
               "type": "string"
            }
         }
      },
      "ProjectMetadata": {
         "title": "ProjectMetadata",
         "description": "Project metadata",
         "type": "object",
```


(continues on next page)

(continued from previous page)


```
        "properties": {
          "file_info": {
            "$ref": "#/definitions/ResisticsFile"
          },
          "ref_time": {
            "title": "Ref Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
              "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
          },
          "location": {
            "title": "Location",
            "default": "",
            "type": "string"
          },
          "country": {
            "title": "Country",
            "default": "",
            "type": "string"
          },
          "year": {
            "title": "Year",
            "default": -999,
            "type": "integer"
          },
          "description": {
            "title": "Description",
            "default": "",
            "type": "string"
          },
          "contributors": {
            "title": "Contributors",
            "default": [],
            "type": "array",
            "items": {
              "type": "string"
            }
          }
        },
        "required": [
          "ref_time"
        ]
      },
      "ChanMetadata": {
        "title": "ChanMetadata",
        "description": "Channel metadata",
        "type": "object",
        "properties": {
          "name": {
            "title": "Name",
            "type": "string"
```


(continues on next page)

(continued from previous page)

```
            },
            "data_files": {
               "title": "Data Files",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "chan_type": {
               "title": "Chan Type",
               "type": "string"
            },
            "chan_source": {
               "title": "Chan Source",
               "type": "string"
            },
            "sensor": {
               "title": "Sensor",
               "default": "",
               "type": "string"
            },
            "serial": {
               "title": "Serial",
               "default": "",
               "type": "string"
            },
            "gain1": {
               "title": "Gain1",
               "default": 1,
               "type": "number"
            },
            "gain2": {
               "title": "Gain2",
               "default": 1,
               "type": "number"
            },
            "scaling": {
               "title": "Scaling",
               "default": 1,
               "type": "number"
            },
            "chopper": {
               "title": "Chopper",
               "default": false,
               "type": "boolean"
            },
            "dipole_dist": {
               "title": "Dipole Dist",
               "default": 1,
               "type": "number"
            },
            "sensor_calibration_file": {
```

(continues on next page)

(continued from previous page)


```
            "title": "Sensor Calibration File",
            "default": "",
            "type": "string"
         },
         "instrument_calibration_file": {
            "title": "Instrument Calibration File",
            "default": "",
            "type": "string"
         }
      },
      "required": [
         "name"
      ]
   },
   "Record": {
      "title": "Record",
      "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
      "type": "object",
      "properties": {
         "time_local": {
            "title": "Time Local",
            "type": "string",
            "format": "date-time"
         },
         "time_utc": {
            "title": "Time Utc",
            "type": "string",
            "format": "date-time"
         },
         "creator": {
            "title": "Creator",
            "type": "object"
         },
         "messages": {
            "title": "Messages",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "record_type": {
            "title": "Record Type",
            "type": "string"
```


(continues on next page)

(continued from previous page)

```
            }
         },
         "required": [
            "creator",
            "messages",
            "record_type"
         ]
      },
      "History": {
         "title": "History",
         "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n     ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n     ␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n      ␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
         "type": "object",
         "properties": {
            "records": {
               "title": "Records",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/Record"
               }
            }
         }
      },
      "TimeMetadata": {
         "title": "TimeMetadata",
         "description": "Time metadata",
         "type": "object",
         "properties": {
            "file_info": {
               "$ref": "#/definitions/ResisticsFile"
            },
            "fs": {
               "title": "Fs",
               "type": "number"
            },
            "chans": {
               "title": "Chans",
               "type": "array",
               "items": {
```

(continues on next page)

(continued from previous page)


```
            "type": "string"
         }
      },
      "n_chans": {
         "title": "N Chans",
         "type": "integer"
      },
      "n_samples": {
         "title": "N Samples",
         "type": "integer"
      },
      "first_time": {
         "title": "First Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "last_time": {
         "title": "Last Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "system": {
         "title": "System",
         "default": "",
         "type": "string"
      },
      "serial": {
         "title": "Serial",
         "default": "",
         "type": "string"
      },
      "wgs84_latitude": {
         "title": "Wgs84 Latitude",
         "default": -999.0,
         "type": "number"
      },
      "wgs84_longitude": {
         "title": "Wgs84 Longitude",
         "default": -999.0,
         "type": "number"
      },
      "easting": {
         "title": "Easting",
         "default": -999.0,
         "type": "number"
      },
      "northing": {
         "title": "Northing",
```


(continues on next page)

(continued from previous page)


```
                "default": -999.0,
                "type": "number"
            },
            "elevation": {
                "title": "Elevation",
                "default": -999.0,
                "type": "number"
            },
            "chans_metadata": {
                "title": "Chans Metadata",
                "type": "object",
                "additionalProperties": {
                    "$ref": "#/definitions/ChanMetadata"
                }
            },
            "history": {
                "title": "History",
                "default": {
                    "records": []
                },
                "allOf": [
                    {
                        "$ref": "#/definitions/History"
                    }
                ]
            }
        },
        "required": [
            "fs",
            "chans",
            "n_samples",
            "first_time",
            "last_time",
            "chans_metadata"
        ]
    },
    "TimeReader": {
        "title": "TimeReader",
        "description": "Base class for resistics processes\n\nResistics processes␣
↪perform operations on data (including read and write\noperations). Each time a␣
↪ResisticsProcess child class is run, it should add\na process record to the␣
↪dataset",
        "type": "object",
        "properties": {
            "name": {
                "title": "Name",
                "type": "string"
            },
            "apply_scalings": {
                "title": "Apply Scalings",
                "default": true,
                "type": "boolean"
```


(continues on next page)

(continued from previous page)

```
            },
            "extension": {
               "title": "Extension",
               "type": "string"
            }
         }
      },
      "Measurement": {
         "title": "Measurement",
         "description": "Class for interfacing with a measurement\n\nThe class␣
↪holds the original time series metadata and can provide\ninformation about other␣
↪types of data",
         "type": "object",
         "properties": {
            "site_name": {
               "title": "Site Name",
               "type": "string"
            },
            "dir_path": {
               "title": "Dir Path",
               "type": "string",
               "format": "path"
            },
            "metadata": {
               "$ref": "#/definitions/TimeMetadata"
            },
            "reader": {
               "$ref": "#/definitions/TimeReader"
            }
         },
         "required": [
            "site_name",
            "dir_path",
            "metadata",
            "reader"
         ]
      },
      "Site": {
         "title": "Site",
         "description": "Class for describing Sites\n\n.. note::\n\n    This should␣
↪essentially describe a single instrument setup. If the same\n    site is re-
↪occupied later with a different instrument setup, it is\n    suggested to split␣
↪this into a different site.",
         "type": "object",
         "properties": {
            "dir_path": {
               "title": "Dir Path",
               "type": "string",
               "format": "path"
            },
            "measurements": {
               "title": "Measurements",
```

(continues on next page)

(continued from previous page)

```
            "type": "object",
            "additionalProperties": {
               "$ref": "#/definitions/Measurement"
            }
         },
         "begin_time": {
            "title": "Begin Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "end_time": {
            "title": "End Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         }
      },
      "required": [
         "dir_path",
         "measurements",
         "begin_time",
         "end_time"
      ]
   }
}
}
```

field dir_path: *Path* **[Required]**

field begin_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field end_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field metadata: *ProjectMetadata* **[Required]**

field sites: *Dict*[*str*, *Site*] = {}

property n_sites: *int*

The number of sites

fs() → List[float]

Get sampling frequencies in the Project

get_site(*site_name: str*) → *Site*

Get a Site object given the Site name

get_sites(*fs: float | None = None*) → Dict[str, *Site*]

Get sites

Parameters
fs (*Optional[float], optional*) – Filter by sites which have at least a single recording at a specified sampling frequency, by default None

Returns
Dictionary of site name to Site

Return type
Dict[str, *Site*]

get_concurrent(*site_name: str*) → List[*Site*]

Find sites that recorded conscurrently to a specified site

Parameters
site_name (*str*) – Search for sites recording concurrently to this site

Returns
List of Site instances which were recording concurrently

Return type
List[*Site*]

plot() → Figure

Plot a timeline of the project

to_dataframe() → DataFrame

Detail Project recordings in a DataFrame

resistics.regression module

The regression module provides functions and classes for the following:

- Preparing gathered data for regression
- Performing the linear regression

Resistics has built in solvers that use scikit learn models, namely

- Ordinary least squares
- RANSAC
- TheilSen

These will perform well in many scenarios. However, the functionality available in resistics makes it possible to use custom solvers if required.

pydantic model resistics.regression.**RegressionInputMetadata**

Bases: *Metadata*

Metadata for regression input data, mainly to track processing history

```
{
   "title": "RegressionInputMetadata",
   "description": "Metadata for regression input data, mainly to track processing␣
↪history",
   "type": "object",
   "properties": {
      "contributors": {
         "title": "Contributors",
         "type": "object",
         "additionalProperties": {
            "anyOf": [
               {
                  "$ref": "#/definitions/SiteCombinedMetadata"
               },
               {
                  "$ref": "#/definitions/SpectraMetadata"
               }
            ]
         }
      },
      "history": {
         "title": "History",
         "default": {
            "records": []
         },
         "allOf": [
            {
               "$ref": "#/definitions/History"
            }
         ]
      }
   },
   "required": [
      "contributors"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
```

(continues on next page)

(continued from previous page)


```
            "title": "Version",
            "type": "string"
         }
      }
   },
   "Record": {
      "title": "Record",
      "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
      "type": "object",
      "properties": {
         "time_local": {
            "title": "Time Local",
            "type": "string",
            "format": "date-time"
         },
         "time_utc": {
            "title": "Time Utc",
            "type": "string",
            "format": "date-time"
         },
         "creator": {
            "title": "Creator",
            "type": "object"
         },
         "messages": {
            "title": "Messages",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "record_type": {
            "title": "Record Type",
            "type": "string"
         }
      },
      "required": [
         "creator",
         "messages",
         "record_type"
      ]
   },
   "History": {
```


(continues on next page)

(continued from previous page)


```
        "title": "History",
        "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n     ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n     ␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n      ␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
        "type": "object",
        "properties": {
           "records": {
              "title": "Records",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/Record"
              }
           }
        }
     },
     "SiteCombinedMetadata": {
        "title": "SiteCombinedMetadata",
        "description": "Metadata for combined data\n\nCombined metadata stores␣
↪metadata for measurements that are combined from\na single site.",
        "type": "object",
        "properties": {
           "file_info": {
              "$ref": "#/definitions/ResisticsFile"
           },
           "site_name": {
              "title": "Site Name",
              "type": "string"
           },
           "fs": {
              "title": "Fs",
              "type": "number"
           },
           "system": {
              "title": "System",
              "default": "",
              "type": "string"
           },
           "serial": {
              "title": "Serial",
              "default": "",
```


(continues on next page)

(continued from previous page)

```
            "type": "string"
         },
         "wgs84_latitude": {
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
         },
         "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
         },
         "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
         },
         "northing": {
            "title": "Northing",
            "default": -999.0,
            "type": "number"
         },
         "elevation": {
            "title": "Elevation",
            "default": -999.0,
            "type": "number"
         },
         "measurements": {
            "title": "Measurements",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "chans": {
            "title": "Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "n_evals": {
            "title": "N Evals",
            "type": "integer"
         },
         "eval_freqs": {
            "title": "Eval Freqs",
            "type": "array",
            "items": {
               "type": "number"
            }
         },
```

(continues on next page)

(continued from previous page)

```
            "histories": {
               "title": "Histories",
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/History"
               }
            }
         },
         "required": [
            "site_name",
            "fs",
            "chans",
            "n_evals",
            "eval_freqs",
            "histories"
         ]
      },
      "ChanMetadata": {
         "title": "ChanMetadata",
         "description": "Channel metadata",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "data_files": {
               "title": "Data Files",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "chan_type": {
               "title": "Chan Type",
               "type": "string"
            },
            "chan_source": {
               "title": "Chan Source",
               "type": "string"
            },
            "sensor": {
               "title": "Sensor",
               "default": "",
               "type": "string"
            },
            "serial": {
               "title": "Serial",
               "default": "",
               "type": "string"
            },
            "gain1": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Gain1",
            "default": 1,
            "type": "number"
         },
         "gain2": {
            "title": "Gain2",
            "default": 1,
            "type": "number"
         },
         "scaling": {
            "title": "Scaling",
            "default": 1,
            "type": "number"
         },
         "chopper": {
            "title": "Chopper",
            "default": false,
            "type": "boolean"
         },
         "dipole_dist": {
            "title": "Dipole Dist",
            "default": 1,
            "type": "number"
         },
         "sensor_calibration_file": {
            "title": "Sensor Calibration File",
            "default": "",
            "type": "string"
         },
         "instrument_calibration_file": {
            "title": "Instrument Calibration File",
            "default": "",
            "type": "string"
         }
      },
      "required": [
         "name"
      ]
   },
   "SpectraLevelMetadata": {
      "title": "SpectraLevelMetadata",
      "description": "Metadata for spectra of a windowed decimation level",
      "type": "object",
      "properties": {
         "fs": {
            "title": "Fs",
            "type": "number"
         },
         "n_wins": {
            "title": "N Wins",
            "type": "integer"
         },
```

(continues on next page)

(continued from previous page)

```
            "win_size": {
               "title": "Win Size",
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "olap_size": {
               "title": "Olap Size",
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "index_offset": {
               "title": "Index Offset",
               "type": "integer"
            },
            "n_freqs": {
               "title": "N Freqs",
               "type": "integer"
            },
            "freqs": {
               "title": "Freqs",
               "type": "array",
               "items": {
                  "type": "number"
               }
            }
         },
         "required": [
            "fs",
            "n_wins",
            "win_size",
            "olap_size",
            "index_offset",
            "n_freqs",
            "freqs"
         ]
      },
      "SpectraMetadata": {
         "title": "SpectraMetadata",
         "description": "Metadata for spectra data",
         "type": "object",
         "properties": {
            "file_info": {
               "$ref": "#/definitions/ResisticsFile"
            },
            "fs": {
               "title": "Fs",
               "type": "array",
               "items": {
                  "type": "number"
               }
            },
            "chans": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "n_chans": {
            "title": "N Chans",
            "type": "integer"
         },
         "n_levels": {
            "title": "N Levels",
            "type": "integer"
         },
         "first_time": {
            "title": "First Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "last_time": {
            "title": "Last Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "system": {
            "title": "System",
            "default": "",
            "type": "string"
         },
         "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
         },
         "wgs84_latitude": {
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
         },
         "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
         },
         "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
```

(continues on next page)

(continued from previous page)

```
            },
            "northing": {
               "title": "Northing",
               "default": -999.0,
               "type": "number"
            },
            "elevation": {
               "title": "Elevation",
               "default": -999.0,
               "type": "number"
            },
            "chans_metadata": {
               "title": "Chans Metadata",
               "type": "object",
               "additionalProperties": {
                  "$ref": "#/definitions/ChanMetadata"
               }
            },
            "levels_metadata": {
               "title": "Levels Metadata",
               "type": "array",
               "items": {
                  "$ref": "#/definitions/SpectraLevelMetadata"
               }
            },
            "ref_time": {
               "title": "Ref Time",
               "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
               "examples": [
                  "2021-01-01 00:00:00.000061_035156_250000_000000"
               ]
            },
            "history": {
               "title": "History",
               "default": {
                  "records": []
               },
               "allOf": [
                  {
                     "$ref": "#/definitions/History"
                  }
               ]
            }
         },
         "required": [
            "fs",
            "chans",
            "n_levels",
            "first_time",
            "last_time",
            "chans_metadata",
            "levels_metadata",
```

(continues on next page)

(continued from previous page)

```
            "ref_time"
         ]
      }
   }
}
```

field contributors: `Dict[str, SiteCombinedMetadata | SpectraMetadata]` **[Required]**

Details about the data contributing to the regression input data

field history: *History* **= History(records=[])**

The processing history

class resistics.regression.**RegressionInputData**(*metadata:* RegressionInputMetadata, *tf:* TransferFunction, *freqs: List[float]*, *obs: List[Dict[str, ndarray]]*, *preds: List[ndarray]*)

Bases: *ResisticsData*

Class to hold data that will be input into a solver

The purpose of regression input data is to provision for many different solvers and user written solvers.

The regression input data has the following key attributes:

- freqs
- obs
- preds

The freqs attribute is a 1-D array of evaluation frequencies.

The obs attribute is a dictionary of dictionaries. The parent dictionary has a key of the evaluation frequency index. The secondary dictionary has key of output channel. The values in the secondary dictionary are the observations for that output channel and have 1-D size:

(n_wins x n_cross_chans x 2).

The factor of 2 is because the real and complex parts of each equation are separated into two equations to allow use of solvers that work exclusively on real data.

The preds attribute is a single level dictionary with key of evaluation frequency index and value of the predictors for the evaluation frequency. The predictors have 2-D shape:

(n_wins x n_cross_chans x 2) x (n_input_channels x 2).

The number of windows is multiplied by 2 for the same reason as the observations. The doubling of the input channels is because one is the predictor for the real part of that transfer function component and one is the predictor for the complex part of the transfer function component.

Considering the impedance tensor as an example with:

- output channels Ex, Ey
- input channels Hx, Hy
- cross channels Hx, Hy

The below shows the arrays for the 0 index evaluation frequency:

Observations

- Ex: [w1_crossHx_RE, w1_crossHx_IM, w1_crossHy_RE, w1_crossHy_IM]

- Ey: [w1_crossHx_RE, w1_crossHx_IM, w1_crossHy_RE, w1_crossHy_IM]

Predictors Ex

- w1_crossHx_RE: Zxx_RE Zxx_IM Zxy_RE Zxy_IM
- w1_crossHx_IM: Zxx_RE Zxx_IM Zxy_RE Zxy_IM
- w1_crossHy_RE: Zxx_RE Zxx_IM Zxy_RE Zxy_IM
- w1_crossHy_IM: Zxx_RE Zxx_IM Zxy_RE Zxy_IM

Predictors Ey

- w1_crossHx_RE: Zyx_RE Zyx_IM Zyy_RE Zyy_IM
- w1_crossHx_IM: Zyx_RE Zyx_IM Zyy_RE Zyy_IM
- w1_crossHy_RE: Zyx_RE Zyx_IM Zyy_RE Zyy_IM
- w1_crossHy_IM: Zyx_RE Zyx_IM Zyy_RE Zyy_IM

Note that the predictors are the same regardless of the output channel, only the observations change.

property n_freqs: int

Get the number of frequencies

get_inputs(*freq_idx: int*, *out_chan: str*) → Tuple[ndarray, ndarray]

Get observations and predictions

Parameters

- **freq_idx** (*int*) – The evaluation frequency index
- **out_chan** (*str*) – The output channel

Returns

Observations and predictons

Return type

Tuple[np.ndarray, np.ndarray]

pydantic model resistics.regression.**RegressionPreparerSpectra**

Bases: *ResisticsProcess*

Prepare regression data directly from spectra data

This can be useful for running a single measurement


```
{
   "title": "RegressionPreparerSpectra",
   "description": "Prepare regression data directly from spectra data\n\nThis can␣
↪be useful for running a single measurement",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```

run(*tf:* TransferFunction, *spec_data:* SpectraData) → *RegressionInputData*

Construct the linear equation for solving

pydantic model resistics.regression.**RegressionPreparerGathered**

Bases: *ResisticsProcess*

Regression preparer for gathered data

In nearly all cases, this is the regresson preparer to use. As input, it requires GatheredData.


```
{
   "title": "RegressionPreparerGathered",
   "description": "Regression preparer for gathered data\n\nIn nearly all cases,␣
↪this is the regresson preparer to use. As input, it\nrequires GatheredData.",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*tf:* TransferFunction, *gathered_data:* GatheredData) → *RegressionInputData*

Create the RegressionInputData

Parameters

- **tf** (TransferFunction) – The transfer function
- **gathered_data** (GatheredData) – The gathered data

Returns

Data that can be used as input into a solver

Return type

RegressionInputData

pydantic model resistics.regression.**Solution**

Bases: *WriteableMetadata*

Class to hold a transfer function solution

Examples

```
>>> from resistics.testing import solution_mt
>>> solution = solution_mt()
>>> print(solution.tf.to_string())
| Ex | = | Ex_Hx Ex_Hy | | Hx |
| Ey |   | Ey_Hx Ey_Hy | | Hy |
>>> solution.n_freqs
6
>>> solution.freqs
[100.0, 80.0, 60.0, 40.0, 20.0, 10.0]
>>> solution.periods.tolist()
[0.01, 0.0125, 0.016666666666666666, 0.025, 0.05, 0.1]
```

(continues on next page)

(continued from previous page)

```
>>> solution.components["ExHx"]
Component(real=[1.0, 1.0, 2.0, 2.0, 3.0, 3.0], imag=[5.0, 5.0, 4.0, 4.0, 3.0, 3.0])
>>> solution.components["ExHy"]
Component(real=[1.0, 2.0, 3.0, 4.0, 5.0, 6.0], imag=[-5.0, -4.0, -3.0, -2.0, -1.0,␣
↪1.0])
```

To get the components as an array, either get_component or subscripting be used

```
>>> solution["ExHy"]
array([1.-5.j, 2.-4.j, 3.-3.j, 4.-2.j, 5.-1.j, 6.+1.j])
>>> solution["ab"]
Traceback (most recent call last):
...
ValueError: Component ab not found in ['ExHx', 'ExHy', 'EyHx', 'EyHy']
```

It is also possible to get the tensor values at a particular evaluation frequency

```
>>> solution.get_tensor(2)
array([[ 2.+4.j,  3.-3.j],
       [-3.+3.j, -2.-4.j]])
```


```
{
   "title": "Solution",
   "description": "Class to hold a transfer function solution\n\nExamples\n--------\
↪n>>> from resistics.testing import solution_mt\n>>> solution = solution_mt()\n>>>␣
↪print(solution.tf.to_string())\n| Ex | = | Ex_Hx Ex_Hy | | Hx |\n| Ey |   | Ey_Hx␣
↪Ey_Hy | | Hy |\n>>> solution.n_freqs\n6\n>>> solution.freqs\n[100.0, 80.0, 60.0,␣
↪40.0, 20.0, 10.0]\n>>> solution.periods.tolist()\n[0.01, 0.0125, 0.
↪016666666666666666, 0.025, 0.05, 0.1]\n>>> solution.components[\"ExHx\"]\
↪nComponent(real=[1.0, 1.0, 2.0, 2.0, 3.0, 3.0], imag=[5.0, 5.0, 4.0, 4.0, 3.0, 3.
↪0])\n>>> solution.components[\"ExHy\"]\nComponent(real=[1.0, 2.0, 3.0, 4.0, 5.0,␣
↪6.0], imag=[-5.0, -4.0, -3.0, -2.0, -1.0, 1.0])\n\nTo get the components as an␣
↪array, either get_component or subscripting\nbe used\n\n>>> solution[\"ExHy\"]\
↪narray([1.-5.j, 2.-4.j, 3.-3.j, 4.-2.j, 5.-1.j, 6.+1.j])\n>>> solution[\"ab\"]\
↪nTraceback (most recent call last):\n...\nValueError: Component ab not found in [
↪'ExHx', 'ExHy', 'EyHx', 'EyHy']\n\nIt is also possible to get the tensor values␣
↪at a particular evaluation\nfrequency\n\n>>> solution.get_tensor(2)\narray([[ 2.
↪+4.j,  3.-3.j],\n       [-3.+3.j, -2.-4.j]])",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "tf": {
         "$ref": "#/definitions/TransferFunction"
      },
      "freqs": {
         "title": "Freqs",
         "type": "array",
         "items": {
            "type": "number"
         }
```


(continues on next page)

(continued from previous page)

```
      },
      "components": {
         "title": "Components",
         "type": "object",
         "additionalProperties": {
            "$ref": "#/definitions/Component"
         }
      },
      "history": {
         "$ref": "#/definitions/History"
      },
      "contributors": {
         "title": "Contributors",
         "type": "object",
         "additionalProperties": {
            "anyOf": [
               {
                  "$ref": "#/definitions/SiteCombinedMetadata"
               },
               {
                  "$ref": "#/definitions/SpectraMetadata"
               }
            ]
         }
      }
   },
   "required": [
      "tf",
      "freqs",
      "components",
      "history",
      "contributors"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
               "title": "Version",
               "type": "string"
```

(continues on next page)

(continued from previous page)

```
            }
         }
      },
      "TransferFunction": {
         "title": "TransferFunction",
         "description": "Define a generic transfer function\n\nThis class is a␣
↪describes generic transfer function, including:\n\n- The output channels for the␣
↪transfer function\n- The input channels for the transfer function\n- The cross␣
↪channels for the transfer function\n\nThe cross channels are the channels that␣
↪will be used to calculate out the\ncross powers for the regression.\n\nThis␣
↪generic parent class has no implemented plotting function. However,\nchild␣
↪classes may have a plotting function as different transfer functions\nmay need␣
↪different types of plots.\n\n.. note::\n\n    Users interested in writing a␣
↪custom transfer function should inherit\n    from this generic Transfer function\
↪n\nSee Also\n--------\nImpandanceTensor : Transfer function for the MT impedance␣
↪tensor\nTipper : Transfer function for the MT tipper\n\nExamples\n--------\nA␣
↪generic example\n\n>>> from resistics.transfunc import TransferFunction\n>>> tf =␣
↪TransferFunction(variation=\"example\", out_chans=[\"bye\", \"see you\", \"ciao\
↪"], in_chans=[\"hello\", \"hi_there\"])\n>>> print(tf.to_string())\n| bye      | ␣
↪ | bye_hello         bye_hi_there      | | hello    |\n| see you  | = | see you_
↪hello     see you_hi_there  | | hi_there |\n| ciao     |   | ciao_hello       ␣
↪ciao_hi_there     |\n\nCombining the impedance tensor and the tipper into one␣
↪TransferFunction\n\n>>> tf = TransferFunction(variation=\"combined\", out_chans=[\
↪"Ex\", \"Ey\"], in_chans=[\"Hx\", \"Hy\", \"Hz\"])\n>>> print(tf.to_string())\n|␣
↪Ex |   | Ex_Hx Ex_Hy Ex_Hz | | Hx |\n| Ey | = | Ey_Hx Ey_Hy Ey_Hz | | Hy |\n     ␣
↪                          | Hz |",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "variation": {
               "title": "Variation",
               "default": "generic",
               "maxLength": 16,
               "type": "string"
            },
            "out_chans": {
               "title": "Out Chans",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "in_chans": {
               "title": "In Chans",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
```

(continues on next page)

(continued from previous page)

```
            "cross_chans": {
               "title": "Cross Chans",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "n_out": {
               "title": "N Out",
               "type": "integer"
            },
            "n_in": {
               "title": "N In",
               "type": "integer"
            },
            "n_cross": {
               "title": "N Cross",
               "type": "integer"
            }
         },
         "required": [
            "out_chans",
            "in_chans"
         ]
      },
      "Component": {
         "title": "Component",
         "description": "Data class for a single component in a Transfer function\n\
↪nExample\n-------\n>>> from resistics.transfunc import Component\n>>> component =␣
↪Component(real=[1, 2, 3, 4, 5], imag=[-5, -4, -3, -2 , -1])\n>>> component.get_
↪value(0)\n(1-5j)\n>>> component.to_numpy()\narray([1.-5.j, 2.-4.j, 3.-3.j, 4.-2.j,
↪ 5.-1.j])",
         "type": "object",
         "properties": {
            "real": {
               "title": "Real",
               "type": "array",
               "items": {
                  "type": "number"
               }
            },
            "imag": {
               "title": "Imag",
               "type": "array",
               "items": {
                  "type": "number"
               }
            }
         },
         "required": [
            "real",
            "imag"
```

(continues on next page)

(continued from previous page)

```
        ]
     },
     "Record": {
        "title": "Record",
        "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
        "type": "object",
        "properties": {
           "time_local": {
              "title": "Time Local",
              "type": "string",
              "format": "date-time"
           },
           "time_utc": {
              "title": "Time Utc",
              "type": "string",
              "format": "date-time"
           },
           "creator": {
              "title": "Creator",
              "type": "object"
           },
           "messages": {
              "title": "Messages",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "record_type": {
              "title": "Record Type",
              "type": "string"
           }
        },
        "required": [
           "creator",
           "messages",
           "record_type"
        ]
     },
     "History": {
        "title": "History",
        "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
```

(continues on next page)

(continued from previous page)

```
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n     ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n     ␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n      ␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
        "type": "object",
        "properties": {
           "records": {
              "title": "Records",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/Record"
              }
           }
        }
     },
     "SiteCombinedMetadata": {
        "title": "SiteCombinedMetadata",
        "description": "Metadata for combined data\n\nCombined metadata stores␣
↪metadata for measurements that are combined from\na single site.",
        "type": "object",
        "properties": {
           "file_info": {
              "$ref": "#/definitions/ResisticsFile"
           },
           "site_name": {
              "title": "Site Name",
              "type": "string"
           },
           "fs": {
              "title": "Fs",
              "type": "number"
           },
           "system": {
              "title": "System",
              "default": "",
              "type": "string"
           },
           "serial": {
              "title": "Serial",
              "default": "",
              "type": "string"
           },
           "wgs84_latitude": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
         },
         "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
         },
         "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
         },
         "northing": {
            "title": "Northing",
            "default": -999.0,
            "type": "number"
         },
         "elevation": {
            "title": "Elevation",
            "default": -999.0,
            "type": "number"
         },
         "measurements": {
            "title": "Measurements",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "chans": {
            "title": "Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "n_evals": {
            "title": "N Evals",
            "type": "integer"
         },
         "eval_freqs": {
            "title": "Eval Freqs",
            "type": "array",
            "items": {
               "type": "number"
            }
         },
         "histories": {
            "title": "Histories",
            "type": "object",
```

(continues on next page)

(continued from previous page)

```
            "additionalProperties": {
               "$ref": "#/definitions/History"
            }
         }
      },
      "required": [
         "site_name",
         "fs",
         "chans",
         "n_evals",
         "eval_freqs",
         "histories"
      ]
   },
   "ChanMetadata": {
      "title": "ChanMetadata",
      "description": "Channel metadata",
      "type": "object",
      "properties": {
         "name": {
            "title": "Name",
            "type": "string"
         },
         "data_files": {
            "title": "Data Files",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "chan_type": {
            "title": "Chan Type",
            "type": "string"
         },
         "chan_source": {
            "title": "Chan Source",
            "type": "string"
         },
         "sensor": {
            "title": "Sensor",
            "default": "",
            "type": "string"
         },
         "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
         },
         "gain1": {
            "title": "Gain1",
            "default": 1,
            "type": "number"
```

(continues on next page)

(continued from previous page)

```
            },
            "gain2": {
               "title": "Gain2",
               "default": 1,
               "type": "number"
            },
            "scaling": {
               "title": "Scaling",
               "default": 1,
               "type": "number"
            },
            "chopper": {
               "title": "Chopper",
               "default": false,
               "type": "boolean"
            },
            "dipole_dist": {
               "title": "Dipole Dist",
               "default": 1,
               "type": "number"
            },
            "sensor_calibration_file": {
               "title": "Sensor Calibration File",
               "default": "",
               "type": "string"
            },
            "instrument_calibration_file": {
               "title": "Instrument Calibration File",
               "default": "",
               "type": "string"
            }
         },
         "required": [
            "name"
         ]
      },
      "SpectraLevelMetadata": {
         "title": "SpectraLevelMetadata",
         "description": "Metadata for spectra of a windowed decimation level",
         "type": "object",
         "properties": {
            "fs": {
               "title": "Fs",
               "type": "number"
            },
            "n_wins": {
               "title": "N Wins",
               "type": "integer"
            },
            "win_size": {
               "title": "Win Size",
               "exclusiveMinimum": 0,
```

(continues on next page)

(continued from previous page)

```
            "type": "integer"
         },
         "olap_size": {
            "title": "Olap Size",
            "exclusiveMinimum": 0,
            "type": "integer"
         },
         "index_offset": {
            "title": "Index Offset",
            "type": "integer"
         },
         "n_freqs": {
            "title": "N Freqs",
            "type": "integer"
         },
         "freqs": {
            "title": "Freqs",
            "type": "array",
            "items": {
               "type": "number"
            }
         }
      },
      "required": [
         "fs",
         "n_wins",
         "win_size",
         "olap_size",
         "index_offset",
         "n_freqs",
         "freqs"
      ]
   },
   "SpectraMetadata": {
      "title": "SpectraMetadata",
      "description": "Metadata for spectra data",
      "type": "object",
      "properties": {
         "file_info": {
            "$ref": "#/definitions/ResisticsFile"
         },
         "fs": {
            "title": "Fs",
            "type": "array",
            "items": {
               "type": "number"
            }
         },
         "chans": {
            "title": "Chans",
            "type": "array",
            "items": {
```

(continues on next page)

(continued from previous page)

```
                "type": "string"
            }
        },
        "n_chans": {
            "title": "N Chans",
            "type": "integer"
        },
        "n_levels": {
            "title": "N Levels",
            "type": "integer"
        },
        "first_time": {
            "title": "First Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
                "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
        },
        "last_time": {
            "title": "Last Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
                "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
        },
        "system": {
            "title": "System",
            "default": "",
            "type": "string"
        },
        "serial": {
            "title": "Serial",
            "default": "",
            "type": "string"
        },
        "wgs84_latitude": {
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
        },
        "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
        },
        "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
        },
        "northing": {
            "title": "Northing",
```

(continues on next page)

(continued from previous page)

```
            "default": -999.0,
            "type": "number"
         },
         "elevation": {
            "title": "Elevation",
            "default": -999.0,
            "type": "number"
         },
         "chans_metadata": {
            "title": "Chans Metadata",
            "type": "object",
            "additionalProperties": {
               "$ref": "#/definitions/ChanMetadata"
            }
         },
         "levels_metadata": {
            "title": "Levels Metadata",
            "type": "array",
            "items": {
               "$ref": "#/definitions/SpectraLevelMetadata"
            }
         },
         "ref_time": {
            "title": "Ref Time",
            "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
            "examples": [
               "2021-01-01 00:00:00.000061_035156_250000_000000"
            ]
         },
         "history": {
            "title": "History",
            "default": {
               "records": []
            },
            "allOf": [
               {
                  "$ref": "#/definitions/History"
               }
            ]
         }
      },
      "required": [
         "fs",
         "chans",
         "n_levels",
         "first_time",
         "last_time",
         "chans_metadata",
         "levels_metadata",
         "ref_time"
      ]
   }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

field tf: *TransferFunction* **[Required]**

The transfer function that was solved

field freqs: List[float] **[Required]**

The evaluation frequencies

field components: Dict[str, *Component*] **[Required]**

The solution

field history: *History* **[Required]**

The processing history

field contributors: Dict[str, *SiteCombinedMetadata* | *SpectraMetadata*] **[Required]**

The contributors to the solution with their respective details

property n_freqs

Get the number of evaluation frequencies

property periods: ndarray

Get the periods

get_component(*key: str*) → ndarray

Get the solution for a single component for all the evaluation frequencies

Parameters
key (*str*) – The component key

Returns
The component data in an array

Return type
np.ndarray

Raises
ValueError – If the component does not exist in the solution

get_tensor(*eval_idx: int*) → ndarray

Get the tensor at a single evaluation frequency. This has shape:

n_out_chans x n_in_chans

Parameters
eval_idx (*int*) – The index of the evaluation frequency

Returns
The tensor as a numpy array

Return type
np.ndarray

to_dataframe() → DataFrame

Get the solution as a pandas DataFrame

pydantic model resistics.regression.**Solver**

Bases: *ResisticsProcess*

General resistics solver

```
{
   "title": "Solver",
   "description": "General resistics solver",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*regression_input:* RegressionInputData) → *Solution*

Every solver should have a run method

pydantic model resistics.regression.**SolverScikit**

Bases: *Solver*

Base class for Scikit learn solvers


```
{
   "title": "SolverScikit",
   "description": "Base class for Scikit learn solvers",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "fit_intercept": {
         "title": "Fit Intercept",
         "default": false,
         "type": "boolean"
      },
      "normalize": {
         "title": "Normalize",
         "default": false,
         "type": "boolean"
      }
   }
}
```


field fit_intercept: bool = False

Flag for adding an intercept term

field normalize: bool = False

Flag for normalizing, only used if fit_intercept is True

pydantic model resistics.regression.**SolverScikitOLS**

Bases: *SolverScikit*

Ordinary least squares solver

This is simply a wrapper around the scikit learn least squares regression https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

```
{
   "title": "SolverScikitOLS",
   "description": "Ordinary least squares solver\n\nThis is simply a wrapper around␣
↪the scikit learn least squares regression\nhttps://scikit-learn.org/stable/
↪modules/generated/sklearn.linear_model.LinearRegression.html",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "fit_intercept": {
         "title": "Fit Intercept",
         "default": false,
         "type": "boolean"
      },
      "normalize": {
         "title": "Normalize",
         "default": false,
         "type": "boolean"
      },
      "n_jobs": {
         "title": "N Jobs",
         "default": -2,
         "type": "integer"
      }
   }
}
```


field n_jobs: `int = -2`

Number of jobs to run

run(*regression_input:* RegressionInputData) → *Solution*

Run ordinary least squares regression on the RegressionInputData

pydantic model resistics.regression.**SolverScikitHuber**

Bases: *SolverScikit*

Scikit Huber solver

This is simply a wrapper around the scikit learn Huber Regressor. For more information, please see https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.HuberRegressor.html


```
{
   "title": "SolverScikitHuber",
   "description": "Scikit Huber solver\n\nThis is simply a wrapper around the␣
↪scikit learn Huber Regressor. For\nmore information, please see\nhttps://scikit-
↪learn.org/stable/modules/generated/sklearn.linear_model.HuberRegressor.html",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
```


(continues on next page)

(continued from previous page)


```
      "fit_intercept": {
         "title": "Fit Intercept",
         "default": false,
         "type": "boolean"
      },
      "normalize": {
         "title": "Normalize",
         "default": false,
         "type": "boolean"
      },
      "epsilon": {
         "title": "Epsilon",
         "default": 1,
         "type": "number"
      }
   }
}
```


field epsilon: `float = 1`

The smaller the epsilon, the more robust it is to outliers.

run(*regression_input:* RegressionInputData) → *Solution*

Run Huber Regressor regression on the RegressionInputData

pydantic model resistics.regression.**SolverScikitTheilSen**

Bases: *SolverScikit*

Scikit Theil Sen solver

This is simply a wrapper around the scikit learn Theil Sen Regressor. For more information, please see https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.TheilSenRegressor.html


```
{
   "title": "SolverScikitTheilSen",
   "description": "Scikit Theil Sen solver\n\nThis is simply a wrapper around the␣
↪scikit learn Theil Sen Regressor. For\nmore information, please see\nhttps://
↪scikit-learn.org/stable/modules/generated/sklearn.linear_model.TheilSenRegressor.
↪html",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "fit_intercept": {
         "title": "Fit Intercept",
         "default": false,
         "type": "boolean"
      },
      "normalize": {
         "title": "Normalize",
         "default": false,
         "type": "boolean"
```


(continues on next page)

(continued from previous page)

```
      },
      "n_jobs": {
         "title": "N Jobs",
         "default": -2,
         "type": "integer"
      },
      "max_subpopulation": {
         "title": "Max Subpopulation",
         "default": 2000,
         "type": "integer"
      },
      "n_subsamples": {
         "title": "N Subsamples",
         "type": "number"
      }
   }
}
```

field n_jobs: `int = -2`

Number of jobs to run

field max_subpopulation: `int = 2000`

Maximum population. Reduce this if the process is taking a long time

field n_subsamples: `float | None = None`

Number of rows to use for each solution

run(*regression_input:* RegressionInputData) → *Solution*

Run TheilSen regression on the RegressionInputData

pydantic model resistics.regression.**SolverScikitRANSAC**

Bases: *SolverScikit*

Run a RANSAC solver with LinearRegression as Base Estimator

This is a wrapper around the scikit learn RANSAC regressor. More information can be found here https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RANSACRegressor.html

```
{
   "title": "SolverScikitRANSAC",
   "description": "Run a RANSAC solver with LinearRegression as Base Estimator\n\
↪nThis is a wrapper around the scikit learn RANSAC regressor. More information\
↪ncan be found here\nhttps://scikit-learn.org/stable/modules/generated/sklearn.
↪linear_model.RANSACRegressor.html",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "fit_intercept": {
         "title": "Fit Intercept",
         "default": false,
         "type": "boolean"
```

(continues on next page)

(continued from previous page)


```
      },
      "normalize": {
         "title": "Normalize",
         "default": false,
         "type": "boolean"
      },
      "min_samples": {
         "title": "Min Samples",
         "default": 0.8,
         "type": "number"
      },
      "max_trials": {
         "title": "Max Trials",
         "default": 20,
         "type": "integer"
      }
   }
}
```


field min_samples: float = 0.8

Minimum number of samples in each solution as a proportion of total

field max_trials: int = 20

The maximum number of trials to run

run(*regression_input:* RegressionInputData) → *Solution*

Run RANSAC regression on the RegressionInputData

pydantic model resistics.regression.**SolverScikitWLS**

Bases: *SolverScikitOLS*

Weighted least squares solver

Warning: This is homespun and is currently only experimental

This is simply a wrapper around the scikit learn least squares regression using the sample_weight option https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html


```
{
   "title": "SolverScikitWLS",
   "description": "Weighted least squares solver\n\n.. warning::\n\n    This is␣
↪homespun and is currently only experimental\n\nThis is simply a wrapper around␣
↪the scikit learn least squares regression\nusing the sample_weight option\nhttps:/
↪/scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.
↪html",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "fit_intercept": {
```


(continues on next page)

(continued from previous page)

```
            "title": "Fit Intercept",
            "default": false,
            "type": "boolean"
        },
        "normalize": {
            "title": "Normalize",
            "default": false,
            "type": "boolean"
        },
        "n_jobs": {
            "title": "N Jobs",
            "default": -2,
            "type": "integer"
        },
        "n_iter": {
            "title": "N Iter",
            "default": 50,
            "type": "integer"
        }
    }
}
```

field n_jobs: *int* **= -2**

Number of jobs to run

field n_iter: *int* **= 50**

Number of iterations before quitting if residual is not low enough

bisquare(*r: ndarray*, *k: float = 4.685*) → ndarray

Bisquare location weights

Parameters

- **r** (*np.ndarray*) – Residuals
- **k** (*float, None*) – Tuning parameter. If None, a standard value will be used.

Returns

weights – The robust weights

Return type

np.ndarray

huber(*r: ndarray*, *k: float = 1.345*) → ndarray

Huber location weights

Parameters

- **r** (*np.ndarray*) – Residuals
- **k** (*float*) – Tuning parameter. If None, a standard value will be used.

Returns

weights – The robust weights

Return type

np.ndarray

trimmed_mean(*r: ndarray*, *k: float = 2*) → ndarray

Trimmed mean location weights

Parameters

- **r** (*np.ndarray*) – Residuals
- **k** (*float*) – Tuning parameter. If None, a standard value will be used.

Returns

weights – The robust weights

Return type

np.ndarray

resistics.sampling module

Module for dealing with sampling and dates including:

- Converting from samples to datetimes
- Converting from datetimes to samples
- All datetime, timedelta types are aliased as RSDateTime and RSTimeDelta
- This is to ease type hinting if the base datetime and timedelta classes change
- Currently, resistics uses attodatetime and attotimedelta from attotime
- attotime is a high precision datetime library

class resistics.sampling.**HighResDateTime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *nanosecond=0*, *tzinfo=None*)

Bases: attodatetime

Wrapper around RSDateTime to use for pydantic

classmethod validate(*val: attodatetime | str | Timestamp | datetime*)

Validator to be used by pydantic

resistics.sampling.**datetime_to_string**(*time: attodatetime*) → str

Convert a datetime to a string.

Parameters

time (*RSDateTime*) – Resistics datetime

Returns

String representation

Return type

str

Examples

```
>>> from resistics.sampling import to_datetime, to_timedelta, datetime_to_string
>>> time = to_datetime("2021-01-01") + to_timedelta(1/16384)
>>> datetime_to_string(time)
'2021-01-01 00:00:00.000061_035156_250000_000000'
```

resistics.sampling.**datetime_from_string**(*time: str*) → attodatetime

Convert a string back to a datetime.

Only a fixed format is allowed %Y-%m-%d %H:%M:%S.%f_%o_%q_%v

Parameters
time (*str*) – time as a string

Returns
The resistics datetime

Return type
RSDateTime

Examples

```
>>> from resistics.sampling import to_datetime, to_timedelta
>>> from resistics.sampling import datetime_to_string, datetime_from_string
>>> time = to_datetime("2021-01-01") + to_timedelta(1/16384)
>>> time_str = datetime_to_string(time)
>>> time_str
'2021-01-01 00:00:00.000061_035156_250000_000000'
>>> datetime_from_string(time_str)
attotime.objects.attodatetime(2021, 1, 1, 0, 0, 0, 61, 35.15625)
```

resistics.sampling.**to_datetime**(*time: str | Timestamp | datetime*) → attodatetime

Convert a string, pd.Timestamp or datetime object to a RSDateTime.

RSDateTime uses attodatetime which is a high precision datetime format helpful for high sampling frequencies.

Parameters
time (*DateTimeLike*) – Input time as either a string, pd.Timestamp or native python datetime

Returns
High precision datetime object

Return type
RSDateTime

Examples

```
>>> import pandas as pd
>>> from resistics.sampling import to_datetime
>>> a = "2021-01-01 00:00:00"
>>> to_datetime(a)
attotime.objects.attodatetime(2021, 1, 1, 0, 0, 0, 0, 0)
>>> str(to_datetime(a))
'2021-01-01 00:00:00'
>>> b = pd.Timestamp(a)
>>> str(to_datetime(b))
'2021-01-01 00:00:00'
>>> c = pd.Timestamp(a).to_pydatetime()
>>> str(to_datetime(c))
'2021-01-01 00:00:00'
```

resistics.sampling.**to_timestamp**(*time: attodatetime*) → Timestamp

Convert a RSDateTime to a pandas Timestamp

Parameters
time (*RSDateTime*) – An RSDateTime instance

Returns
RSDateTime converted to Timestamp

Return type
pd.Timestamp

Examples

```
>>> from resistics.sampling import to_datetime, to_timestamp
>>> time = to_datetime("2021-01-01 00:30:00.345")
>>> print(time)
2021-01-01 00:30:00.345
>>> to_timestamp(time)
Timestamp('2021-01-01 00:30:00.345000')
```

resistics.sampling.**to_timedelta**(*delta: float | timedelta | Timedelta*) → attotimedelta

Get a RSTimeDelta object by providing seconds as a float or a pd.Timedelta.

RSTimeDelta uses attotimedelta, a high precision timedelta object. This can be useful for high sampling frequencies.

Warning: At high time resolutions, there are machine precision errors that come into play. Therefore, if nanoseconds < 0.0001, it will be zeroed out

Parameters
delta (*TimeDeltaLike*) – Timedelta as a float (assumed to be seconds), timedelta or pd.Timedelta

Returns
High precision timedelta

Return type
RSTimeDelta

Examples

```
>>> import pandas as pd
>>> from resistics.sampling import to_timedelta
```

Low frequency sampling

```
>>> fs = 0.0000125
>>> to_timedelta(1/fs)
attotime.objects.attotimedelta(0, 80000)
>>> str(to_timedelta(1/fs))
'22:13:20'
>>> fs = 0.004
>>> to_timedelta(1/fs)
attotime.objects.attotimedelta(0, 250)
>>> str(to_timedelta(1/fs))
'0:04:10'
>>> fs = 0.3125
>>> str(to_timedelta(1/fs))
'0:00:03.2'
```

Higher frequency sampling

```
>>> fs = 4096
>>> to_timedelta(1/fs)
attotime.objects.attotimedelta(0, 0, 244, 140.625)
>>> str(to_timedelta(1/fs))
'0:00:00.000244140625'
>>> fs = 65_536
>>> str(to_timedelta(1/fs))
'0:00:00.0000152587890625'
>>> fs = 524_288
>>> str(to_timedelta(1/fs))
'0:00:00.0000019073486328125'
```

to_timedelta can also accept pandas Timedelta objects

```
>>> str(to_timedelta(pd.Timedelta(1, "s")))
'0:00:01'
```

resistics.sampling.**to_seconds**(*delta: attotimedelta*) → Tuple[float, float]

Convert a timedelta to seconds as a float.

Returns a Tuple, the first value being the days in the delta converted to seconds, the second entry in the Tuple is the remaining amount of time converted to seconds.

Parameters
delta (*RSTimeDelta*) – timedelta

Returns

- *days_in_seconds* – The days in the delta converted to seconds

- *remaining_in_seconds* – The remaining amount of time in the delta converted to seconds

Examples

Example with a small timedelta

```
>>> from resistics.sampling import to_datetime, to_timedelta, to_seconds
>>> a = to_timedelta(1/4_096)
>>> str(a)
'0:00:00.000244140625'
>>> days_in_seconds, remaining_in_seconds = to_seconds(a)
>>> days_in_seconds
0
>>> remaining_in_seconds
0.000244140625
```

Example with a larger timedelta

```
>>> a = to_datetime("2021-01-01 00:00:00")
>>> b = to_datetime("2021-02-01 08:24:30")
>>> days_in_seconds, remaining_in_seconds = to_seconds(b-a)
>>> days_in_seconds
2678400
>>> remaining_in_seconds
30270.0
```

resistics.sampling.**to_n_samples**(*delta: attotimedelta*, *fs: float*, *method: str = 'round'*) → int

Convert a timedelta to number of samples

This method is inclusive of start and end sample.

Parameters

- **delta** (*RSTimeDelta*) – The timedelta
- **fs** (*float*) – The sampling frequency
- **method** (*str*) – Method to deal with floats, default is 'round'. Other options include 'ceil' and 'floor'

Returns

The number of samples in the timedelta

Return type

int

Examples

With sampling frequency of 4096 Hz

```
>>> from resistics.sampling import to_timedelta, to_n_samples
>>> fs = 4096
>>> delta = to_timedelta(8*3600 + (21/fs))
>>> str(delta)
'8:00:00.005126953125'
```

(continues on next page)

(continued from previous page)

```
>>> to_n_samples(delta, fs=fs)
117964822
>>> check = (8*3600)*fs + 21
>>> check
117964821
>>> check_inclusive = check + 1
>>> check_inclusive
117964822
```

With a sampling frequency of 65536 Hz

```
>>> fs = 65_536
>>> delta = to_timedelta(2*3600 + (40_954/fs))
>>> str(delta)
'2:00:00.624908447265625'
>>> to_n_samples(delta, fs=fs)
471900155
>>> check = 2*3600*fs + 40_954
>>> check
471900154
>>> check_inclusive = check + 1
>>> check_inclusive
471900155
```

resistics.sampling.**check_sample**(*n_samples: int*, *sample: int*) → bool

Check sample is between 0 <= from_sample < n_samples

Parameters

- **n_samples** (*int*) – Number of samples
- **sample** (*int*) – Sample to check

Returns

Return True if no errors

Return type

bool

Raises

- **ValueError** – If sample < 0
- **ValueError** – If sample > n_samples

Examples

```
>>> from resistics.sampling import check_sample
>>> check_sample(100, 45)
True
>>> check_sample(100, 100)
Traceback (most recent call last):
...
ValueError: Sample 100 must be < 100
>>> check_sample(100, -1)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: Sample -1 must be >= 0
```

resistics.sampling.**sample_to_datetime**(*fs: float*, *first_time: attodatetime*, *sample: int*, *n_samples: int | None = None*) → attodatetime

Convert a sample to a pandas Timestamp.

Parameters

- **fs** (*float*) – The sampling frequency
- **first_time** (*RSDateTime*) – The first time
- **sample** (*int*) – The sample
- **n_samples** (*Optional[int], optional*) – The number of samples, used for checking, by default None. If provided, the sample is checked to make sure it's not out of bounds.

Returns

The timestamp of the sample

Return type

RSDateTime

Raises

ValueError – If n_samples is provided and sample is < 0 or >= n_samples

Examples

```
>>> import pandas as pd
>>> from resistics.sampling import to_datetime, sample_to_datetime
>>> fs = 512
>>> first_time = to_datetime("2021-01-02 00:00:00")
>>> sample = 512
>>> sample_datetime = sample_to_datetime(fs, first_time, sample)
>>> str(sample_datetime)
'2021-01-02 00:00:01'
```

resistics.sampling.**samples_to_datetimes**(*fs: float*, *first_time: attodatetime*, *from_sample: int*, *to_sample: int*) → Tuple[attodatetime, attodatetime]

Convert from and to samples to datetimes.

The first sample is assumed to be 0.

Parameters

- **fs** (*float*) – The sampling frequency in seconds
- **first_time** (*RSDateTime*) – The time of the first sample
- **from_sample** (*int*) – The sample to read data from
- **to_sample** (*int*) – The sample to read data to

Returns

- **from_time** (*RSDateTime*) – The timestamp to read data from

- **to_time** (*RSDateTime*) – The timestamp to read data to

Raises

ValueError – If from sample is greater than or equal to to sample

Examples

```
>>> import pandas as pd
>>> from resistics.sampling import to_datetime, samples_to_datetimes
>>> fs = 512
>>> first_time = to_datetime("2021-01-02 00:00:00")
>>> from_sample = 512
>>> to_sample = 1024
>>> from_time, to_time = samples_to_datetimes(fs, first_time, from_sample, to_
↪sample)
>>> str(from_time)
'2021-01-02 00:00:01'
>>> str(to_time)
'2021-01-02 00:00:02'
```

resistics.sampling.**check_from_time**(*first_time: attodatetime*, *last_time: attodatetime*, *from_time: attodatetime*) → attodatetime

Check a from time.

- If first time <= from_time <= last_time, it will be returned unchanged.
- If from_time < first time, then first time will be returned.
- If from_time > last time, it will raise a ValueError.

Parameters

- **first_time** (*RSDateTime*) – The time of the first sample
- **last_time** (*RSDateTime*) – The time of the last sample
- **from_time** (*RSDateTime*) – Time to get the data from

Returns

A from time adjusted as needed given the first and last sample time

Return type

RSDateTime

Raises

ValueError – If the from time is after the time of the last sample

Examples

With a from time between first and last time. This should be the normal use case.

```
>>> from resistics.sampling import to_datetime, check_from_time
>>> first_time = to_datetime("2021-01-02 00:00:00")
>>> last_time = to_datetime("2021-01-02 23:00:00")
>>> from_time = to_datetime("2021-01-02 03:00:00")
>>> from_time = check_from_time(first_time, last_time, from_time)
```

(continues on next page)

(continued from previous page)

```
>>> str(from_time)
'2021-01-02 03:00:00'
```

An alternative scenario when from time is before the time of the first sample

```
>>> from_time = to_datetime("2021-01-01 23:00:00")
>>> from_time = check_from_time(first_time, last_time, from_time)
>>> str(from_time)
'2021-01-02 00:00:00'
```

An error will be raised when from time is after the time of the last sample

```
>>> from_time = to_datetime("2021-01-02 23:30:00")
>>> from_time = check_from_time(first_time, last_time, from_time)
Traceback (most recent call last):
...
ValueError: From time 2021-01-02 23:30:00 greater than time of last sample 2021-01-
↪02 23:00:00
```

resistics.sampling.**check_to_time**(*first_time: attodatetime*, *last_time: attodatetime*, *to_time: attodatetime*) → attodatetime

Check a to time.

- If first time <= to time <= last time, it will be returned unchanged.
- If to time > last time, then last time will be returned.
- If to time < first time, it will raise a ValueError.

Parameters

- **first_time** (*RSDateTime*) – The time of the first sample
- **last_time** (*RSDateTime*) – The time of the last sample
- **to_time** (*RSDateTime*) – Time to get the data to

Returns
A to time adjusted as needed

Return type
RSDateTime

Raises
ValueError – If the to time is before the time of the first sample

Examples

With a to time between first and last time. This should be the normal use case.

```
>>> from resistics.sampling import to_datetime, check_to_time
>>> first_time = to_datetime("2021-01-02 00:00:00")
>>> last_time = to_datetime("2021-01-02 23:00:00")
>>> to_time = to_datetime("2021-01-02 20:00:00")
>>> to_time = check_to_time(to_time, last_time, to_time)
```

(continues on next page)

(continued from previous page)

```
>>> str(to_time)
'2021-01-02 20:00:00'
```

An alternative scenario when to time is after the time of the last sample

```
>>> to_time = to_datetime("2021-01-02 23:30:00")
>>> to_time = check_to_time(first_time, last_time, to_time)
>>> str(to_time)
'2021-01-02 23:00:00'
```

An error will be raised when to time is before the time of the first sample

```
>>> to_time = to_datetime("2021-01-01 23:30:00")
>>> to_time = check_to_time(first_time, last_time, to_time)
Traceback (most recent call last):
...
ValueError: To time 2021-01-01 23:30:00 less than time of first sample 2021-01-02␣
↪00:00:00
```

resistics.sampling.**from_time_to_sample**(*fs: float*, *first_time: attodatetime*, *last_time: attodatetime*, *from_time: attodatetime*) → int

Get the sample for the from time.

Parameters

- **fs** (*float*) – Sampling frequency Hz
- **first_time** (*RSDateTime*) – Time of first sample
- **last_time** (*RSDateTime*) – Time of last sample
- **from_time** (*RSDateTime*) – From time

Returns

The sample coincident with or after the from time

Return type

int

Examples

```
>>> from resistics.sampling import to_datetime, from_time_to_sample
>>> first_time = to_datetime("2021-01-01 00:00:00")
>>> last_time = to_datetime("2021-01-02 00:00:00")
>>> fs = 128
>>> fs * 60 * 60
460800
>>> from_time = to_datetime("2021-01-01 01:00:00")
>>> from_time_to_sample(fs, first_time, last_time, from_time)
460800
>>> from_time = to_datetime("2021-01-01 01:00:00.0078125")
>>> from_time_to_sample(fs, first_time, last_time, from_time)
460801
```

resistics.sampling.**to_time_to_sample**(*fs: float*, *first_time: attodatetime*, *last_time: attodatetime*, *to_time: attodatetime*) → int

Get the to time sample.

Warning: This will return the sample of the to time. In cases where this will be used for a range, 1 should be added to it to ensure it is included.

Parameters

- **fs** (*float*) – Sampling frequency Hz
- **first_time** (*RSDateTime*) – Time of first sample
- **last_time** (*RSDateTime*) – Time of last sample
- **to_time** (*RSDateTime*) – The to time

Returns
The sample coincident with or immediately before the to time

Return type
int

Examples

```
>>> from resistics.sampling import to_time_to_sample
>>> first_time = to_datetime("2021-01-01 04:00:00")
>>> last_time = to_datetime("2021-01-01 13:00:00")
>>> fs = 4096
>>> fs * 60 * 60
14745600
>>> to_time = to_datetime("2021-01-01 05:00:00")
>>> to_time_to_sample(fs, first_time, last_time, to_time)
14745600
>>> fs * 70 * 60
17203200
>>> to_time = to_datetime("2021-01-01 05:10:00")
>>> to_time_to_sample(fs, first_time, last_time, to_time)
17203200
```

resistics.sampling.**datetimes_to_samples**(*fs: float*, *first_time: attodatetime*, *last_time: attodatetime*, *from_time: attodatetime*, *to_time: attodatetime*) → Tuple[int, int]

Convert from and to time to samples.

Warning: If using these samples in ranging, the from sample can be left unchanged but one should be added to the to sample to ensure it is included.

Note: If from_time is not a sample timestamp, the next sample is taken If to_time is not a sample timestamp, the previous sample is taken

Parameters

- **fs** (*float*) – The sampling frequency in Hz
- **first_time** (*RSDateTime*) – The time of the first sample
- **last_time** (*RSDateTime*) – The time of the last sample
- **from_time** (*RSDateTime*) – A from time
- **to_time** (*RSDateTime*) – A to time

Returns

- **from_sample** (*int*) – Sample to read data from
- **to_sample** (*int*) – Sample to read data to

Examples

```
>>> from resistics.sampling import to_datetime, datetimes_to_samples
>>> first_time = to_datetime("2021-01-01 04:00:00")
>>> last_time = to_datetime("2021-01-01 05:30:00")
>>> from_time = to_datetime("2021-01-01 05:00:00")
>>> to_time = to_datetime("2021-01-01 05:10:00")
>>> fs = 16_384
>>> fs * 60 * 60
58982400
>>> fs * 70 * 60
68812800
>>> from_sample, to_sample = datetimes_to_samples(fs, first_time, last_time, from_
↪time, to_time)
>>> from_sample
58982400
>>> to_sample
68812800
```

resistics.sampling.**datetime_array**(*first_time: attodatetime*, *fs: float*, *n_samples: int | None = None*, *samples: ndarray | None = None*) → ndarray

Get a datetime array in high resolution.

This will return a high resolution datetime array. This method is more computationally demanding than a pandas date_range. As a result, in cases where exact datetimes are not required, it is suggested to use datetime_array_estimate instead.

Parameters

- **first_time** (*RSDateTime*) – The first time
- **fs** (*float*) – The sampling frequency
- **n_samples** (*Optional[int], optional*) – The number of samples, by default None
- **samples** (*Optional[np.ndarray], optional*) – The samples for which to return a datetime, by default None

Returns

Numpy array of RSDateTimes

Return type

np.ndarray

Raises

ValueError – If both n_samples and samples is None

Examples

This examples shows the value of using higher resolution datetimes, however this is computationally more expensive.

```
>>> import pandas as pd
>>> from resistics.sampling import to_datetime, datetime_array
>>> first_time = to_datetime("2021-01-01 00:00:00")
>>> fs = 4096
>>> n_samples = 100
>>> arr = datetime_array(first_time, fs, n_samples=n_samples)
>>> str(arr[-1])
'2021-01-01 00:00:00.024169921875'
>>> pdarr = pd.date_range(start="2021-01-01 00:00:00", freq=pd.Timedelta(1/4096, "s
↪"), periods=n_samples)
>>> pdarr[-1]
Timestamp('2021-01-01 00:00:00.024169959')
```

resistics.sampling.**datetime_array_estimate**(*first_time: attodatetime | str | Timestamp | datetime*, *fs: float*, *n_samples: int | None = None*, *samples: ndarray | None = None*) → DatetimeIndex

Estimate datetime array with lower precision but much faster performance.

Parameters

- **first_time** (*Union[RSDateTime, datetime, str, pd.Timestamp]*) – The first time
- **fs** (*float*) – The sampling frequency
- **n_samples** (*Optional[int], optional*) – The number of samples, by default None
- **samples** (*Optional[np.ndarray], optional*) – An array of samples to return datetimes for, by default None

Returns

A pandas DatetimeIndex

Return type

pd.DatetimeIndex

Raises

ValueError – If both n_samples and samples are None

Examples

```
>>> import pandas as pd
>>> from resistics.sampling import to_datetime, datetime_array_estimate
>>> first_time = to_datetime("2021-01-01 00:00:00")
>>> fs = 128
>>> n_samples = 1_000
>>> arr = datetime_array_estimate(first_time, fs, n_samples=n_samples)
>>> print(f"{arr[0]} - {arr[-1]}")
2021-01-01 00:00:00 - 2021-01-01 00:00:07.804687500
```

resistics.spectra module

Module containing functions and classes related to Spectra calculation and manipulation

Spectra are calculated from the windowed, decimated time data. The inbuilt Fourier transform implementation is inspired by the implementation of the scipy stft function.

pydantic model resistics.spectra.**SpectraLevelMetadata**

Bases: *Metadata*

Metadata for spectra of a windowed decimation level


```
{
   "title": "SpectraLevelMetadata",
   "description": "Metadata for spectra of a windowed decimation level",
   "type": "object",
   "properties": {
      "fs": {
         "title": "Fs",
         "type": "number"
      },
      "n_wins": {
         "title": "N Wins",
         "type": "integer"
      },
      "win_size": {
         "title": "Win Size",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "olap_size": {
         "title": "Olap Size",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "index_offset": {
         "title": "Index Offset",
         "type": "integer"
      },
      "n_freqs": {
         "title": "N Freqs",
         "type": "integer"
      },
      "freqs": {
         "title": "Freqs",
         "type": "array",
         "items": {
            "type": "number"
         }
      }
   },
   "required": [
      "fs",
      "n_wins",
      "win_size",
```


(continues on next page)

(continued from previous page)

```
      "olap_size",
      "index_offset",
      "n_freqs",
      "freqs"
   ]
}
```

field fs: float [Required]

The sampling frequency of the decimation level

field n_wins: int [Required]

The number of windows

field win_size: PositiveInt [Required]

The window size in samples

Constraints

- **exclusiveMinimum** = 0

field olap_size: PositiveInt [Required]

The overlap size in samples

Constraints

- **exclusiveMinimum** = 0

field index_offset: int [Required]

The global window offset for local window 0

field n_freqs: int [Required]

The number of frequencies in the frequency data

field freqs: List[float] [Required]

List of frequencies

property nyquist: float

Get the nyquist frequency

pydantic model resistics.spectra.**SpectraMetadata**

Bases: *WriteableMetadata*

Metadata for spectra data

```
{
   "title": "SpectraMetadata",
   "description": "Metadata for spectra data",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "fs": {
         "title": "Fs",
         "type": "array",
         "items": {
```

(continues on next page)

(continued from previous page)

```
            "type": "number"
         }
      },
      "chans": {
         "title": "Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "n_chans": {
         "title": "N Chans",
         "type": "integer"
      },
      "n_levels": {
         "title": "N Levels",
         "type": "integer"
      },
      "first_time": {
         "title": "First Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "last_time": {
         "title": "Last Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "system": {
         "title": "System",
         "default": "",
         "type": "string"
      },
      "serial": {
         "title": "Serial",
         "default": "",
         "type": "string"
      },
      "wgs84_latitude": {
         "title": "Wgs84 Latitude",
         "default": -999.0,
         "type": "number"
      },
      "wgs84_longitude": {
         "title": "Wgs84 Longitude",
         "default": -999.0,
         "type": "number"
      },
```

(continues on next page)

(continued from previous page)


```
      "easting": {
         "title": "Easting",
         "default": -999.0,
         "type": "number"
      },
      "northing": {
         "title": "Northing",
         "default": -999.0,
         "type": "number"
      },
      "elevation": {
         "title": "Elevation",
         "default": -999.0,
         "type": "number"
      },
      "chans_metadata": {
         "title": "Chans Metadata",
         "type": "object",
         "additionalProperties": {
            "$ref": "#/definitions/ChanMetadata"
         }
      },
      "levels_metadata": {
         "title": "Levels Metadata",
         "type": "array",
         "items": {
            "$ref": "#/definitions/SpectraLevelMetadata"
         }
      },
      "ref_time": {
         "title": "Ref Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "history": {
         "title": "History",
         "default": {
            "records": []
         },
         "allOf": [
            {
               "$ref": "#/definitions/History"
            }
         ]
      }
   },
   "required": [
      "fs",
      "chans",
      "n_levels",
```


(continues on next page)

(continued from previous page)

```
      "first_time",
      "last_time",
      "chans_metadata",
      "levels_metadata",
      "ref_time"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
         "type": "object",
         "properties": {
            "created_on_local": {
               "title": "Created On Local",
               "type": "string",
               "format": "date-time"
            },
            "created_on_utc": {
               "title": "Created On Utc",
               "type": "string",
               "format": "date-time"
            },
            "version": {
               "title": "Version",
               "type": "string"
            }
         }
      },
      "ChanMetadata": {
         "title": "ChanMetadata",
         "description": "Channel metadata",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "data_files": {
               "title": "Data Files",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "chan_type": {
               "title": "Chan Type",
               "type": "string"
            },
            "chan_source": {
               "title": "Chan Source",
               "type": "string"
            },
```

(continues on next page)

(continued from previous page)

```
            "sensor": {
               "title": "Sensor",
               "default": "",
               "type": "string"
            },
            "serial": {
               "title": "Serial",
               "default": "",
               "type": "string"
            },
            "gain1": {
               "title": "Gain1",
               "default": 1,
               "type": "number"
            },
            "gain2": {
               "title": "Gain2",
               "default": 1,
               "type": "number"
            },
            "scaling": {
               "title": "Scaling",
               "default": 1,
               "type": "number"
            },
            "chopper": {
               "title": "Chopper",
               "default": false,
               "type": "boolean"
            },
            "dipole_dist": {
               "title": "Dipole Dist",
               "default": 1,
               "type": "number"
            },
            "sensor_calibration_file": {
               "title": "Sensor Calibration File",
               "default": "",
               "type": "string"
            },
            "instrument_calibration_file": {
               "title": "Instrument Calibration File",
               "default": "",
               "type": "string"
            }
         },
         "required": [
            "name"
         ]
      },
      "SpectraLevelMetadata": {
         "title": "SpectraLevelMetadata",
```

(continues on next page)

(continued from previous page)


```
        "description": "Metadata for spectra of a windowed decimation level",
        "type": "object",
        "properties": {
           "fs": {
              "title": "Fs",
              "type": "number"
           },
           "n_wins": {
              "title": "N Wins",
              "type": "integer"
           },
           "win_size": {
              "title": "Win Size",
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "olap_size": {
              "title": "Olap Size",
              "exclusiveMinimum": 0,
              "type": "integer"
           },
           "index_offset": {
              "title": "Index Offset",
              "type": "integer"
           },
           "n_freqs": {
              "title": "N Freqs",
              "type": "integer"
           },
           "freqs": {
              "title": "Freqs",
              "type": "array",
              "items": {
                 "type": "number"
              }
           }
        },
        "required": [
           "fs",
           "n_wins",
           "win_size",
           "olap_size",
           "index_offset",
           "n_freqs",
           "freqs"
        ]
     },
     "Record": {
        "title": "Record",
        "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
```


(continues on next page)

(continued from previous page)

```
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
        "type": "object",
        "properties": {
           "time_local": {
              "title": "Time Local",
              "type": "string",
              "format": "date-time"
           },
           "time_utc": {
              "title": "Time Utc",
              "type": "string",
              "format": "date-time"
           },
           "creator": {
              "title": "Creator",
              "type": "object"
           },
           "messages": {
              "title": "Messages",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "record_type": {
              "title": "Record Type",
              "type": "string"
           }
        },
        "required": [
           "creator",
           "messages",
           "record_type"
        ]
     },
     "History": {
        "title": "History",
        "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n    ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
```

(continues on next page)

(continued from previous page)

```
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
         "type": "object",
         "properties": {
            "records": {
               "title": "Records",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/Record"
               }
            }
         }
      }
   }
}
```

field fs: `List[float]` **[Required]**

field chans: `List[str]` **[Required]**

field n_chans: `int | None` = **None**

Validated by

- validate_n_chans

field n_levels: `int` **[Required]**

field first_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = [‘2021-01-01 00:00:00.000061_035156_250000_000000’]

field last_time: *HighResDateTime* **[Required]**

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = [‘2021-01-01 00:00:00.000061_035156_250000_000000’]

field system: `str` = **''**

field serial: `str` = **''**

field wgs84_latitude: `float` = **-999.0**

field wgs84_longitude: `float` = **-999.0**

field easting: `float` = **-999.0**

field northing: `float` = **-999.0**

field elevation: float = -999.0

field chans_metadata: Dict[str, ChanMetadata] [Required]

field levels_metadata: List[SpectraLevelMetadata] [Required]

field ref_time: HighResDateTime [Required]

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field history: History = History(records=[])

class resistics.spectra.**SpectraData**(*metadata:* SpectraMetadata, *data: Dict[int, ndarray]*)

Bases: ResisticsData

Class for holding spectra data

The spectra data is stored in the class as a dictionary mapping decimation level to numpy array. The shape of the array for each decimation level is:

n_wins x n_chans x n_freqs

get_level(*level: int*) → ndarray

Get the spectra data for a decimation level

get_chan(*level: int*, *chan: str*) → ndarray

Get the channel spectra data for a decimation level

get_chans(*level: int*, *chans: List[str]*) → ndarray

Get the channels spectra data for a decimation level

get_freq(*level: int*, *idx: int*) → ndarray

Get the spectra data at a frequency index for a decimation level

get_mag_phs(*level: int*, *unwrap: bool = False*) → Tuple[ndarray, ndarray]

Get magnitude and phase for a decimation level

get_timestamps(*level: int*) → DatetimeIndex

Get the start time of each window

Note that this does not use high resolution timestamps

Parameters
level (*int*) – The decimation level

Returns
The starts of each window

Return type
pd.DatetimeIndex

Raises
ValueError – If the level is out of range

plot(*max_pts: int | None = 10000*) → Figure

Stack spectra data for all decimation levels

Parameters
max_pts (*Optional[int], optional*) – The maximum number of points in any individual plot before applying lttbc downsampling, by default 10_000. If set to None, no downsampling will be applied.

Returns
The plotly figure

Return type
go.Figure

plot_level_stack(*level: int*, *max_pts: int = 10000*, *grouping: str | None = None*, *offset: str = '0h'*) → Figure

Stack the spectra for a decimation level with optional time grouping

Parameters

- **level** (*int*) – The decimation level
- **max_pts** (*int, optional*) – The maximum number of points in any individual plot before applying lttbc downsampling, by default 10_000
- **grouping** (*Optional[str], optional*) – A grouping interval as a pandas freq string, by default None
- **offset** (*str, optional*) – A time offset to add to the grouping, by default "0h". For instance, to plot night time and day time spectra, set grouping to "12h" and offset to "6h"

Returns
The plotly figure

Return type
go.Figure

plot_level_section(*level: int*, *grouping='30T'*) → Figure

Plot a spectra section

Parameters

- **level** (*int*) – The decimation level to plot
- **grouping** (*str, optional*) – The time domain resolution, by default "30T"

Returns
A plotly figure

Return type
go.Figure

pydantic model resistics.spectra.**FourierTransform**

Bases: *ResisticsProcess*

Perform a Fourier transform of the windowed data

The processor is inspired by the scipy.signal.stft function which performs a similar process and involves a Fourier transform along the last axis of the windowed data.

Parameters

- **win_fnc** (*Union[str, Tuple[str, float]]*) – The window to use before performing the FFT, by default ("kaiser", 14)

- **detrend** (*Union[str, None]*) – Type of detrending to apply before performing FFT, by default linear detrend. Setting to None will not apply any detrending to the data prior to the FFT
- **workers** (*int*) – The number of CPUs to use, by default max - 2

Examples

This example will get periodic decimated data, perfrom windowing and run the Fourier transform on the windowed data.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from resistics.testing import decimated_data_periodic
>>> from resistics.window import WindowSetup, Windower
>>> from resistics.spectra import FourierTransform
>>> frequencies = {"chan1": [870, 590, 110, 32, 12], "chan2": [480, 375, 210, 60,␣
↪45]}
>>> dec_data = decimated_data_periodic(frequencies, fs=128)
>>> dec_data.metadata.chans
['chan1', 'chan2']
>>> print(dec_data.to_string())
<class 'resistics.decimate.DecimatedData'>
           fs        dt  n_samples           first_time                         last_
↪time
level
0      2048.0  0.000488      16384  2021-01-01 00:00:00  2021-01-01 00:00:07.
↪99951171875
1       512.0  0.001953       4096  2021-01-01 00:00:00    2021-01-01 00:00:07.
↪998046875
2       128.0  0.007812       1024  2021-01-01 00:00:00      2021-01-01 00:00:07.
↪9921875
```

Perform the windowing

```
>>> win_params = WindowSetup().run(dec_data.metadata.n_levels, dec_data.metadata.fs)
>>> win_data = Windower().run(dec_data.metadata.first_time, win_params, dec_data)
```

And then the Fourier transform. By default, the data will be (linearly) detrended and mutliplied by a Kaiser window prior to the Fourier transform

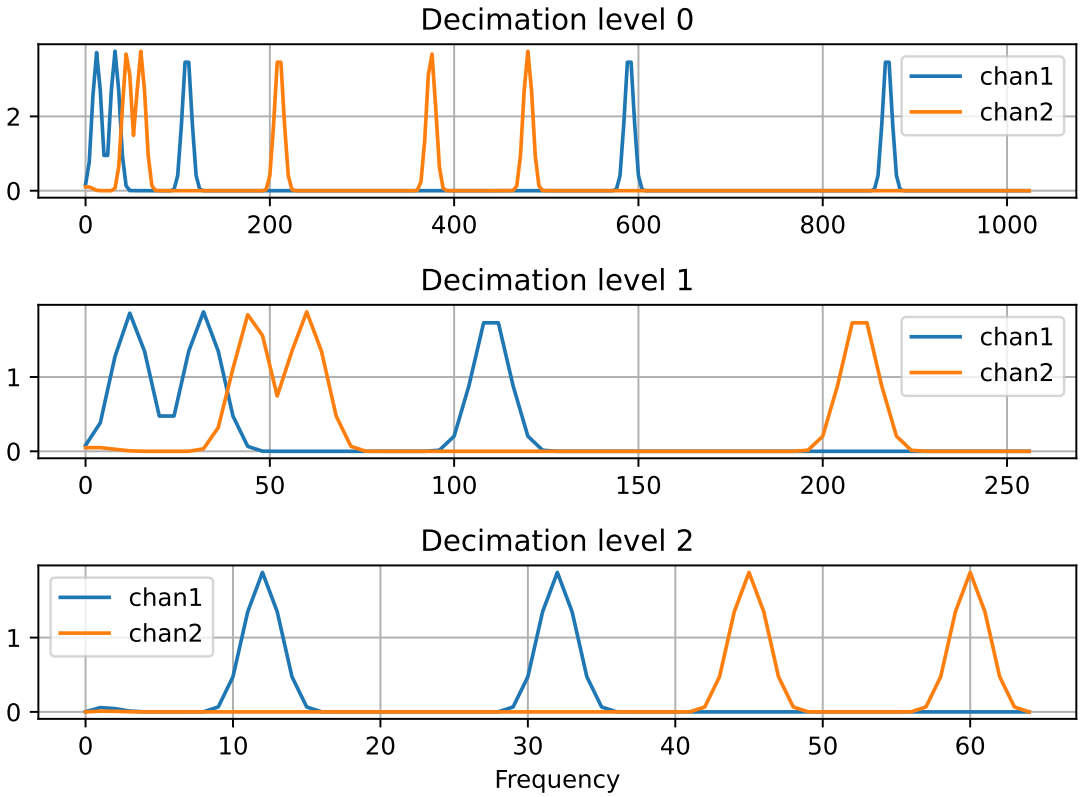
```
>>> spec_data = FourierTransform().run(win_data)
```

For plotting of magnitude, let's stack the spectra

```
>>> freqs_0 = spec_data.metadata.levels_metadata[0].freqs
>>> data_0 = np.absolute(spec_data.data[0]).mean(axis=0)
>>> freqs_1 = spec_data.metadata.levels_metadata[1].freqs
>>> data_1 = np.absolute(spec_data.data[1]).mean(axis=0)
>>> freqs_2 = spec_data.metadata.levels_metadata[2].freqs
>>> data_2 = np.absolute(spec_data.data[2]).mean(axis=0)
```

Now plot

```
>>> plt.subplot(3,1,1)
>>> plt.plot(freqs_0, data_0[0], label="chan1")
>>> plt.plot(freqs_0, data_0[1], label="chan2")
>>> plt.grid()
>>> plt.title("Decimation level 0")
>>> plt.legend()
>>> plt.subplot(3,1,2)
>>> plt.plot(freqs_1, data_1[0], label="chan1")
>>> plt.plot(freqs_1, data_1[1], label="chan2")
>>> plt.grid()
>>> plt.title("Decimation level 1")
>>> plt.legend()
>>> plt.subplot(3,1,3)
>>> plt.plot(freqs_2, data_2[0], label="chan1")
>>> plt.plot(freqs_2, data_2[1], label="chan2")
>>> plt.grid()
>>> plt.title("Decimation level 2")
>>> plt.legend()
>>> plt.xlabel("Frequency")
>>> plt.tight_layout()
>>> plt.show()
```




```
{
   "title": "FourierTransform",
```

(continues on next page)

(continued from previous page)

```
  "description": "Perform a Fourier transform of the windowed data\n\nThe␣
↪processor is inspired by the scipy.signal.stft function which performs\na similar␣
↪process and involves a Fourier transform along the last axis of\nthe windowed␣
↪data.\n\nParameters\n----------\nwin_fnc : Union[str, Tuple[str, float]]\n    The␣
↪window to use before performing the FFT, by default (\"kaiser\", 14)\ndetrend :␣
↪Union[str, None]\n    Type of detrending to apply before performing FFT, by␣
↪default linear\n    detrend. Setting to None will not apply any detrending to the␣
↪data prior\n    to the FFT\nworkers : int\n    The number of CPUs to use, by␣
↪default max - 2\n\nExamples\n--------\nThis example will get periodic decimated␣
↪data, perfrom windowing and run the\nFourier transform on the windowed data.\n\n..
↪ plot::\n    :width: 90%\n\n    >>> import matplotlib.pyplot as plt\n    >>>␣
↪import numpy as np\n    >>> from resistics.testing import decimated_data_periodic\
↪n    >>> from resistics.window import WindowSetup, Windower\n    >>> from␣
↪resistics.spectra import FourierTransform\n    >>> frequencies = {\"chan1\": [870,
↪ 590, 110, 32, 12], \"chan2\": [480, 375, 210, 60, 45]}\n    >>> dec_data =␣
↪decimated_data_periodic(frequencies, fs=128)\n    >>> dec_data.metadata.chans\n  ␣
↪ ['chan1', 'chan2']\n    >>> print(dec_data.to_string())\n    <class 'resistics.
↪decimate.DecimatedData'>\n               fs        dt  n_samples           first_
↪time                      last_time\n    level\n    0      2048.0  0.000488    ␣
↪ 16384  2021-01-01 00:00:00  2021-01-01 00:00:07.99951171875\n    1       512.0 ␣
↪0.001953       4096  2021-01-01 00:00:00    2021-01-01 00:00:07.998046875\n    2 ␣
↪     128.0  0.007812       1024  2021-01-01 00:00:00      2021-01-01 00:00:07.
↪9921875\n\n    Perform the windowing\n\n    >>> win_params = WindowSetup().
↪run(dec_data.metadata.n_levels, dec_data.metadata.fs)\n    >>> win_data =␣
↪Windower().run(dec_data.metadata.first_time, win_params, dec_data)\n\n    And␣
↪then the Fourier transform. By default, the data will be (linearly)\n   ␣
↪detrended and mutliplied by a Kaiser window prior to the Fourier\n    transform\n\
↪n    >>> spec_data = FourierTransform().run(win_data)\n\n    For plotting of␣
↪magnitude, let's stack the spectra\n\n    >>> freqs_0 = spec_data.metadata.levels_
↪metadata[0].freqs\n    >>> data_0 = np.absolute(spec_data.data[0]).mean(axis=0)\n␣
↪    >>> freqs_1 = spec_data.metadata.levels_metadata[1].freqs\n    >>> data_1 = np.
↪absolute(spec_data.data[1]).mean(axis=0)\n    >>> freqs_2 = spec_data.metadata.
↪levels_metadata[2].freqs\n    >>> data_2 = np.absolute(spec_data.data[2]).
↪mean(axis=0)\n\n    Now plot\n\n    >>> plt.subplot(3,1,1) # doctest: +SKIP\n    >
↪>> plt.plot(freqs_0, data_0[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_0, data_0[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 0\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.subplot(3,1,2) # doctest: +SKIP\n    >>> plt.
↪plot(freqs_1, data_1[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_1, data_1[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 1\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.subplot(3,1,3) # doctest: +SKIP\n    >>> plt.
↪plot(freqs_2, data_2[0], label=\"chan1\") # doctest: +SKIP\n    >>> plt.
↪plot(freqs_2, data_2[1], label=\"chan2\") # doctest: +SKIP\n    >>> plt.grid()\n ␣
↪  >>> plt.title(\"Decimation level 2\") # doctest: +SKIP\n    >>> plt.legend() #␣
↪doctest: +SKIP\n    >>> plt.xlabel(\"Frequency\") # doctest: +SKIP\n    >>> plt.
↪tight_layout() # doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP",
  "type": "object",
  "properties": {
    "name": {
      "title": "Name",
      "type": "string"
```

(continues on next page)

(continued from previous page)

```
      },
      "win_fnc": {
         "title": "Win Fnc",
         "default": [
            "kaiser",
            14
         ],
         "anyOf": [
            {
               "type": "string"
            },
            {
               "type": "array",
               "minItems": 2,
               "maxItems": 2,
               "items": [
                  {
                     "type": "string"
                  },
                  {
                     "type": "number"
                  }
               ]
            }
         ]
      },
      "detrend": {
         "title": "Detrend",
         "default": "linear",
         "type": "string"
      },
      "workers": {
         "title": "Workers",
         "default": -2,
         "type": "integer"
      }
   }
}
```

field win_fnc: `str | Tuple[str, float] = ('kaiser', 14)`

field detrend: `str | None = 'linear'`

field workers: `int = -2`

run(*win_data:* WindowedData) → *SpectraData*

Perform the FFT

Data is padded to the next fast length before performing the FFT to speed up processing. Therefore, the output length may not be as expected.

Parameters

win_data (WindowedData) – The input windowed data

Returns
The Fourier transformed output

Return type
SpectraData

pydantic model resistics.spectra.**EvaluationFreqs**

Bases: *ResisticsProcess*

Calculate the spectra values at the evaluation frequencies

This is done using linear interpolation in the complex domain

Example

The example will show interpolation to evaluation frequencies on a very simple example. Begin by generating some example spectra data.

```
>>> from resistics.decimate import DecimationSetup
>>> from resistics.spectra import EvaluationFreqs
>>> from resistics.testing import spectra_data_basic
>>> spec_data = spectra_data_basic()
>>> spec_data.metadata.n_levels
1
>>> spec_data.metadata.chans
['chan1']
>>> spec_data.metadata.levels_metadata[0].summary()
{
    'fs': 180.0,
    'n_wins': 2,
    'win_size': 20,
    'olap_size': 5,
    'index_offset': 0,
    'n_freqs': 10,
    'freqs': [0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0]
}
```

The spectra data has only a single channel and a single level which has 2 windows. Now define our evaluation frequencies.

```
>>> eval_freqs = [1, 12, 23, 34, 45, 56, 67, 78, 89]
>>> dec_setup = DecimationSetup(n_levels=1, per_level=9, eval_freqs=eval_freqs)
>>> dec_params = dec_setup.run(spec_data.metadata.fs[0])
>>> dec_params.summary()
{
    'fs': 180.0,
    'n_levels': 1,
    'per_level': 9,
    'min_samples': 256,
    'eval_freqs': [1.0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0],
    'dec_factors': [1],
    'dec_increments': [1],
    'dec_fs': [180.0]
}
```

Now calculate the spectra at the evaluation frequencies

```
>>> eval_data = EvaluationFreqs().run(dec_params, spec_data)
>>> eval_data.metadata.levels_metadata[0].summary()
{
    'fs': 180.0,
    'n_wins': 2,
    'win_size': 20,
    'olap_size': 5,
    'index_offset': 0,
    'n_freqs': 9,
    'freqs': [1.0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0]
}
```

To double check everything is as expected, let's compare the data. Comparing window 1 gives

```
>>> print(spec_data.data[0][0, 0])
[0.+0.j 1.+1.j 2.+2.j 3.+3.j 4.+4.j 5.+5.j 6.+6.j 7.+7.j 8.+8.j 9.+9.j]
>>> print(eval_data.data[0][0, 0])
[0.1+0.1j 1.2+1.2j 2.3+2.3j 3.4+3.4j 4.5+4.5j 5.6+5.6j 6.7+6.7j 7.8+7.8j
 8.9+8.9j]
```

And window 2

```
>>> print(spec_data.data[0][1, 0])
[-1. +1.j  0. +2.j  1. +3.j  2. +4.j  3. +5.j  4. +6.j  5. +7.j  6. +8.j
  7. +9.j  8.+10.j]
>>> print(eval_data.data[0][1, 0])
[-0.9+1.1j  0.2+2.2j  1.3+3.3j  2.4+4.4j  3.5+5.5j  4.6+6.6j  5.7+7.7j
  6.8+8.8j  7.9+9.9j]
```


```
{
   "title": "EvaluationFreqs",
   "description": "Calculate the spectra values at the evaluation frequencies\n\
↪nThis is done using linear interpolation in the complex domain\n\nExample\n-------
↪\nThe example will show interpolation to evaluation frequencies on a very\nsimple␣
↪example. Begin by generating some example spectra data.\n\n>>> from resistics.
↪decimate import DecimationSetup\n>>> from resistics.spectra import␣
↪EvaluationFreqs\n>>> from resistics.testing import spectra_data_basic\n>>> spec_
↪data = spectra_data_basic()\n>>> spec_data.metadata.n_levels\n1\n>>> spec_data.
↪metadata.chans\n['chan1']\n>>> spec_data.metadata.levels_metadata[0].summary()\n{\
↪n    'fs': 180.0,\n    'n_wins': 2,\n    'win_size': 20,\n    'olap_size': 5,\n  ␣
↪ 'index_offset': 0,\n    'n_freqs': 10,\n    'freqs': [0.0, 10.0, 20.0, 30.0, 40.
↪0, 50.0, 60.0, 70.0, 80.0, 90.0]\n}\n\nThe spectra data has only a single channel␣
↪and a single level which has 2\nwindows. Now define our evaluation frequencies.\n\
↪n>>> eval_freqs = [1, 12, 23, 34, 45, 56, 67, 78, 89]\n>>> dec_setup =␣
↪DecimationSetup(n_levels=1, per_level=9, eval_freqs=eval_freqs)\n>>> dec_params =␣
↪dec_setup.run(spec_data.metadata.fs[0])\n>>> dec_params.summary()\n{\n    'fs':␣
↪180.0,\n    'n_levels': 1,\n    'per_level': 9,\n    'min_samples': 256,\n
↪'eval_freqs': [1.0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0],\n    'dec_
↪factors': [1],\n    'dec_increments': [1],\n    'dec_fs': [180.0]\n}\n\nNow␣
↪calculate the spectra at the evaluation frequencies\n\n>>> eval_data =␣
↪EvaluationFreqs().run(dec_params, spec_data)\n>>> eval_data.metadata.levels_
```


(continues on next page)

(continued from previous page)

```
↪metadata[0].summary()\n{\n    'fs': 180.0,\n    'n_wins': 2,\n    'win_size': 20,\
↪n    'olap_size': 5,\n    'index_offset': 0,\n    'n_freqs': 9,\n    'freqs': [1.
↪0, 12.0, 23.0, 34.0, 45.0, 56.0, 67.0, 78.0, 89.0]\n}\n\nTo double check␣
↪everything is as expected, let's compare the data. Comparing\nwindow 1 gives\n\n>>
↪> print(spec_data.data[0][0, 0])\n[0.+0.j 1.+1.j 2.+2.j 3.+3.j 4.+4.j 5.+5.j 6.+6.
↪j 7.+7.j 8.+8.j 9.+9.j]\n>>> print(eval_data.data[0][0, 0])\n[0.1+0.1j 1.2+1.2j 2.
↪3+2.3j 3.4+3.4j 4.5+4.5j 5.6+5.6j 6.7+6.7j 7.8+7.8j\n 8.9+8.9j]\n\nAnd window 2\n\
↪n>>> print(spec_data.data[0][1, 0])\n[-1. +1.j  0. +2.j  1. +3.j  2. +4.j  3. +5.
↪j  4. +6.j  5. +7.j  6. +8.j\n  7. +9.j  8.+10.j]\n>>> print(eval_data.data[0][1,␣
↪0])\n[-0.9+1.1j  0.2+2.2j  1.3+3.3j  2.4+4.4j  3.5+5.5j  4.6+6.6j  5.7+7.7j\n  6.
↪8+8.8j  7.9+9.9j]",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```

run(*dec_params:* DecimationParameters, *spec_data:* SpectraData) → *SpectraData*

Interpolate spectra data to the evaluation frequencies

This is a simple linear interpolation.

Parameters

- **dec_params** (DecimationParameters) – The decimation parameters which have the evaluation frequencies for each decimation level
- **spec_data** (SpectraData) – The spectra data

Returns

The spectra data at the evaluation frequencies

Return type

SpectraData

pydantic model resistics.spectra.**SpectraDataWriter**

Bases: *ResisticsWriter*

Writer of resistics spectra data

```
{
   "title": "SpectraDataWriter",
   "description": "Writer of resistics spectra data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "overwrite": {
         "title": "Overwrite",
         "default": true,
```

(continues on next page)

(continued from previous page)

```
            "type": "boolean"
        }
    }
}
```

run(*dir_path: Path*, *spec_data:* SpectraData) → None

Write out SpectraData

Parameters

- **dir_path** (*Path*) – The directory path to write to
- **spec_data** (SpectraData) – Spectra data to write out

Raises

WriteError – If unable to write to the directory

pydantic model resistics.spectra.**SpectraDataReader**

Bases: *ResisticsProcess*

Reader of resistics spectra data

```
{
   "title": "SpectraDataReader",
   "description": "Reader of resistics spectra data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```

run(*dir_path: Path*, *metadata_only: bool = False*) → *SpectraMetadata* | *SpectraData*

Read SpectraData

Parameters

- **dir_path** (*Path*) – The directory path to read from
- **metadata_only** (*bool, optional*) – Flag for getting metadata only, by default False

Returns

The SpectraData or SpectraMetadata if metadata_only is True

Return type

Union[*SpectraMetadata*, *SpectraData*]

Raises

ReadError – If the directory does not exist

pydantic model resistics.spectra.**SpectraProcess**

Bases: *ResisticsProcess*

Parent class for spectra processes

```
{
   "title": "SpectraProcess",
   "description": "Parent class for spectra processes",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*spec_data:* SpectraData) → *SpectraData*

Run a spectra processor

pydantic model resistics.spectra.**SpectraSmootherUniform**

Bases: *SpectraProcess*

Smooth a spectra with a uniform filter

For more information, please refer to: https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.uniform_filter1d.html

Examples

Smooth a simple spectra data instance

```
>>> from resistics.spectra import SpectraSmootherUniform
>>> from resistics.testing import spectra_data_basic
>>> spec_data = spectra_data_basic()
>>> smooth_data = SpectraSmootherUniform(length_proportion=0.5).run(spec_data)
```

Look at the results for the two windows

```
>>> spec_data.data[0][0,0]
array([0.+0.j, 1.+1.j, 2.+2.j, 3.+3.j, 4.+4.j, 5.+5.j, 6.+6.j, 7.+7.j,
       8.+8.j, 9.+9.j])
>>> smooth_data.data[0][0,0]
array([0.8+0.8j, 1.2+1.2j, 2. +2.j , 3. +3.j , 4. +4.j , 5. +5.j ,
       6. +6.j , 7. +7.j , 7.8+7.8j, 8.2+8.2j])
```


```
{
   "title": "SpectraSmootherUniform",
   "description": "Smooth a spectra with a uniform filter\n\nFor more information,␣
↪please refer to:\nhttps://docs.scipy.org/doc/scipy/reference/generated/scipy.
↪ndimage.uniform_filter1d.html\n\nExamples\n--------\nSmooth a simple spectra data␣
↪instance\n\n>>> from resistics.spectra import SpectraSmootherUniform\n>>> from␣
↪resistics.testing import spectra_data_basic\n>>> spec_data = spectra_data_basic()\
↪n>>> smooth_data = SpectraSmootherUniform(length_proportion=0.5).run(spec_data)\n\
↪nLook at the results for the two windows\n\n>>> spec_data.data[0][0,0]\narray([0.
↪+0.j, 1.+1.j, 2.+2.j, 3.+3.j, 4.+4.j, 5.+5.j, 6.+6.j, 7.+7.j,\n       8.+8.j, 9.
↪+9.j])\n>>> smooth_data.data[0][0,0]\narray([0.8+0.8j, 1.2+1.2j, 2. +2.j , 3. +3.
↪j , 4. +4.j , 5. +5.j ,\n       6. +6.j , 7. +7.j , 7.8+7.8j, 8.2+8.2j])",
```


(continues on next page)

(continued from previous page)

```
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "length_proportion": {
         "title": "Length Proportion",
         "default": 0.1,
         "type": "number"
      }
   }
}
```

field length_proportion: float = 0.1

run(*spec_data:* SpectraData) → *SpectraData*

Smooth spectra data with a uniform smoother

Parameters
spec_data (SpectraData) – The input spectra data

Returns
The output spectra data

Return type
SpectraData

pydantic model resistics.spectra.**SpectraSmootherGaussian**

Bases: *SpectraProcess*

Smooth a spectra with a gaussian filter

For more information, please refer to: https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian_filter1d.html

Examples

Smooth a simple spectra data instance

```
>>> from resistics.spectra import SpectraSmootherGaussian
>>> from resistics.testing import spectra_data_basic
>>> spec_data = spectra_data_basic()
>>> smooth_data = SpectraSmootherGaussian().run(spec_data)
```

Look at the results for the two windows

```
>>> spec_data.data[0][0,0]
array([0.+0.j, 1.+1.j, 2.+2.j, 3.+3.j, 4.+4.j, 5.+5.j, 6.+6.j, 7.+7.j,
       8.+8.j, 9.+9.j])
>>> smooth_data.data[0][0,0]
array([1.93603671+1.93603671j, 2.1921536 +2.1921536j ,
       2.67507336+2.67507336j, 3.33255376+3.33255376j,
       4.09862656+4.09862656j, 4.90137344+4.90137344j,
```

(continues on next page)

(continued from previous page)

```
       5.66744624+5.66744624j, 6.32492664+6.32492664j,
       6.8078464 +6.8078464j , 7.06396329+7.06396329j])
```


```
{
   "title": "SpectraSmootherGaussian",
   "description": "Smooth a spectra with a gaussian filter\n\nFor more information,␣
↪please refer to:\nhttps://docs.scipy.org/doc/scipy/reference/generated/scipy.
↪ndimage.gaussian_filter1d.html\n\nExamples\n--------\nSmooth a simple spectra␣
↪data instance\n\n>>> from resistics.spectra import SpectraSmootherGaussian\n>>>␣
↪from resistics.testing import spectra_data_basic\n>>> spec_data = spectra_data_
↪basic()\n>>> smooth_data = SpectraSmootherGaussian().run(spec_data)\n\nLook at␣
↪the results for the two windows\n\n>>> spec_data.data[0][0,0]\narray([0.+0.j, 1.
↪+1.j, 2.+2.j, 3.+3.j, 4.+4.j, 5.+5.j, 6.+6.j, 7.+7.j,\n       8.+8.j, 9.+9.j])\n>>
↪> smooth_data.data[0][0,0]\narray([1.93603671+1.93603671j, 2.1921536 +2.1921536j ,
↪\n       2.67507336+2.67507336j, 3.33255376+3.33255376j,\n       4.09862656+4.
↪09862656j, 4.90137344+4.90137344j,\n       5.66744624+5.66744624j, 6.32492664+6.
↪32492664j,\n       6.8078464 +6.8078464j , 7.06396329+7.06396329j])",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "sigma": {
         "title": "Sigma",
         "default": 3,
         "type": "number"
      }
   }
}
```


field sigma: float = 3

run(*spec_data:* SpectraData) → *SpectraData*

Run Gaussian filtering of spectra data

Parameters
spec_data (SpectraData) – Input spectra data

Returns
Output spectra data

Return type
SpectraData

resistics.testing module

Module for producing testing data for resistics and helper functions to compare instances of the same object.

This includes testing data for:

- Record
- History
- TimeMetadata
- TimeData
- DecimatedData
- SpectraData
- Evaluation frequency SpectraData
- RegressionInputMetadata
- Solution

resistics.testing.**record_example1**() → *Record*

Get an example Record

resistics.testing.**record_example2**() → *Record*

Get an example Record

resistics.testing.**history_example**() → *History*

Get a History example

resistics.testing.**time_metadata_1chan**(*fs: float = 10*, *first_time: str = '2021-01-01 00:00:00'*, *n_samples: int = 11*) → *TimeMetadata*

Get TimeMetadata for a single channel, "chan1"

Parameters

- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*str, optional*) – The first time, by default "2021-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 11

Returns

TimeMetadata

Return type

TimeMetadata

resistics.testing.**time_metadata_2chan**(*fs: float = 10*, *first_time: str = '2021-01-01 00:00:00'*, *n_samples: int = 11*) → *TimeMetadata*

Get a TimeMetadata instance with two channels, "chan1" and "chan2"

Parameters

- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*str, optional*) – The first time, by default "2021-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 11

Returns

TimeMetadata

Return type
TimeMetadata

resistics.testing.**time_metadata_general**(*chans: List[str]*, *fs: float = 10*, *first_time: str = '2020-01-01 00:00:00'*, *n_samples: int = 11*) → *TimeMetadata*

Get general time metadata

Parameters

- **chans** (*List[str]*) – The channels in the time data
- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*_type_, optional*) – The time of the first sample, by default "2020-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 11

Returns
An instance of TimeMetadata with the approripate properties

Return type
TimeMetadata

resistics.testing.**time_metadata_mt**(*fs: float = 10*, *first_time: str = '2020-01-01 00:00:00'*, *n_samples: int = 11*) → *TimeMetadata*

Get a magnetotelluric time metadata with four channels "Ex", "Ey", "Hx", "Hy"

Parameters

- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*str, optional*) – The first time, by default "2020-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 11

Returns
TimeMetadata

Return type
TimeMetadata

resistics.testing.**time_data_ones**(*fs: float = 10*, *first_time: str = '2020-01-01 00:00:00'*, *n_samples: int = 10*, *dtype: Type | None = None*) → *TimeData*

TimeData with all ones

Parameters

- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*str, optional*) – The time of the first sample, by default "2020-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 10
- **dtype** (*Optional[Type], optional*) – The data type for the values, by default None

Returns
The TimeData

Return type
TimeData

resistics.testing.**time_data_simple**(*fs: float = 10*, *first_time: str = '2020-01-01 00:00:00'*, *dtype: Type | None = None*) → *TimeData*

Time data with 16 samples

Parameters

- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*str, optional*) – The time of the first sample, by default "2020-01-01 00:00:00"
- **dtype** (*Optional[Type], optional*) – The data type for the values, by default None

Returns
The TimeData

Return type
TimeData

resistics.testing.**time_data_with_nans**(*fs: float = 10*, *first_time: str = '2020-01-01 00:00:00'*, *dtype: Type | None = None*) → *TimeData*

TimeData with 16 samples and some nan values

Parameters

- **fs** (*float, optional*) – Sampling frequency, by default 10
- **first_time** (*str, optional*) – The time of the first sample, by default "2020-01-01 00:00:00"
- **dtype** (*Optional[Type], optional*) – The data type for the values, by default None

Returns
The TimeData

Return type
TimeData

resistics.testing.**time_data_linear**(*fs: float = 10*, *first_time: str = '2020-01-01 00:00:00'*, *n_samples: int = 10*, *dtype: Type | None = None*) → *TimeData*

Get TimeData with linear data

Parameters

- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*str, optional*) – Time of first sample, by default "2020-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 10
- **dtype** (*Optional[Type], optional*) – The data type for the values, by default None

Returns
TimeData with linear values

Return type
TimeData

resistics.testing.**time_data_random**(*fs: float = 10*, *first_time: str = '2020-01-01 00:00:00'*, *n_samples: int = 10*, *dtype: Type | None = None*) → *TimeData*

TimeData with random values and specifiable number of samples

Parameters

- **fs** (*float, optional*) – The sampling frequency, by default 10

- **first_time** (*str, optional*) – Time of first sample, by default "2020-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 10
- **dtype** (*Optional[Type], optional*) – The data type for the values, by default None

Returns
The TimeData

Return type
TimeData

resistics.testing.**time_data_periodic**(*frequencies: List[float], fs: float = 50, first_time: str = '2020-01-01 00:00:00', n_samples: int = 100, dtype: Type | None = None*) → *TimeData*

Get period TimeData

Parameters

- **frequencies** (*List[float]*) – Frequencies to include
- **fs** (*float, optional*) – Sampling frequency, by default 50
- **first_time** (*str, optional*) – The first time, by default "2020-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 100
- **dtype** (*Optional[Type], optional*) – The data type for the values, by default None

Returns
Periodic TimeData

Return type
TimeData

resistics.testing.**time_data_with_offset**(*offset=0.05, fs: float = 10, first_time: str = '2020-01-01 00:00:00', n_samples: int = 11, dtype: Type | None = None*) → *TimeData*

Get TimeData with an offset on the sampling

Parameters

- **offset** (*float, optional*) – The offset on the sampling in seconds, by default 0.05
- **fs** (*float, optional*) – The sampling frequency, by default 10
- **first_time** (*str, optional*) – The first time of the TimeData, by default "2020-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 11
- **dtype** (*Optional[Type], optional*) – The data type for the values, by default None

Returns
The TimeData

Return type
TimeData

resistics.testing.**decimated_metadata**(*fs: float = 0.25, first_time: str = '2021-01-01 00:00:00', n_samples: int = 1024, n_levels: int = 3, factor: int = 4*) → *DecimatedMetadata*

Get example decimated metadata

The final level has n_samples. The number of samples for all other levels is calculated using a decimation factor of 4.

Similarly for the sampling frequencies, the final level is assumed to have a sample frequency of fs and all other levels sampling frequencies are calculated from there.

Parameters

- **fs** (*float, optional*) – The sampling frequency of the last level, by default 0.25
- **first_time** (*str, optional*) – The time of the first sample, by default "2021-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 1024
- **n_levels** (*int, optional*) – The number of decimation levels, by default 3
- **factor** (*int, optional*) – The decimation factor for each level, by default 4

Returns
DecimatedMetadata

Return type
DecimatedMetadata

resistics.testing.**decimated_data_random**(*fs: float = 0.25, first_time: str = '2021-01-01 00:00:00', n_samples: int = 1024, n_levels: int = 3, factor: int = 4*) → *DecimatedData*

Get random decimated data

Parameters

- **fs** (*float, optional*) – Sampling frequency, by default 10
- **first_time** (*str, optional*) – The time of the first sample, by default "2021-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 1024
- **n_levels** (*int, optional*) – The number of levels, by default 3
- **factor** (*int, optional*) – The decimation factor for each level, by default 4

Returns
The decimated data

Return type
DecimatedData

resistics.testing.**decimated_data_linear**(*fs: float = 0.25, first_time: str = '2021-01-01 00:00:00', n_samples: int = 1024, n_levels: int = 3, factor: int = 4*)

Get linear decimated data

Parameters

- **fs** (*float, optional*) – Sampling frequency, by default 10
- **first_time** (*str, optional*) – The time of the first sample, by default "2021-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 1024
- **n_levels** (*int, optional*) – The number of levels, by default 3
- **factor** (*int, optional*) – The decimation factor for each level, by default 4

Returns
The decimated data

Return type
DecimatedData

resistics.testing.**decimated_data_periodic**(*frequencies: Dict[str, List[float]]*, *fs: float = 0.25*, *first_time: str = '2021-01-01 00:00:00'*, *n_samples: int = 1024*, *n_levels: int = 3*, *factor: int = 4*)

Get periodic decimated data

Parameters

- **frequencies** (*Dict[str, List[float]]*) – Mapping from channel to list of frequencies to add
- **fs** (*float, optional*) – Sampling frequency, by default 10
- **first_time** (*str, optional*) – The time of the first sample, by default "2021-01-01 00:00:00"
- **n_samples** (*int, optional*) – The number of samples, by default 1024
- **n_levels** (*int, optional*) – The number of levels, by default 3
- **factor** (*int, optional*) – The decimation factor for each level, by default 4

Returns
The decimated data

Return type
DecimatedData

resistics.testing.**spectra_metadata_multilevel**(*fs: float = 128*, *n_levels: int = 3*, *n_wins: List[int] | int = 2*, *index_offset: List[int] | int = 0*, *chans: List[str] | None = None*) → *SpectraMetadata*

Get spectra metadata with multiple levels and two channels

Parameters

- **fs** (*float, optional*) – The original sampling frequency, by default 128
- **n_levels** (*int, optional*) – The number of levels, by default 3
- **n_wins** (*Union[List[int], int]*) – The number of windows for each level
- **index_offset** (*Union[List[int], int], optional*) – The index offset vs. the reference time, by default 0
- **chans** (*Optional[List[str]]*) – The channels in the data, by default None. If None, the channels will be chan1 and chan2

Returns
SpectraMetadata with n_levels

Return type
SpectraMetadata

Raises
ValueError – If the number of user input channels does not equal two

resistics.testing.**spectra_data_basic**() → *SpectraData*

Spectra data with a single decimation level

Returns
Spectra data with a single level, a single channel and two windows

Return type
SpectraData

resistics.testing.**generate_evaluation_data**(*chans: List[str]*, *soln:* Solution, *n_wins: int*) → ndarray

Generate evaluation frequency data that satisfies a provided solution

The returned array has the shape: n_wins x n_chans x n_evals Which is close to the shape required for spectra data

There is an extra check provided to check if a channel appears in both the input and output channels, which could be a tricky scenario.

The data is produced randomly using np.random.randn, meaning that it is sampled from a standard normal distribution

Parameters

- **chans** (*List[str]*) – The channels in the data
- **soln** (Solution) – The Solution that needs to be satisfied
- **n_wins** (*int*) – The number of windows to generate

Returns
The evaluation frequency data array

Return type
np.ndarray

resistics.testing.**evaluation_data**(*fs: float*, *dec_params:* DecimationParameters, *n_wins: int*, *soln:* Solution) → *SpectraData*

Generate evaluation frequency data that will satisfy a given solution. This will generate random data between the low and high values

Parameters

- **fs** (*float*) – The sampling frequency of the original data
- **dec_params** (DecimationParameters) – The data decimation information
- **n_wins** (*int*) – The number of windows to generate
- **soln** (Solution) – The solution that the generated data should satisfy

Returns
The evaluation frequency data

Return type
SpectraData

Raises
ValueError – If the number of evaluation frequencies is not exactly divisible by the number of levels

resistics.testing.**transfer_function_random**(*n_in: int*, *n_out: int*) → *TransferFunction*

Generate a random transfer function

n_in and n_out must be less than or equal to 26 as the random samples are taken from the alphabet

Parameters

- **n_in** (*int*) – Number of input channels
- **n_out** (*int*) – Number of output channels

Returns
A randomly generated transfer function

Return type
TransferFunction

Raises
ValueError – If any of the channel names is duplicated

resistics.testing.**regression_input_metadata_single_site**(*fs: float*, *freqs: List[float]*, *tf:* TransferFunction) → *RegressionInputMetadata*

Given a transfer function, get example regression input metadata assuming a single site

Parameters

- **fs** (*float*) – The sampling frequency
- **freqs** (*List[float]*) – The evaluation frequencies
- **tf** (TransferFunction) – The transfer function for which to create RegressionInputMetadata

Returns
Example regression input metadata with fs=128 and 5 evaluation frequencies

Return type
RegressionInputMetadata

resistics.testing.**components_mt**() → Dict[str, *Component*]

Get example components for the Impedance Tensor

Returns
Dictionary of component values (ExHx, ExHy, EyHx, EyHy)

Return type
Dict[str, *Component*]

resistics.testing.**solution_mt**() → *Solution*

Get an example impedance tensor solution

Returns
The solution for an MT dataset

Return type
Solution

resistics.testing.**solution_general**(*fs: float*, *tf:* TransferFunction, *n_evals: int*, *components: Dict[str,* Component*]*) → *Solution*

Create a Solution instance from the specified components

Parameters

- **fs** (*float*) – The sampling frequency of the original data
- **tf** (TransferFunction) – The transfer function to be solved
- **n_evals** (*int*) – The number of evaluation frequencies
- **components** (*Dict[str, Component]*) – The components of the solution

Returns
The Solution instance

Return type
Solution

resistics.testing.**solution_random_int**(*fs: float*, *tf:* TransferFunction, *n_evals=10*, *low: int = -10*, *high: int = 10*) → *Solution*

Generate a set of random integer components for a solution

Parameters

- **fs** (*float*) – The original sampling frequency of the data
- **tf** (TransferFunction) – The transfer function
- **n_evals** (*int, optional*) – The number of evaluation frequencies, by default 10
- **low** (*int, optional*) – A low value for the integers, by default -10
- **high** (*int, optional*) – A high value for the integers, by default 10

Returns
A randomly generated solution for the transfer function

Return type
Solution

resistics.testing.**solution_random_float**(*fs: float*, *tf:* TransferFunction, *n_evals=10*) → *Solution*

Generate a set of random float components for a solution

This uses the numpy np.random.randn which generates numbers on a standard distribution and then multiplies that with a random integer between 0 and 10.

Parameters

- **fs** (*float*) – The original sampling frequency of the data
- **tf** (TransferFunction) – The transfer function
- **n_evals** (*int, optional*) – The number of evaluation frequencies, by default 10

Returns
A randomly generated solution for the transfer function

Return type
Solution

resistics.testing.**remove_record_times**(*records: Dict*) → Dict

Remove timestamps from records

Timestamps can make comparision of two data objects harder as processes need to have been run at exactly the same time for equality, which is unlikely to be the case in tests

Parameters
records (*Dict*) – The history records

Returns
The history records with timestamps removed

Return type
Dict

resistics.testing.**assert_time_data_equal**(*time_data1:* TimeData, *time_data2:* TimeData, *history_times: bool = True*)

Assert that two time data instances are equal

Parameters

- **time_data1** (TimeData) – Time data 1
- **time_data2** (TimeData) – Time data 2
- **history_times** (*bool, optional*) – Flag to include history timestamps in the comparison, by default True. Including timestamps will cause a failure if processes were not run at exactly the same time.

resistics.testing.**assert_soln_equal**(*soln1:* Solution, *soln2:* Solution)

Check that two solutions are nearly the same

Parameters

- **soln1** (Solution) – The first solution
- **soln2** (Solution) – The second solution

resistics.time module

Classes and methods for storing and manipulating time data, including:

- The TimeMetadata model for defining metadata for TimeData
- The TimeData class for storing TimeData
- Implementations of time data readers for numpy and ascii formatted TimeData
- TimeData processors

pydantic model resistics.time.**ChanMetadata**

Bases: *Metadata*

Channel metadata


```
{
   "title": "ChanMetadata",
   "description": "Channel metadata",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "data_files": {
         "title": "Data Files",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "chan_type": {
         "title": "Chan Type",
         "type": "string"
      },
      "chan_source": {
         "title": "Chan Source",
         "type": "string"
      },
```


(continues on next page)

(continued from previous page)


```
      "sensor": {
         "title": "Sensor",
         "default": "",
         "type": "string"
      },
      "serial": {
         "title": "Serial",
         "default": "",
         "type": "string"
      },
      "gain1": {
         "title": "Gain1",
         "default": 1,
         "type": "number"
      },
      "gain2": {
         "title": "Gain2",
         "default": 1,
         "type": "number"
      },
      "scaling": {
         "title": "Scaling",
         "default": 1,
         "type": "number"
      },
      "chopper": {
         "title": "Chopper",
         "default": false,
         "type": "boolean"
      },
      "dipole_dist": {
         "title": "Dipole Dist",
         "default": 1,
         "type": "number"
      },
      "sensor_calibration_file": {
         "title": "Sensor Calibration File",
         "default": "",
         "type": "string"
      },
      "instrument_calibration_file": {
         "title": "Instrument Calibration File",
         "default": "",
         "type": "string"
      }
   },
   "required": [
      "name"
   ]
}
```


field name: `str` **[Required]**

The name of the channel

field data_files: List[str] | None = None

The data files

Validated by

- validate_data_files

field chan_type: str | None = None

The channel type, electric, magnetic or unknown

Validated by

- validate_chan_type

field chan_source: str | None = None

The name of channel in the data source, can be ignored if not required

field sensor: str = ''

The name of the sensor

field serial: str = ''

The serial number of the sensor

field gain1: float = 1

Primary channel gain

field gain2: float = 1

Secondary channel gain

field scaling: float = 1

Scaling to apply to the data. May include the gains and other scaling

field chopper: bool = False

Boolean flag for chopper on

field dipole_dist: float = 1

Dipole spacing for the channel

field sensor_calibration_file: str = ''

Explicit name of sensor calibration file

field instrument_calibration_file: str = ''

Explicit name of instrument calibration file

electric() → bool

True if the channel is an electric channel

magnetic() → bool

True if the channel is a magnetic channel

pydantic model resistics.time.**TimeMetadata**

Bases: *WriteableMetadata*

Time metadata

```
{
   "title": "TimeMetadata",
   "description": "Time metadata",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "fs": {
         "title": "Fs",
         "type": "number"
      },
      "chans": {
         "title": "Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "n_chans": {
         "title": "N Chans",
         "type": "integer"
      },
      "n_samples": {
         "title": "N Samples",
         "type": "integer"
      },
      "first_time": {
         "title": "First Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "last_time": {
         "title": "Last Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "system": {
         "title": "System",
         "default": "",
         "type": "string"
      },
      "serial": {
         "title": "Serial",
         "default": "",
         "type": "string"
      },
      "wgs84_latitude": {
```


(continues on next page)

(continued from previous page)

```
            "title": "Wgs84 Latitude",
            "default": -999.0,
            "type": "number"
         },
         "wgs84_longitude": {
            "title": "Wgs84 Longitude",
            "default": -999.0,
            "type": "number"
         },
         "easting": {
            "title": "Easting",
            "default": -999.0,
            "type": "number"
         },
         "northing": {
            "title": "Northing",
            "default": -999.0,
            "type": "number"
         },
         "elevation": {
            "title": "Elevation",
            "default": -999.0,
            "type": "number"
         },
         "chans_metadata": {
            "title": "Chans Metadata",
            "type": "object",
            "additionalProperties": {
               "$ref": "#/definitions/ChanMetadata"
            }
         },
         "history": {
            "title": "History",
            "default": {
               "records": []
            },
            "allOf": [
               {
                  "$ref": "#/definitions/History"
               }
            ]
         }
      },
      "required": [
         "fs",
         "chans",
         "n_samples",
         "first_time",
         "last_time",
         "chans_metadata"
      ],
      "definitions": {
```

(continues on next page)

(continued from previous page)

```
    "ResisticsFile": {
       "title": "ResisticsFile",
       "description": "Required information for writing out a resistics file",
       "type": "object",
       "properties": {
          "created_on_local": {
             "title": "Created On Local",
             "type": "string",
             "format": "date-time"
          },
          "created_on_utc": {
             "title": "Created On Utc",
             "type": "string",
             "format": "date-time"
          },
          "version": {
             "title": "Version",
             "type": "string"
          }
       }
    },
    "ChanMetadata": {
       "title": "ChanMetadata",
       "description": "Channel metadata",
       "type": "object",
       "properties": {
          "name": {
             "title": "Name",
             "type": "string"
          },
          "data_files": {
             "title": "Data Files",
             "type": "array",
             "items": {
                "type": "string"
             }
          },
          "chan_type": {
             "title": "Chan Type",
             "type": "string"
          },
          "chan_source": {
             "title": "Chan Source",
             "type": "string"
          },
          "sensor": {
             "title": "Sensor",
             "default": "",
             "type": "string"
          },
          "serial": {
             "title": "Serial",
```

(continues on next page)

(continued from previous page)


```
                "default": "",
                "type": "string"
            },
            "gain1": {
                "title": "Gain1",
                "default": 1,
                "type": "number"
            },
            "gain2": {
                "title": "Gain2",
                "default": 1,
                "type": "number"
            },
            "scaling": {
                "title": "Scaling",
                "default": 1,
                "type": "number"
            },
            "chopper": {
                "title": "Chopper",
                "default": false,
                "type": "boolean"
            },
            "dipole_dist": {
                "title": "Dipole Dist",
                "default": 1,
                "type": "number"
            },
            "sensor_calibration_file": {
                "title": "Sensor Calibration File",
                "default": "",
                "type": "string"
            },
            "instrument_calibration_file": {
                "title": "Instrument Calibration File",
                "default": "",
                "type": "string"
            }
        },
        "required": [
            "name"
        ]
    },
    "Record": {
        "title": "Record",
        "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...    ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
```


(continues on next page)

(continued from previous page)

```
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
        "type": "object",
        "properties": {
           "time_local": {
              "title": "Time Local",
              "type": "string",
              "format": "date-time"
           },
           "time_utc": {
              "title": "Time Utc",
              "type": "string",
              "format": "date-time"
           },
           "creator": {
              "title": "Creator",
              "type": "object"
           },
           "messages": {
              "title": "Messages",
              "type": "array",
              "items": {
                 "type": "string"
              }
           },
           "record_type": {
              "title": "Record Type",
              "type": "string"
           }
        },
        "required": [
           "creator",
           "messages",
           "record_type"
        ]
     },
     "History": {
        "title": "History",
        "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n    ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n    ␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n    ␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
```

(continues on next page)

(continued from previous page)

```
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
        "type": "object",
        "properties": {
           "records": {
              "title": "Records",
              "default": [],
              "type": "array",
              "items": {
                 "$ref": "#/definitions/Record"
              }
           }
        }
     }
  }
}
```

field fs: *float* **[Required]**

The sampling frequency

field chans: *List[str]* **[Required]**

List of channels

field n_chans: *int | None* **= None**

The number of channels

Validated by

- validate_n_chans

field n_samples: *int* **[Required]**

The number of samples

field first_time: *HighResDateTime* **[Required]**

The datetime of the first sample

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field last_time: *HighResDateTime* **[Required]**

The datetime of the last sample

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field system: *str* **= ''**

The system used for recording

field serial: *str* **= ''**

Serial number of the system

field wgs84_latitude: *float* **= -999.0**

Latitude in WGS84

field wgs84_longitude: float = -999.0

Longitude in WGS84

field easting: float = -999.0

The easting of the site in local cartersian coordinates

field northing: float = -999.0

The northing of the site in local cartersian coordinates

field elevation: float = -999.0

The elevation of the site

field chans_metadata: Dict[str, *ChanMetadata*] [Required]

List of channel metadata

field history: *History* = History(records=[])

Processing history

property dt: float

Get the sampling frequency

property duration: attotimedelta

Get the duration of the recording

property nyquist: float

Get the nyquist frequency

get_chan_types() → List[str]

Get all the different channel types

Returns

A list of different channel types

Return type

List[str]

Examples

```
>>> from resistics.testing import time_metadata_mt
>>> metadata = time_metadata_mt()
>>> metadata.get_chan_types()
['electric', 'magnetic']
```

get_chans_with_type(*chan_type: str*) → List[str]

Get channels with the given type

Parameters

chan_type (*str*) – The channel type

Returns

A list of channels with the given channel type

Return type

List[str]

Examples

```
>>> from resistics.testing import time_metadata_mt
>>> metadata = time_metadata_mt()
>>> metadata.get_chans_with_type("magnetic")
['Hx', 'Hy']
```

get_electric_chans() → List[str]

Get list of electric channels

Returns
List of electric channels

Return type
List[str]

Examples

```
>>> from resistics.testing import time_metadata_mt
>>> metadata = time_metadata_mt()
>>> metadata.get_electric_chans()
['Ex', 'Ey']
```

get_magnetic_chans() → List[str]

Get list of magnetic channels

Returns
List of magnetic channels

Return type
List[str]

Examples

```
>>> from resistics.testing import time_metadata_mt
>>> metadata = time_metadata_mt()
>>> metadata.get_magnetic_chans()
['Hx', 'Hy']
```

any_electric() → bool

True if any channels are electric

any_magnetic() → bool

True if any channels are magnetic

resistics.time.**get_time_metadata**(*time_dict: Dict[str, Any]*, *chans_dict: Dict[str, Dict[str, Any]]*) → *TimeMetadata*

Get metadata for TimeData

The time and channel dictionaries must have the TimeMetadata required fields. For more information about the required fields, see *TimeMetadata*

Parameters

- **time_dict** (*Dict[str, Any]*) – Dictionary with metadata for the whole dataset

- **chans_dict** (*Dict[str, Dict[str, Any]]*) – Dictionary of dictionaries with metadata for each channel

Returns
Metadata for TimeData

Return type
TimeMetadata

See also:

TimeMetadata
The TimeMetadata class which is returned

Examples

```
>>> from resistics.time import get_time_metadata
>>> time_dict = {
...     "fs": 10,
...     "n_samples": 100,
...     "chans": ["Ex", "Hy"],
...     "n_chans": 2,
...     "first_time": "2021-01-01 00:00:00",
...     "last_time": "2021-01-01 00:01:00"
... }
>>> chans_dict = {
...     "Ex": {"name": "Ex", "data_files": "example.ascii"},
...     "Hy": {"name": "Hy", "data_files": "example2.ascii", "sensor": "MFS"}
... }
>>> metadata = get_time_metadata(time_dict, chans_dict)
>>> metadata.summary()
{
    'file_info': None,
    'fs': 10.0,
    'chans': ['Ex', 'Hy'],
    'n_chans': 2,
    'n_samples': 100,
    'first_time': '2021-01-01 00:00:00.000000_000000_000000_000000',
    'last_time': '2021-01-01 00:01:00.000000_000000_000000_000000',
    'system': '',
    'serial': '',
    'wgs84_latitude': -999.0,
    'wgs84_longitude': -999.0,
    'easting': -999.0,
    'northing': -999.0,
    'elevation': -999.0,
    'chans_metadata': {
        'Ex': {
            'name': 'Ex',
            'data_files': ['example.ascii'],
            'chan_type': 'electric',
            'chan_source': None,
            'sensor': '',
            'serial': '',
```

(continues on next page)

(continued from previous page)

```
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        },
        'Hy': {
            'name': 'Hy',
            'data_files': ['example2.ascii'],
            'chan_type': 'magnetic',
            'chan_source': None,
            'sensor': 'MFS',
            'serial': '',
            'gain1': 1,
            'gain2': 1,
            'scaling': 1,
            'chopper': False,
            'dipole_dist': 1,
            'sensor_calibration_file': '',
            'instrument_calibration_file': ''
        }
    },
    'history': {'records': []}
}
```

resistics.time.**adjust_time_metadata**(*metadata:* TimeMetadata, *fs: float*, *first_time: attodatetime*, *n_samples: int*) → *TimeMetadata*

Adjust time data metadata

This is required if changes have been made to the sampling frequency, the time of the first sample of the number of samples. This might occur in processes such as resampling or decimating.

Warning: The metadata passed in will be changed in place. If the original metadata should be retained, pass through a deepcopy

Parameters

- **metadata** (TimeMetadata) – Metadata to adjust
- **fs** (*float*) – The sampling frequency
- **first_time** (*RSDateTime*) – The first time of the data
- **n_samples** (*int*) – The number of samples

Returns
Adjusted metadata

Return type
TimeMetadata

Examples

```
>>> from resistics.sampling import to_datetime
>>> from resistics.time import adjust_time_metadata
>>> from resistics.testing import time_metadata_2chan
>>> metadata = time_metadata_2chan(fs=10, first_time="2021-01-01 00:00:00", n_
↪samples=101)
>>> metadata.fs
10.0
>>> metadata.n_samples
101
>>> metadata.first_time
attotime.objects.attodatetime(2021, 1, 1, 0, 0, 0, 0, 0)
>>> metadata.last_time
attotime.objects.attodatetime(2021, 1, 1, 0, 0, 10, 0, 0)
>>> metadata = adjust_time_metadata(metadata, 20, to_datetime("2021-03-01 00:01:00
↪"), 50)
>>> metadata.fs
20.0
>>> metadata.n_samples
50
>>> metadata.first_time
attotime.objects.attodatetime(2021, 3, 1, 0, 1, 0, 0, 0)
>>> metadata.last_time
attotime.objects.attodatetime(2021, 3, 1, 0, 1, 2, 450000, 0)
```

class resistics.time.**TimeData**(*metadata:* TimeMetadata, *data:* *ndarray*)

Bases: *ResisticsData*

Class for holding time data

The data values are stored in an numpy array attribute named data. This has shape:

n_chans x n_samples

Parameters

- **metadata** (TimeMetadata) – Metadata for the TimeData
- **data** (*np.ndarray*) – Numpy array of the data

Examples

```
>>> import numpy as np
>>> from resistics.testing import time_metadata_2chan
>>> from resistics.time import TimeData
>>> data = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], [11, 10, 9, 8, 7, 6, 5, 4, 3, 2,␣
↪1]]
>>> time_data = TimeData(time_metadata_2chan(), np.array(data))
>>> time_data.metadata.chans
['chan1', 'chan2']
>>> time_data.get_chan("chan1")
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> time_data["chan1"]
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

get_chan_index(*chan: str*) → int

Get the channel index in the data

Parameters
chan (*str*) – The channel

Returns
The index

Return type
int

get_chan(*chan: str*) → ndarray

Get the time data for a channel

Parameters
chan (*str*) – The channel for which to get the time data

Returns
pandas Series with channel data and datetime index

Return type
np.ndarray

set_chan(*chan: str*, *chan_data: ndarray*) → None

Set channel time data

Parameters

- **chan** (*str*) – The channel to set the data for
- **chan_data** (*np.ndarray*) – The new channel data

Raises

- **ValueError** – If the data has incorrect size
- **ValueError** – If the data has incorrect dtype

get_timestamps(*samples: ndarray | None = None*, *estimate: bool = True*) → ndarray | DatetimeIndex

Get an array of timestamps

Parameters

- **samples** (*Optional[np.ndarray], optional*) – If provided, timestamps are only returned for the specified samples, by default None
- **estimate** (*bool, optional*) – Flag for using estimates instead of high precision datetimes, by default True

Returns
The return dates. This will be a numpy array of RSDateTime objects if estimate is False, else it will be a pandas DatetimeIndex

Return type
Union[np.ndarray, pd.DatetimeIndex]

subsection(*from_time: str | Timestamp | datetime*, *to_time: str | Timestamp | datetime*) → *TimeData*

Get a subsection of the TimeData

Returns a new TimeData object

Parameters

- **from_time** (*DateTimeLike*) – Start of subsection

- **to_time** (*DateTimeLike*) – End of subsection

Returns
Subsection as new TimeData

Return type
TimeData

subsamples(*from_sample: int | None = None*, *to_sample: int | None = None*) → *TimeData*

Get a subsample range of the TimeData

Returns a new TimeData object

Parameters

- **from_sample** (*Optional[int], optional*) – The sample to go from, by default None. If not provided, this will be the first sample.
- **to_sample** (*Optional[int], optional*) – The sample to end at, by default None. If not provided, this will be the last sample

Returns
The subsamples as a new TimeData object

Return type
TimeData

copy() → *TimeData*

Get a deepcopy of the time data object

plot(*fig: Figure | None = None*, *chans: List[str] | None = None*, *color: str = 'blue'*, *legend: str = 'TimeData'*, *max_pts: int | None = 10000*) → Figure

Plot time series data

Parameters

- **fig** (*Optional[go.Figure], optional*) – A figure if appending the data to an existing plot, by default None
- **chans** (*Optional[List[str]], optional*) – Explicit definition of channels to plot, by default None
- **color** (*str, optional*) – The color for the data, by default "blue"
- **legend** (*str, optional*) – The legend group to use, by default "TimeData". This is more useful when plotting multiple TimeData
- **max_pts** (*Optional[int], optional*) – The maximum number of points for any channel plot before applying lttbc downsampling, by default 10_000. If set to None, no downsampling will be applied.

Returns
Plotly Figure

Return type
go.Figure

Raises
ValueError – If a figure is provided and channels have not been explicitly defined

to_string() → str

Class details as a string

pydantic model resistics.time.**TimeReader**

Bases: *ResisticsProcess*


```
{
   "title": "TimeReader",
   "description": "Base class for resistics processes\n\nResistics processes␣
↪perform operations on data (including read and write\noperations). Each time a␣
↪ResisticsProcess child class is run, it should add\na process record to the␣
↪dataset",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "apply_scalings": {
         "title": "Apply Scalings",
         "default": true,
         "type": "boolean"
      },
      "extension": {
         "title": "Extension",
         "type": "string"
      }
   }
}
```


field apply_scalings: *bool* **= True**

field extension: *str* | *None* **= None**

run(*dir_path: Path*, *metadata_only: bool* | *None = False*, *metadata:* TimeMetadata | *None = None*, *from_time: str* | *Timestamp* | *datetime* | *None = None*, *to_time: str* | *Timestamp* | *datetime* | *None = None*, *from_sample: int* | *None = None*, *to_sample: int* | *None = None*) → *TimeMetadata* | *TimeData*

Read time series data

Parameters

- **dir_path** (*Path*) – The directory path
- **metadata_only** (*Optional[bool], optional*) – Read only the metadata, by default False
- **metadata** (*Optional[TimeMetadata], optional*) – Pass the metadata if its already been read in, by default None.
- **from_time** (*Union[DateTimeLike, None], optional*) – Timestamp to read from, by default None
- **to_time** (*Union[DateTimeLike, None], optional*) – Timestamp to read to, by default None
- **from_sample** (*Union[int, None], optional*) – Sample to read from, by default None
- **to_sample** (*Union[int, None], optional*) – Sample to read to, by default None

Returns

A TimeData instance

Return type
TimeData

read_metadata(*dir_path: Path*) → *TimeMetadata*

Read time series data metadata

Parameters
dir_path (*Path*) – The directory path of the time series data

Raises
NotImplementedError – To be implemented in child classes

read_data(*dir_path: Path*, *metadata:* TimeMetadata, *read_from: int*, *read_to: int*) → *TimeData*

Read raw data with minimal scalings applied

Parameters

- **dir_path** (*path*) – The directory path to read from
- **metadata** (TimeMetadata) – Time series data metadata
- **read_from** (*int*) – Sample to read data from
- **read_to** (*int*) – Sample to read data to

Raises
NotImplementedError – To be implemented in child TimeReader classes

scale_data(*time_data:* TimeData) → *TimeData*

Scale data to physically meaningful units.

For magnetotelluric data, this is assumed to be mV/km for electric channels, mV for magnetic channels (or nT for certain sensors)

The base class assumes the data is already in the correct units and requires no scaling.

Parameters
time_data (TimeData) – TimeData read in from file

Returns
TimeData scaled to give physically meaningful units

Return type
TimeData

pydantic model resistics.time.**TimeReaderJSON**

Bases: *TimeReader*

Base class for TimeReaders that use a resistics JSON header


```
{
   "title": "TimeReaderJSON",
   "description": "Base class for TimeReaders that use a resistics JSON header",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "apply_scalings": {
         "title": "Apply Scalings",
```


(continues on next page)

(continued from previous page)


```
            "default": true,
            "type": "boolean"
        },
        "extension": {
            "title": "Extension",
            "type": "string"
        }
    }
}
```


read_metadata(*dir_path: Path*) → *TimeMetadata*

Read the time series data metadata and return

Parameters
dir_path (*Path*) – Path to time series data directory

Returns
Metadata for time series data

Return type
TimeMetadata

Raises

- ***MetadataReadError*** – If the headers cannot be parsed
- ***TimeDataReadError*** – If the data files do not match the expected extension

pydantic model resistics.time.**TimeReaderAscii**

Bases: *TimeReaderJSON*

Class for reading Ascii data

Ascii data expected to be a single file with all the data. The delimiter can be set using the delimiter class attribute as can the number of header lines with the n_header attribute.


```
{
   "title": "TimeReaderAscii",
   "description": "Class for reading Ascii data\n\nAscii data expected to be a␣
↪single file with all the data. The delimiter can\nbe set using the delimiter␣
↪class attribute as can the number of header\nlines with the n_header attribute.",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "apply_scalings": {
         "title": "Apply Scalings",
         "default": true,
         "type": "boolean"
      },
      "extension": {
         "title": "Extension",
         "default": ".txt",
         "type": "string"
```


(continues on next page)

(continued from previous page)

```
      },
      "delimiter": {
         "title": "Delimiter",
         "type": "string"
      },
      "n_header": {
         "title": "N Header",
         "default": 0,
         "type": "integer"
      }
   }
}
```

field extension: str = '.txt'

field delimiter: str | None = None

field n_header: int = 0

read_data(*dir_path: Path*, *metadata:* TimeMetadata, *read_from: int*, *read_to: int*) → *TimeData*

Read data from Ascii files

Parameters

- **dir_path** (*path*) – The directory path to read from
- **metadata** (TimeMetadata) – Time series data metadata
- **read_from** (*int*) – Sample to read data from
- **read_to** (*int*) – Sample to read data to

Returns

TimeData

Return type

TimeData

Raises

ValueError – If metadata is None

pydantic model resistics.time.**TimeReaderNumpy**

Bases: *TimeReaderJSON*

Class for reading Numpy data

This is expected to be a single data file for all channels. The ordering is assumed to be the same as the channels definition in the metadata.

```
{
   "title": "TimeReaderNumpy",
   "description": "Class for reading Numpy data\n\nThis is expected to be a single␣
↪data file for all channels. The ordering is\nassumed to be the same as the␣
↪channels definition in the metadata.",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
```

(continues on next page)

(continued from previous page)


```
            "type": "string"
        },
        "apply_scalings": {
            "title": "Apply Scalings",
            "default": true,
            "type": "boolean"
        },
        "extension": {
            "title": "Extension",
            "default": ".npy",
            "type": "string"
        }
    }
}
```


field extension: str = '.npy'

read_data(*dir_path: Path*, *metadata:* TimeMetadata, *read_from: int*, *read_to: int*) → *TimeData*

Read raw data saved in numpy data

Parameters

- **dir_path** (*path*) – The directory path to read from
- **metadata** (TimeMetadata) – Time series data metadata
- **read_from** (*int*) – Sample to read data from
- **read_to** (*int*) – Sample to read data to

Returns
TimeData

Return type
TimeData

Raises
ValueError – If metadata is None

pydantic model resistics.time.**TimeWriterNumpy**

Bases: *ResisticsWriter*

Write out time data in numpy binary format

Data is written out as a single data file including all channels


```
{
   "title": "TimeWriterNumpy",
   "description": "Write out time data in numpy binary format\n\nData is written␣
↪out as a single data file including all channels",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "overwrite": {
```


(continues on next page)

(continued from previous page)

```
            "title": "Overwrite",
            "default": true,
            "type": "boolean"
        }
    }
}
```

run(*dir_path: Path*, *time_data:* TimeData) → None

Write out TimeData

Parameters

- **dir_path** (*Path*) – The directory path to write to
- **time_data** (TimeData) – TimeData to write out

Raises

WriteError – If unable to write to the directory

pydantic model resistics.time.**TimeWriterAscii**

Bases: *ResisticsWriter*

Write out time data in ascii format

```
{
   "title": "TimeWriterAscii",
   "description": "Write out time data in ascii format",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "overwrite": {
         "title": "Overwrite",
         "default": true,
         "type": "boolean"
      }
   }
}
```

run(*dir_path: Path*, *time_data:* TimeData) → None

Write out TimeData

Parameters

- **dir_path** (*Path*) – The directory path to write to
- **time_data** (TimeData) – TimeData to write out

Raises

WriteError – If unable to write to the directory

resistics.time.**new_time_data**(*time_data:* TimeData, *metadata:* TimeMetadata | *None* = *None*, *data: ndarray* | *None* = *None*, *record:* Record | *None* = *None*) → *TimeData*

Get a new TimeData

Values are taken from an existing TimeData where they are not explicitly specified. This is useful in a process where only some aspects of the TimeData have been changed

Parameters

- **time_data** (TimeData) – The existing TimeData
- **metadata** (*Optional[TimeMetadata], optional*) – A new TimeMetadata, by default None
- **data** (*Optional[np.ndarray], optional*) – New data, by default None
- **record** (*Optional[Record], optional*) – A new record to add, by default None

Returns

A new TimeData instance

Return type

TimeData

pydantic model resistics.time.**TimeProcess**

Bases: *ResisticsProcess*

Parent class for processing time data


```
{
   "title": "TimeProcess",
   "description": "Parent class for processing time data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*time_data:* TimeData) → *TimeData*

Run the time processor

pydantic model resistics.time.**Subsection**

Bases: *TimeProcess*

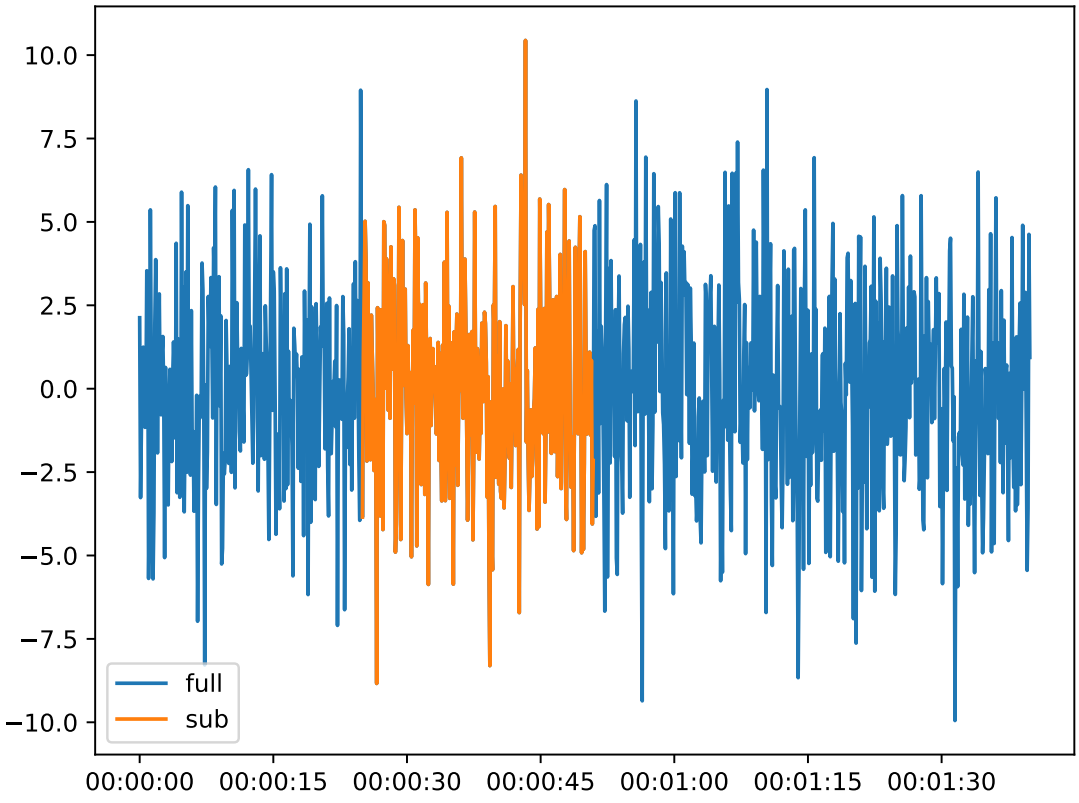
Get a subsection of time data

Parameters

- **from_time** (*DateTimeLike*) – Time to take subsection from
- **to_time** (*DateTimeLike*) – Time to take subsection to

Examples

```
>>> import matplotlib.pyplot as plt
>>> from resistics.testing import time_data_random
>>> from resistics.time import Subsection
>>> time_data = time_data_random(n_samples=1000)
>>> print(time_data.metadata.first_time, time_data.metadata.last_time)
2020-01-01 00:00:00 2020-01-01 00:01:39.9
>>> process = Subsection(from_time="2020-01-01 00:00:25", to_time="2020-01-01␣
↪00:00:50.9")
>>> subsection = process.run(time_data)
>>> print(subsection.metadata.first_time, subsection.metadata.last_time)
2020-01-01 00:00:25 2020-01-01 00:00:50.9
>>> subsection.metadata.n_samples
260
>>> plt.plot(time_data.get_timestamps(), time_data["Ex"], label="full")
>>> plt.plot(subsection.get_timestamps(), subsection["Ex"], label="sub")
>>> plt.legend(loc=3)
>>> plt.tight_layout()
>>> plt.show()
```



See also:

Subsamples
Getting a section of data using samples rather than dates

```
{
   "title": "Subsection",
   "description": "Get a subsection of time data\n\nParameters\n----------\nfrom_
↪time : DateTimeLike\n    Time to take subsection from\nto_time : DateTimeLike\n  ␣
↪ Time to take subsection to\n\nExamples\n--------\n.. plot::\n    :width: 90%\n\n␣
↪   >>> import matplotlib.pyplot as plt\n    >>> from resistics.testing import␣
↪time_data_random\n    >>> from resistics.time import Subsection\n    >>> time_
↪data = time_data_random(n_samples=1000)\n    >>> print(time_data.metadata.first_
↪time, time_data.metadata.last_time)\n    2020-01-01 00:00:00 2020-01-01 00:01:39.
↪9\n    >>> process = Subsection(from_time=\"2020-01-01 00:00:25\", to_time=\"2020-
↪01-01 00:00:50.9\")\n    >>> subsection = process.run(time_data)\n    >>>␣
↪print(subsection.metadata.first_time, subsection.metadata.last_time)\n    2020-01-
↪01 00:00:25 2020-01-01 00:00:50.9\n    >>> subsection.metadata.n_samples\n    260\
↪n    >>> plt.plot(time_data.get_timestamps(), time_data[\"Ex\"], label=\"full\")
↪# doctest: +SKIP\n    >>> plt.plot(subsection.get_timestamps(), subsection[\"Ex\
↪"], label=\"sub\") # doctest: +SKIP\n    >>> plt.legend(loc=3) # doctest: +SKIP\n␣
↪   >>> plt.tight_layout() # doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP\n\
↪nSee Also\n--------\nSubsamples : Getting a section of data using samples rather␣
↪than dates",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "from_time": {
         "title": "From Time",
         "anyOf": [
            {
               "type": "string"
            },
            {
               "type": "string",
               "format": "date-time"
            },
            {
               "type": "string",
               "format": "date-time"
            }
         ]
      },
      "to_time": {
         "title": "To Time",
         "anyOf": [
            {
               "type": "string"
            },
            {
               "type": "string",
               "format": "date-time"
            },
            {
```


(continues on next page)

(continued from previous page)

```
                "type": "string",
                "format": "date-time"
            }
         ]
      }
   },
   "required": [
      "from_time",
      "to_time"
   ]
}
```

field from_time: `str | Timestamp | datetime` **[Required]**

field to_time: `str | Timestamp | datetime` **[Required]**

run(*time_data:* TimeData) → *TimeData*

Take a subsection from TimeData

Parameters
time_data (TimeData) – TimeData to take subsection from

Returns
Subsection TimeData

Return type
TimeData

pydantic model resistics.time.**Subsamples**

Bases: *TimeProcess*

Get a subsamples of time data, an alternative to getting a subsection by times

Parameters

- **from_sample** (*int*) – Sample to begin from
- **to_time** (*DateTimeLike*) – Sample to end at

Examples

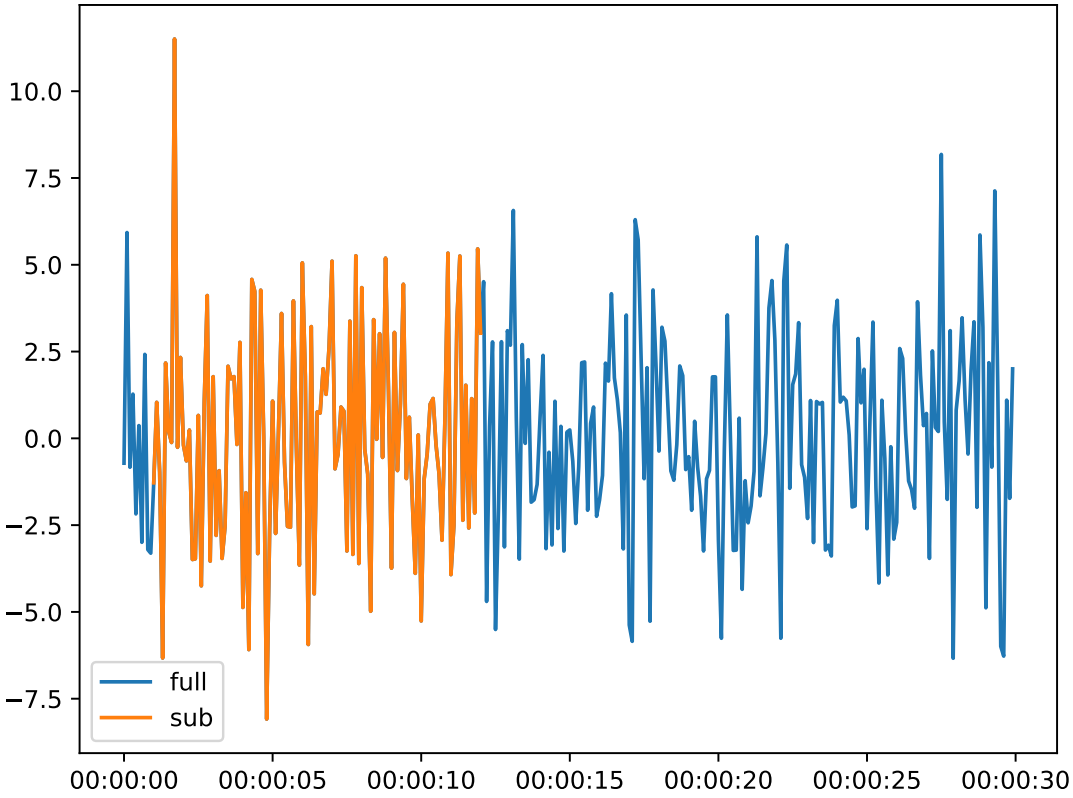
Taking subsample using positive sample numbers. Sample 0 is the first sample and sample -1 is the last sample.

```
>>> import matplotlib.pyplot as plt
>>> from resistics.testing import time_data_random
>>> from resistics.time import Subsamples
>>> time_data = time_data_random(n_samples=300)
>>> print(time_data.metadata.first_time, time_data.metadata.last_time)
2020-01-01 00:00:00 2020-01-01 00:00:29.9
>>> process = Subsamples(from_sample=10, to_sample=120)
>>> subsample = process.run(time_data)
>>> print(subsample.metadata.first_time, subsample.metadata.last_time)
2020-01-01 00:00:01 2020-01-01 00:00:12
>>> subsample.metadata.n_samples
111
```

(continues on next page)

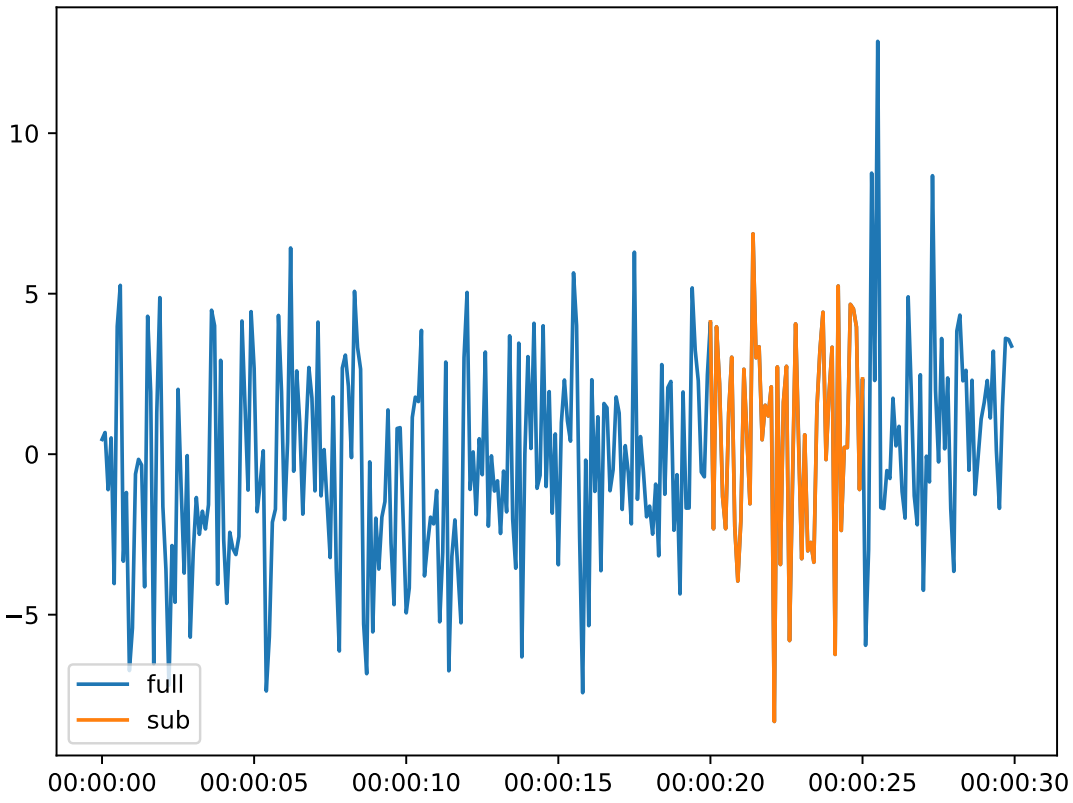
(continued from previous page)

```
>>> plt.plot(time_data.get_timestamps(), time_data["Ex"], label="full")
>>> plt.plot(subsample.get_timestamps(), subsample["Ex"], label="sub")
>>> plt.legend(loc=3)
>>> plt.tight_layout()
>>> plt.show()
```



Another option is to use negative sample numbers which counts back from the end of the time data.

```
>>> import matplotlib.pyplot as plt
>>> from resistics.testing import time_data_random
>>> from resistics.time import Subsamples
>>> time_data = time_data_random(n_samples=300)
>>> print(time_data.metadata.first_time, time_data.metadata.last_time)
2020-01-01 00:00:00 2020-01-01 00:00:29.9
>>> process = Subsamples(from_sample=-100, to_sample=-50)
>>> subsample = process.run(time_data)
>>> print(subsample.metadata.first_time, subsample.metadata.last_time)
2020-01-01 00:00:20 2020-01-01 00:00:25
>>> subsample.metadata.n_samples
51
>>> plt.plot(time_data.get_timestamps(), time_data["Ex"], label="full")
>>> plt.plot(subsample.get_timestamps(), subsample["Ex"], label="sub")
>>> plt.legend(loc=3)
>>> plt.tight_layout()
>>> plt.show()
```

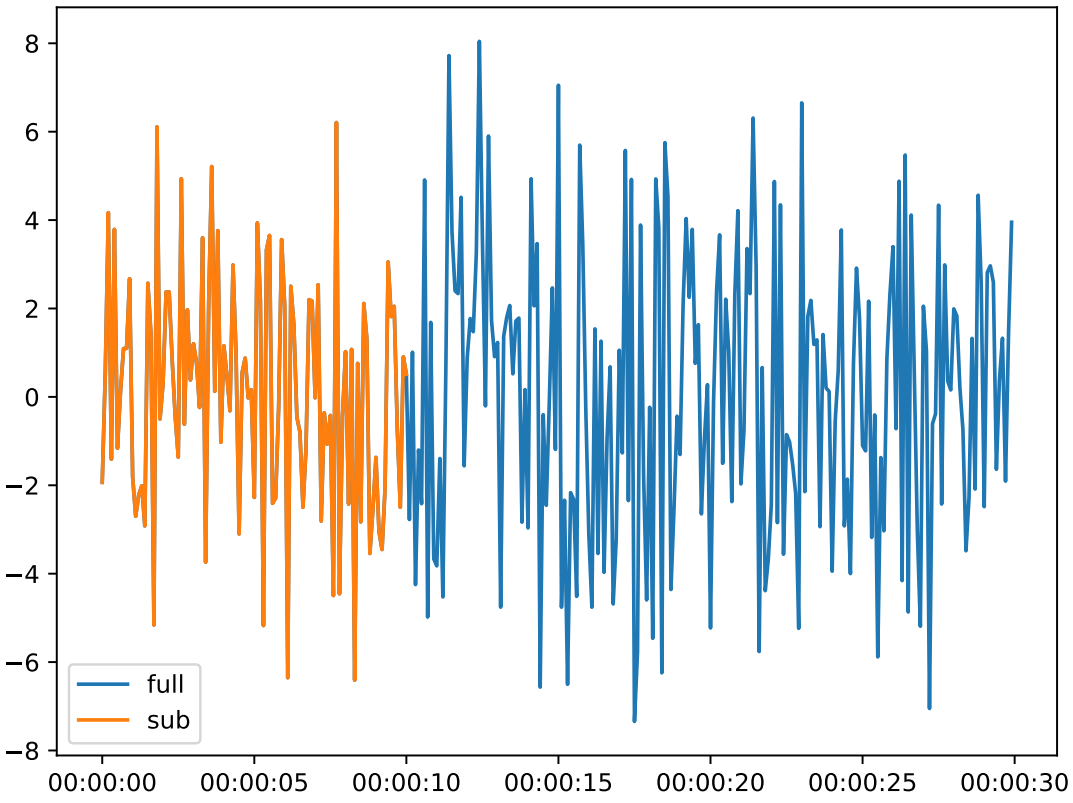


If from_sample is not passed, it will be set to 0. If to_sample is not passed this will default to the last sample.

```
>>> import matplotlib.pyplot as plt
>>> from resistics.testing import time_data_random
>>> from resistics.time import Subsamples
>>> time_data = time_data_random(n_samples=300)
>>> print(time_data.metadata.first_time, time_data.metadata.last_time)
2020-01-01 00:00:00 2020-01-01 00:00:29.9
>>> process = Subsamples(to_sample=100)
>>> subsample = process.run(time_data)
>>> print(subsample.metadata.first_time, subsample.metadata.last_time)
2020-01-01 00:00:00 2020-01-01 00:00:10
>>> subsample.metadata.n_samples
101
>>> plt.plot(time_data.get_timestamps(), time_data["Ex"], label="full")
>>> plt.plot(subsample.get_timestamps(), subsample["Ex"], label="sub")
>>> plt.legend(loc=3)
>>> plt.tight_layout()
>>> plt.show()
```

See also:

Subsection
: For taking a subsection using dates




```
{
   "title": "Subsamples",
   "description": "Get a subsamples of time data, an alternative to getting a␣
↪subsection by\ntimes\n\nParameters\n----------\nfrom_sample : int\n    Sample to␣
↪begin from\nto_time : DateTimeLike\n    Sample to end at\n\nExamples\n--------\
↪nTaking subsample using positive sample numbers. Sample 0 is the first\nsample␣
↪and sample -1 is the last sample.\n\n.. plot::\n    :width: 90%\n\n    >>> import␣
↪matplotlib.pyplot as plt\n    >>> from resistics.testing import time_data_random\
↪n    >>> from resistics.time import Subsamples\n    >>> time_data = time_data_
↪random(n_samples=300)\n    >>> print(time_data.metadata.first_time, time_data.
↪metadata.last_time)\n    2020-01-01 00:00:00 2020-01-01 00:00:29.9\n    >>>␣
↪process = Subsamples(from_sample=10, to_sample=120)\n    >>> subsample = process.
↪run(time_data)\n    >>> print(subsample.metadata.first_time, subsample.metadata.
↪last_time)\n    2020-01-01 00:00:01 2020-01-01 00:00:12\n    >>> subsample.
↪metadata.n_samples\n    111\n    >>> plt.plot(time_data.get_timestamps(), time_
↪data[\"Ex\"], label=\"full\") # doctest: +SKIP\n    >>> plt.plot(subsample.get_
↪timestamps(), subsample[\"Ex\"], label=\"sub\") # doctest: +SKIP\n    >>> plt.
↪legend(loc=3) # doctest: +SKIP\n    >>> plt.tight_layout() # doctest: +SKIP\n    >
↪>> plt.show() # doctest: +SKIP\n\nAnother option is to use negative sample␣
↪numbers which counts back from the\nend of the time data.\n\n.. plot::\n    ␣
↪:width: 90%\n\n    >>> import matplotlib.pyplot as plt\n    >>> from resistics.
↪testing import time_data_random\n    >>> from resistics.time import Subsamples\n ␣
↪  >>> time_data = time_data_random(n_samples=300)\n    >>> print(time_data.
↪metadata.first_time, time_data.metadata.last_time)\n    2020-01-01 00:00:00 2020-
```


(continues on next page)

(continued from previous page)


```
↪01-01 00:00:29.9\n    >>> process = Subsamples(from_sample=-100, to_sample=-50)\n␣
↪   >>> subsample = process.run(time_data)\n    >>> print(subsample.metadata.first_
↪time, subsample.metadata.last_time)\n    2020-01-01 00:00:20 2020-01-01 00:00:25\
↪n    >>> subsample.metadata.n_samples\n    51\n    >>> plt.plot(time_data.get_
↪timestamps(), time_data[\"Ex\"], label=\"full\") # doctest: +SKIP\n    >>> plt.
↪plot(subsample.get_timestamps(), subsample[\"Ex\"], label=\"sub\") # doctest:␣
↪+SKIP\n    >>> plt.legend(loc=3) # doctest: +SKIP\n    >>> plt.tight_layout() #␣
↪doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP\n\nIf from_sample is not␣
↪passed, it will be set to 0. If to_sample is not\npassed this will default to the␣
↪last sample.\n\n.. plot::\n    :width: 90%\n\n    >>> import matplotlib.pyplot as␣
↪plt\n    >>> from resistics.testing import time_data_random\n    >>> from␣
↪resistics.time import Subsamples\n    >>> time_data = time_data_random(n_
↪samples=300)\n    >>> print(time_data.metadata.first_time, time_data.metadata.
↪last_time)\n    2020-01-01 00:00:00 2020-01-01 00:00:29.9\n    >>> process =␣
↪Subsamples(to_sample=100)\n    >>> subsample = process.run(time_data)\n    >>>␣
↪print(subsample.metadata.first_time, subsample.metadata.last_time)\n    2020-01-
↪01 00:00:00 2020-01-01 00:00:10\n    >>> subsample.metadata.n_samples\n    101\n ␣
↪  >>> plt.plot(time_data.get_timestamps(), time_data[\"Ex\"], label=\"full\") #␣
↪doctest: +SKIP\n    >>> plt.plot(subsample.get_timestamps(), subsample[\"Ex\"],␣
↪label=\"sub\") # doctest: +SKIP\n    >>> plt.legend(loc=3) # doctest: +SKIP\n    >
↪>> plt.tight_layout() # doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP\n\
↪nSee Also\n--------\nSubsection : For taking a subsection using dates",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "from_sample": {
         "title": "From Sample",
         "type": "integer"
      },
      "to_sample": {
         "title": "To Sample",
         "type": "integer"
      }
   }
}
```


field from_sample: int | None = None

field to_sample: int | None = None

run(*time_data:* TimeData) → *TimeData*

Take a subsection from TimeData

Parameters
time_data (TimeData) – TimeData to take subsection from

Returns
Subsection TimeData

Return type
TimeData

Raises

- ***ProcessRunError*** – If from_sample is not less than to_sample
- ***ProcessRunError*** – If from_sample is out of range
- ***ProcessRunError*** – If to_sample is out of range

pydantic model resistics.time.**InterpolateNans**

Bases: *TimeProcess*

Interpolate nan values in the data

Preserve the data type of the input time data

Examples

```
>>> from resistics.testing import time_data_with_nans
>>> from resistics.time import InterpolateNans
>>> time_data = time_data_with_nans()
>>> time_data["Hx"]
array([nan,  2.,  3.,  5.,  1.,  2.,  3.,  4.,  2.,  6.,  7., nan, nan,
        4.,  3.,  2.], dtype=float32)
>>> process = InterpolateNans()
>>> time_data_new = process.run(time_data)
>>> time_data_new["Hx"]
array([2., 2., 3., 5., 1., 2., 3., 4., 2., 6., 7., 6., 5., 4., 3., 2.],
      dtype=float32)
```

```
{
   "title": "InterpolateNans",
   "description": "Interpolate nan values in the data\n\nPreserve the data type of␣
↪the input time data\n\nExamples\n--------\n>>> from resistics.testing import time_
↪data_with_nans\n>>> from resistics.time import InterpolateNans\n>>> time_data =␣
↪time_data_with_nans()\n>>> time_data[\"Hx\"]\narray([nan,  2.,  3.,  5.,  1.,  2.,
↪  3.,  4.,  2.,  6.,  7., nan, nan,\n        4.,  3.,  2.], dtype=float32)\n>>>␣
↪process = InterpolateNans()\n>>> time_data_new = process.run(time_data)\n>>> time_
↪data_new[\"Hx\"]\narray([2., 2., 3., 5., 1., 2., 3., 4., 2., 6., 7., 6., 5., 4.,␣
↪3., 2.],\n      dtype=float32)",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```

run(*time_data:* TimeData) → *TimeData*

Interpolate nan values

Parameters

time_data (TimeData) – TimeData to remove nan values from

Returns

TimeData with no nan values

Return type
TimeData

pydantic model resistics.time.**RemoveMean**

Bases: *TimeProcess*

Remove channel mean value from each channel

Preserve the data type of the input time data

Examples

```
>>> import numpy as np
>>> from resistics.testing import time_data_simple
>>> from resistics.time import RemoveMean
>>> time_data = time_data_simple()
>>> process = RemoveMean()
>>> time_data_new = process.run(time_data)
>>> time_data_new["Hx"]
array([-2.5, -1.5, -0.5,  1.5, -2.5, -1.5, -0.5,  0.5, -1.5,  2.5,  3.5,
        2.5,  1.5,  0.5, -0.5, -1.5], dtype=float32)
>>> hx_test = time_data["Hx"] - np.mean(time_data["Hx"])
>>> hx_test
array([-2.5, -1.5, -0.5,  1.5, -2.5, -1.5, -0.5,  0.5, -1.5,  2.5,  3.5,
        2.5,  1.5,  0.5, -0.5, -1.5], dtype=float32)
>>> np.all(hx_test == time_data_new["Hx"])
True
```


```
{
   "title": "RemoveMean",
   "description": "Remove channel mean value from each channel\n\nPreserve the data␣
↪type of the input time data\n\nExamples\n--------\n>>> import numpy as np\n>>>␣
↪from resistics.testing import time_data_simple\n>>> from resistics.time import␣
↪RemoveMean\n>>> time_data = time_data_simple()\n>>> process = RemoveMean()\n>>>␣
↪time_data_new = process.run(time_data)\n>>> time_data_new[\"Hx\"]\narray([-2.5, -
↪1.5, -0.5,  1.5, -2.5, -1.5, -0.5,  0.5, -1.5,  2.5,  3.5,\n        2.5,  1.5,  0.
↪5, -0.5, -1.5], dtype=float32)\n>>> hx_test = time_data[\"Hx\"] - np.mean(time_
↪data[\"Hx\"])\n>>> hx_test\narray([-2.5, -1.5, -0.5,  1.5, -2.5, -1.5, -0.5,  0.5,
↪ -1.5,  2.5,  3.5,\n        2.5,  1.5,  0.5, -0.5, -1.5], dtype=float32)\n>>> np.
↪all(hx_test == time_data_new[\"Hx\"])\nTrue",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


run(*time_data:* TimeData) → *TimeData*

Remove mean from TimeData

Parameters
time_data (TimeData) – TimeData input

Returns
TimeData with mean removed

Return type
TimeData

pydantic model resistics.time.**Add**

Bases: *TimeProcess*

Add values to channels

Add can be used to add a constant value to all channels or values for specific channels can be provided.

Add preserves the data type of the original data

Parameters
add (*Union[float, Dict[str, float]]*) – Either a scalar to add to all channels or dictionary with values to add to each channel

Examples

Using a constant value for all channels passed as a scalar

```
>>> from resistics.testing import time_data_ones
>>> from resistics.time import Add
>>> time_data = time_data_ones()
>>> process = Add(add=5)
>>> time_data_new = process.run(time_data)
>>> time_data_new["Ex"] - time_data["Ex"]
array([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.], dtype=float32)
>>> time_data_new["Ey"] - time_data["Ey"]
array([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.], dtype=float32)
```

Variable values for the channels provided as a dictionary

```
>>> time_data = time_data_ones()
>>> process = Add(add={"Ex": 3, "Hy": -7})
>>> time_data_new = process.run(time_data)
>>> time_data_new["Ex"] - time_data["Ex"]
array([3., 3., 3., 3., 3., 3., 3., 3., 3., 3.], dtype=float32)
>>> time_data_new["Hy"] - time_data["Hy"]
array([-7., -7., -7., -7., -7., -7., -7., -7., -7., -7.], dtype=float32)
>>> time_data_new["Ey"] - time_data["Ey"]
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```


```
{
   "title": "Add",
   "description": "Add values to channels\n\nAdd can be used to add a constant␣
↪value to all channels or values for\nspecific channels can be provided.\n\nAdd␣
↪preserves the data type of the original data\n\nParameters\n----------\nadd :␣
↪Union[float, Dict[str, float]]\n    Either a scalar to add to all channels or␣
↪dictionary with values to\n    add to each channel\n\nExamples\n--------\nUsing a␣
↪constant value for all channels passed as a scalar\n\n>>> from resistics.testing␣
↪import time_data_ones\n>>> from resistics.time import Add\n>>> time_data = time_
↪data_ones()\n>>> process = Add(add=5)\n>>> time_data_new = process.run(time_data)\
```


(continues on next page)

(continued from previous page)

```
↪n>>> time_data_new[\"Ex\"] - time_data[\"Ex\"]\narray([5., 5., 5., 5., 5., 5., 5.,
↪ 5., 5., 5.], dtype=float32)\n>>> time_data_new[\"Ey\"] - time_data[\"Ey\"]\
↪narray([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.], dtype=float32)\n\nVariable␣
↪values for the channels provided as a dictionary\n\n>>> time_data = time_data_
↪ones()\n>>> process = Add(add={\"Ex\": 3, \"Hy\": -7})\n>>> time_data_new =␣
↪process.run(time_data)\n>>> time_data_new[\"Ex\"] - time_data[\"Ex\"]\narray([3.,␣
↪3., 3., 3., 3., 3., 3., 3., 3., 3.], dtype=float32)\n>>> time_data_new[\"Hy\"] -␣
↪time_data[\"Hy\"]\narray([-7., -7., -7., -7., -7., -7., -7., -7., -7., -7.],␣
↪dtype=float32)\n>>> time_data_new[\"Ey\"] - time_data[\"Ey\"]\narray([0., 0., 0.,␣
↪0., 0., 0., 0., 0., 0., 0.], dtype=float32)",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "add": {
         "title": "Add",
         "anyOf": [
            {
               "type": "number"
            },
            {
               "type": "object",
               "additionalProperties": {
                  "type": "number"
               }
            }
         ]
      }
   },
   "required": [
      "add"
   ]
}
```

field add: `float | Dict[str, float]` **[Required]**

run(*time_data:* TimeData) → *TimeData*

Add values to the data

Parameters
time_data (TimeData) – The input TimeData

Returns
TimeData with values added

Return type
TimeData

pydantic model resistics.time.**Multiply**

Bases: *TimeProcess*

Multiply channels by values

Multiply can be used to add a constant value to all channels or values for specific channels can be provided.

Multiply preseves the original type of the time data

Parameters

multiplier (*Union[Dict[str, float], float]*) – Either a float to multiply all channels with the same value or a dictionary to specify different values for each channel

Examples

Using a constant value for all channels passed as a scalar

```
>>> from resistics.testing import time_data_ones
>>> from resistics.time import Multiply
>>> time_data = time_data_ones()
>>> process = Multiply(multiplier=5)
>>> time_data_new = process.run(time_data)
>>> time_data_new["Ex"]/time_data["Ex"]
array([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.], dtype=float32)
>>> time_data_new["Ey"]/time_data["Ey"]
array([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.], dtype=float32)
```

Variable values for the channels provided as a dictionary

```
>>> time_data = time_data_ones()
>>> process = Multiply(multiplier={"Ex": 3, "Hy": -7})
>>> time_data_new = process.run(time_data)
>>> time_data_new["Ex"]/time_data["Ex"]
array([3., 3., 3., 3., 3., 3., 3., 3., 3., 3.], dtype=float32)
>>> time_data_new["Hy"]/time_data["Hy"]
array([-7., -7., -7., -7., -7., -7., -7., -7., -7., -7.], dtype=float32)
>>> time_data_new["Ey"]/time_data["Ey"]
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32)
```

```
{
   "title": "Multiply",
   "description": "Multiply channels by values\n\nMultiply can be used to add a␣
↪constant value to all channels or values for\nspecific channels can be provided.\
↪n\nMultiply preseves the original type of the time data\n\nParameters\n----------\
↪nmultiplier : Union[Dict[str, float], float]\n    Either a float to multiply all␣
↪channels with the same value or a\n    dictionary to specify different values for␣
↪each channel\n\nExamples\n--------\nUsing a constant value for all channels␣
↪passed as a scalar\n\n>>> from resistics.testing import time_data_ones\n>>> from␣
↪resistics.time import Multiply\n>>> time_data = time_data_ones()\n>>> process =␣
↪Multiply(multiplier=5)\n>>> time_data_new = process.run(time_data)\n>>> time_data_
↪new[\"Ex\"]/time_data[\"Ex\"]\narray([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.],␣
↪dtype=float32)\n>>> time_data_new[\"Ey\"]/time_data[\"Ey\"]\narray([5., 5., 5., 5.
↪, 5., 5., 5., 5., 5., 5.], dtype=float32)\n\nVariable values for the channels␣
↪provided as a dictionary\n\n>>> time_data = time_data_ones()\n>>> process =␣
↪Multiply(multiplier={\"Ex\": 3, \"Hy\": -7})\n>>> time_data_new = process.
↪run(time_data)\n>>> time_data_new[\"Ex\"]/time_data[\"Ex\"]\narray([3., 3., 3., 3.
↪, 3., 3., 3., 3., 3., 3.], dtype=float32)\n>>> time_data_new[\"Hy\"]/time_data[\
↪"Hy\"]\narray([-7., -7., -7., -7., -7., -7., -7., -7., -7., -7.], dtype=float32)\
↪n>>> time_data_new[\"Ey\"]/time_data[\"Ey\"]\narray([1., 1., 1., 1., 1., 1., 1.,␣
↪1., 1., 1.], dtype=float32)",
```

(continues on next page)

(continued from previous page)

```
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "multiplier": {
         "title": "Multiplier",
         "anyOf": [
            {
               "type": "number"
            },
            {
               "type": "object",
               "additionalProperties": {
                  "type": "number"
               }
            }
         ]
      }
   },
   "required": [
      "multiplier"
   ]
}
```

field multiplier: float | Dict[str, float] [Required]

run(*time_data:* TimeData) → *TimeData*

Multiply the channels

Parameters
time_data (TimeData) – Input TimeData

Returns
TimeData with channels multiplied by the specified numbers

Return type
TimeData

pydantic model resistics.time.**LowPass**

Bases: *TimeProcess*

Apply low pass filter

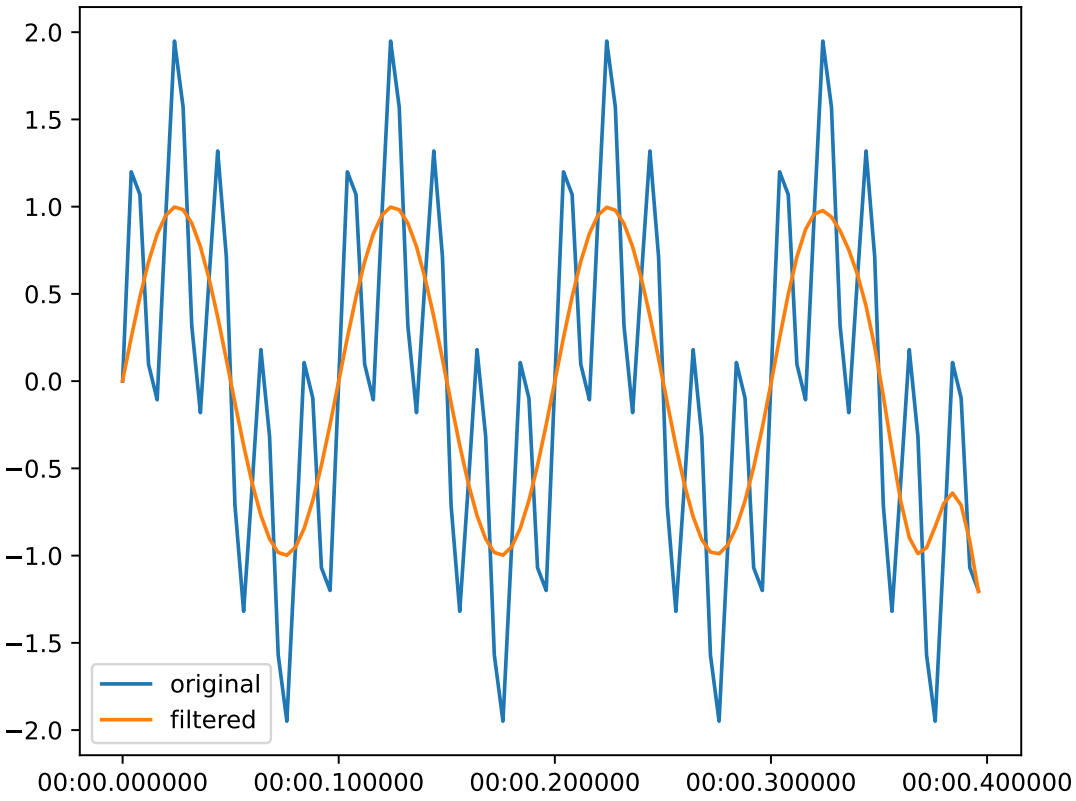
Parameters

- **cutoff** (*float*) – The cutoff for the low pass
- **order** (*int, optional*) – Order of the filter, by default 10

Examples

Low pass to remove 20 Hz from a time series sampled at 50 Hz

```
import matplotlib.pyplot as plt
from resistics.testing import time_data_periodic
from resistics.time import LowPass
time_data = time_data_periodic([10, 50], fs=250, n_samples=100)
process = LowPass(cutoff=30)
filtered = process.run(time_data)
plt.plot(time_data.get_timestamps(), time_data["chan1"], label="original")
plt.plot(filtered.get_timestamps(), filtered["chan1"], label="filtered")
plt.legend(loc=3)
plt.tight_layout()
plt.plot()
```




```
{
   "title": "LowPass",
   "description": "Apply low pass filter\n\nParameters\n----------\ncutoff : float\
↪n    The cutoff for the low pass\norder : int, optional\n    Order of the filter,␣
↪by default 10\n\nExamples\n--------\nLow pass to remove 20 Hz from a time series␣
↪sampled at 50 Hz\n\n.. plot::\n    :width: 90%\n\n    import matplotlib.pyplot as␣
↪plt\n    from resistics.testing import time_data_periodic\n    from resistics.
↪time import LowPass\n    time_data = time_data_periodic([10, 50], fs=250, n_
↪samples=100)\n    process = LowPass(cutoff=30)\n    filtered = process.run(time_
```


(continues on next page)

(continued from previous page)


```
↪data)\n    plt.plot(time_data.get_timestamps(), time_data[\"chan1\"], label=\
↪"original\")\n    plt.plot(filtered.get_timestamps(), filtered[\"chan1\"], label=\
↪"filtered\")\n    plt.legend(loc=3)\n    plt.tight_layout()\n    plt.plot()",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "cutoff": {
         "title": "Cutoff",
         "type": "number"
      },
      "order": {
         "title": "Order",
         "default": 10,
         "type": "integer"
      }
   },
   "required": [
      "cutoff"
   ]
}
```


field cutoff: *float* **[Required]**

field order: *int* **= 10**

run(*time_data:* TimeData) → *TimeData*

Apply the low pass filter

Parameters
time_data (TimeData) – The input TimeData

Returns
The low pass filtered TimeData

Return type
TimeData

Raises
ProcessRunError – If cutoff > nyquist

pydantic model resistics.time.**HighPass**

Bases: *TimeProcess*

High pass filter time data

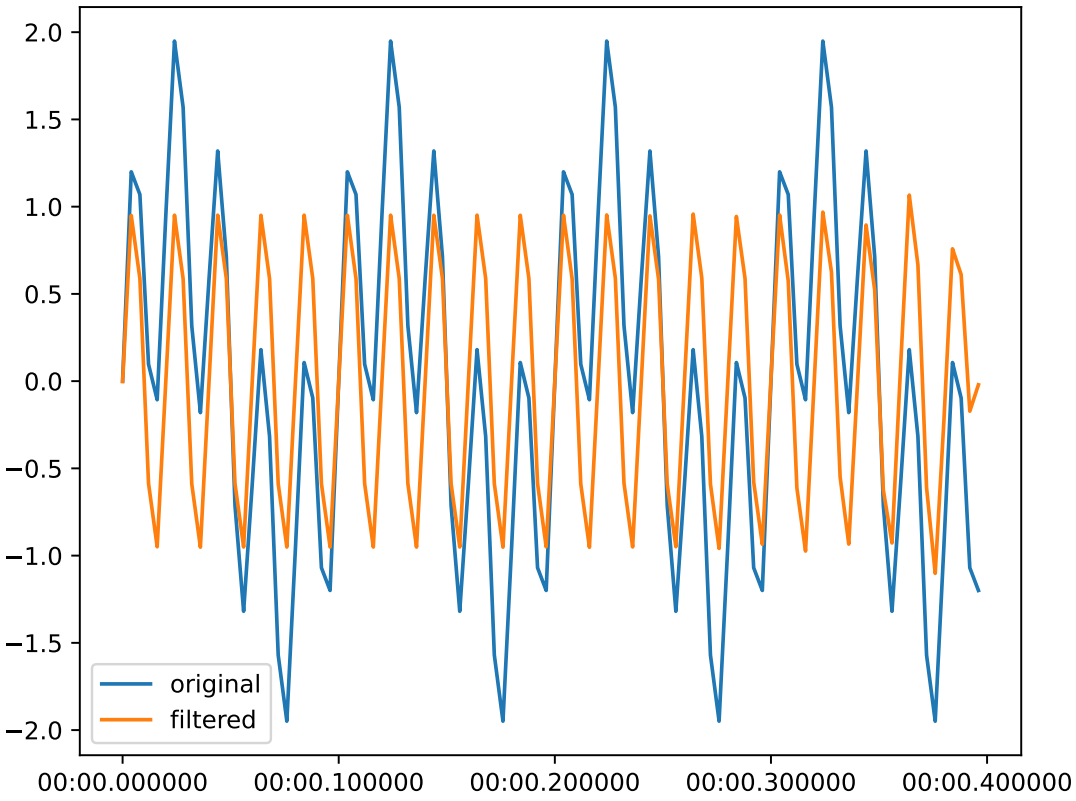
Parameters

- **cutoff** (*float*) – Cutoff for the high pass filter
- **order** (*int, optional*) – Order of the filter, by default 10

Examples

High pass to remove 3 Hz from signal sampled at 50 Hz

```
import matplotlib.pyplot as plt
from resistics.testing import time_data_periodic
from resistics.time import HighPass
time_data = time_data_periodic([10, 50], fs=250, n_samples=100)
process = HighPass(cutoff=30)
filtered = process.run(time_data)
plt.plot(time_data.get_timestamps(), time_data["chan1"], label="original")
plt.plot(filtered.get_timestamps(), filtered["chan1"], label="filtered")
plt.legend(loc=3)
plt.tight_layout()
plt.plot()
```




```
{
   "title": "HighPass",
   "description": "High pass filter time data\n\nParameters\n----------\ncutoff :␣
↪float\n    Cutoff for the high pass filter\norder : int, optional\n    Order of␣
↪the filter, by default 10\n\nExamples\n--------\nHigh pass to remove 3 Hz from␣
↪signal sampled at 50 Hz\n\n.. plot::\n    :width: 90%\n\n    import matplotlib.
↪pyplot as plt\n    from resistics.testing import time_data_periodic\n    from␣
↪resistics.time import HighPass\n    time_data = time_data_periodic([10, 50],␣
↪fs=250, n_samples=100)\n    process = HighPass(cutoff=30)\n    filtered = process.
```


(continues on next page)

(continued from previous page)

```
↪run(time_data)\n    plt.plot(time_data.get_timestamps(), time_data[\"chan1\"],␣
↪label=\"original\")\n    plt.plot(filtered.get_timestamps(), filtered[\"chan1\"],␣
↪label=\"filtered\")\n    plt.legend(loc=3)\n    plt.tight_layout()\n    plt.plot()
↪",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "cutoff": {
         "title": "Cutoff",
         "type": "number"
      },
      "order": {
         "title": "Order",
         "default": 10,
         "type": "integer"
      }
   },
   "required": [
      "cutoff"
   ]
}
```

field cutoff: float [Required]

field order: int = 10

run(*time_data:* TimeData) → *TimeData*

Apply the high pass filter

Parameters
time_data (TimeData) – The input TimeData

Returns
The high pass filtered TimeData

Return type
TimeData

Raises
ProcessRunError – If cutoff > nyquist

pydantic model resistics.time.**BandPass**

Bases: *TimeProcess*

Band pass filter time data

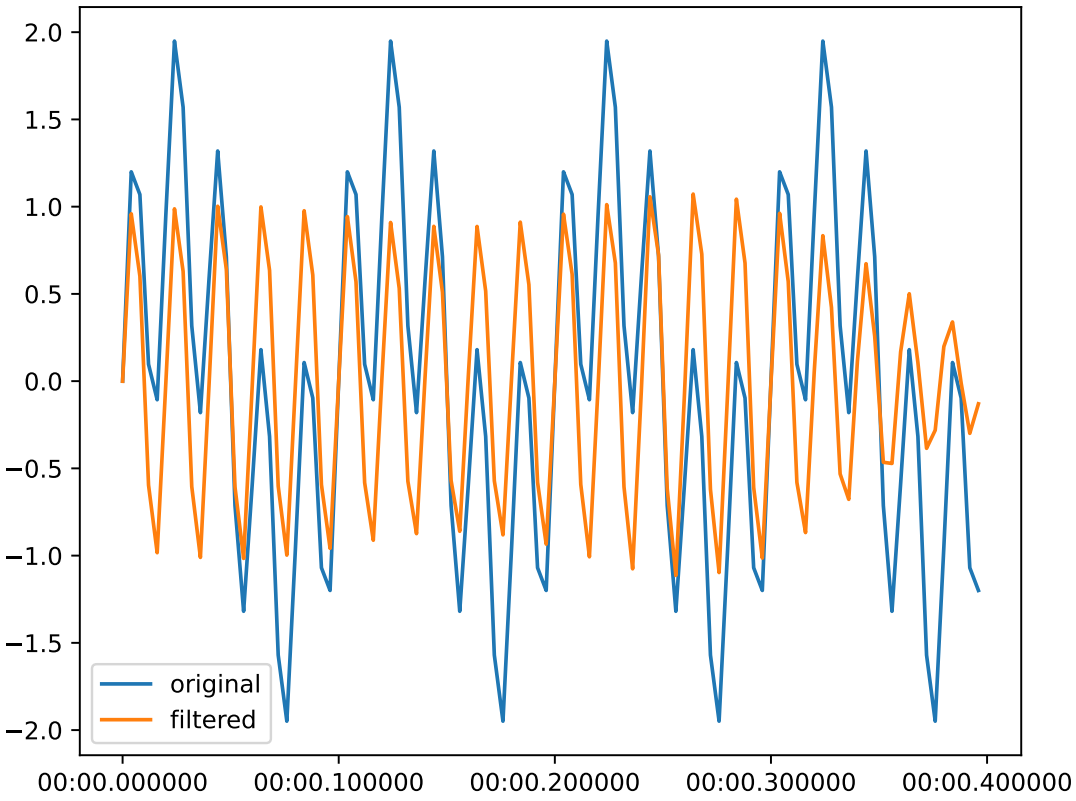
Parameters

- **cutoff_low** (*float*) – The low cutoff for the band pass filter
- **cutoff_high** (*float*) – The high cutoff for the band pass filter
- **order** (*int, optional*) – The order of the filter, by default 10

Examples

Band pass to isolate 12 Hz signal

```
import matplotlib.pyplot as plt
from resistics.testing import time_data_periodic
from resistics.time import BandPass
time_data = time_data_periodic([10, 50], fs=250, n_samples=100)
process = BandPass(cutoff_low=45, cutoff_high=55)
filtered = process.run(time_data)
plt.plot(time_data.get_timestamps(), time_data["chan1"], label="original")
plt.plot(filtered.get_timestamps(), filtered["chan1"], label="filtered")
plt.legend(loc=3)
plt.tight_layout()
plt.plot()
```




```
{
   "title": "BandPass",
   "description": "Band pass filter time data\n\nParameters\n----------\ncutoff_low␣
↪: float\n    The low cutoff for the band pass filter\ncutoff_high : float\n   ␣
↪The high cutoff for the band pass filter\norder : int, optional\n    The order of␣
↪the filter, by default 10\n\nExamples\n--------\nBand pass to isolate 12 Hz␣
↪signal\n\n.. plot::\n    :width: 90%\n\n    import matplotlib.pyplot as plt\n   ␣
↪from resistics.testing import time_data_periodic\n    from resistics.time import␣
↪BandPass\n    time_data = time_data_periodic([10, 50], fs=250, n_samples=100)\n  ␣
```

(continues on next page)

(continued from previous page)

```
↪ process = BandPass(cutoff_low=45, cutoff_high=55)\n    filtered = process.
↪run(time_data)\n    plt.plot(time_data.get_timestamps(), time_data[\"chan1\"],␣
↪label=\"original\")\n    plt.plot(filtered.get_timestamps(), filtered[\"chan1\"],␣
↪label=\"filtered\")\n    plt.legend(loc=3)\n    plt.tight_layout()\n    plt.plot()
↪",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "cutoff_low": {
         "title": "Cutoff Low",
         "type": "number"
      },
      "cutoff_high": {
         "title": "Cutoff High",
         "type": "number"
      },
      "order": {
         "title": "Order",
         "default": 10,
         "type": "integer"
      }
   },
   "required": [
      "cutoff_low",
      "cutoff_high"
   ]
}
```

field cutoff_low: float [Required]

field cutoff_high: float [Required]

field order: int = 10

run(*time_data:* TimeData) → *TimeData*

Apply the band pass filter

Parameters
time_data (TimeData) – The input TimeData

Returns
The band pass filtered TimeData

Return type
TimeData

Raises

- ***ProcessRunError*** – If cutoff_low > cutoff_high
- ***ProcessRunError*** – If cutoff_high > nyquist

pydantic model resistics.time.**Notch**

Bases: *TimeProcess*

Notch filter time data

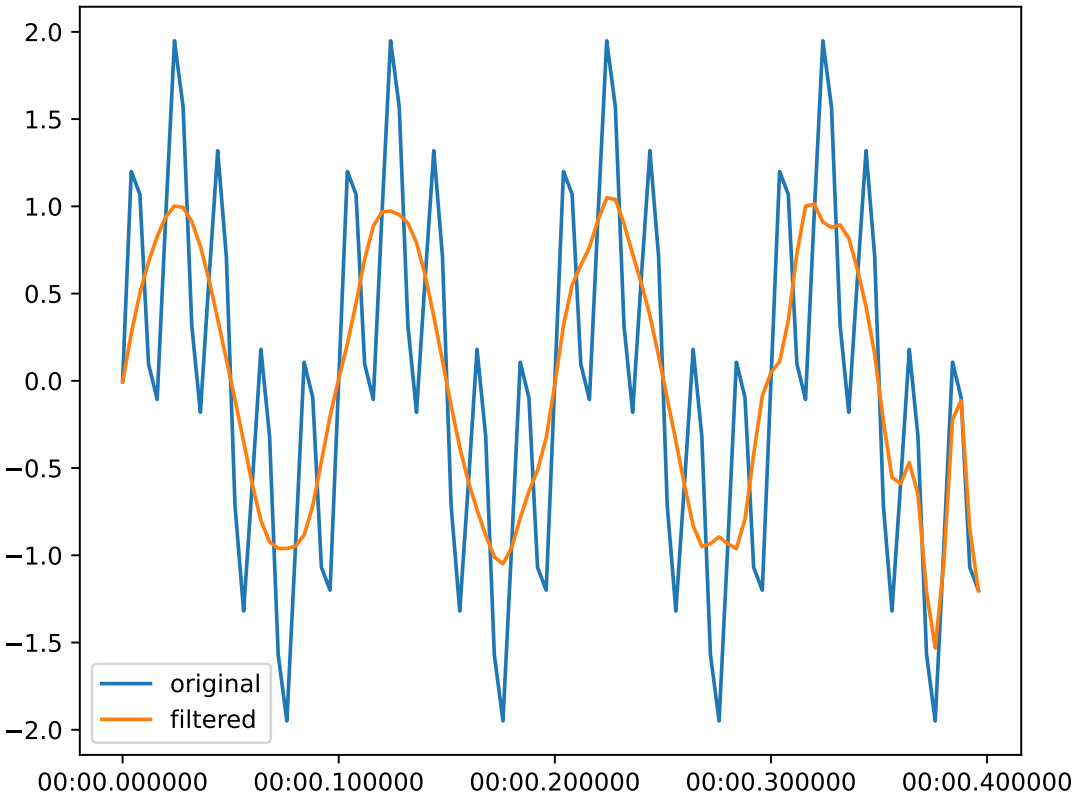
Parameters

- **notch** (*float*) – The frequency to notch
- **band** (*Optional[float], optional*) – The bandwidth of the filter, by default None
- **order** (*int, optional*) – The order of the filter, by default 10

Examples

Notch to remove a 50 Hz signal, for example powerline noise

```
import matplotlib.pyplot as plt
from resistics.testing import time_data_periodic
from resistics.time import Notch
time_data = time_data_periodic([10, 50], fs=250, n_samples=100)
process = Notch(notch=50, band=10)
filtered = process.run(time_data)
plt.plot(time_data.get_timestamps(), time_data["chan1"], label="original")
plt.plot(filtered.get_timestamps(), filtered["chan1"], label="filtered")
plt.legend(loc=3)
plt.tight_layout()
plt.plot()
```



```
{
   "title": "Notch",
   "description": "Notch filter time data\n\nParameters\n----------\nnotch : float\
↪n    The frequency to notch\nband : Optional[float], optional\n    The bandwidth␣
↪of the filter, by default None\norder : int, optional\n    The order of the␣
↪filter, by default 10\n\nExamples\n--------\nNotch to remove a 50 Hz signal, for␣
↪example powerline noise\n\n.. plot::\n    :width: 90%\n\n    import matplotlib.
↪pyplot as plt\n    from resistics.testing import time_data_periodic\n    from␣
↪resistics.time import Notch\n    time_data = time_data_periodic([10, 50], fs=250,␣
↪n_samples=100)\n    process = Notch(notch=50, band=10)\n    filtered = process.
↪run(time_data)\n    plt.plot(time_data.get_timestamps(), time_data[\"chan1\"],␣
↪label=\"original\")\n    plt.plot(filtered.get_timestamps(), filtered[\"chan1\"],␣
↪label=\"filtered\")\n    plt.legend(loc=3)\n    plt.tight_layout()\n    plt.plot()
↪",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "notch": {
         "title": "Notch",
         "type": "number"
      },
      "band": {
         "title": "Band",
         "type": "number"
      },
      "order": {
         "title": "Order",
         "default": 10,
         "type": "integer"
      }
   },
   "required": [
      "notch"
   ]
}
```


field notch: float [Required]

field band: float | None = None

field order: int = 10

run(*time_data:* TimeData) → *TimeData*

Apply notch filter to TimeData

Parameters
time_data (TimeData) – Input TimeData

Returns
Filtered TimeData

Return type
TimeData

Raises
ProcessRunError – If notch frequency > nyquist

pydantic model resistics.time.**Resample**

Bases: *TimeProcess*

Resample TimeData

Note that resampling is done on np.float64 data and this will lead to a temporary increase in memory usage. Once resampling is complete, the data is converted back to its original data type.

Parameters
new_fs (*int*) – The new sampling frequency

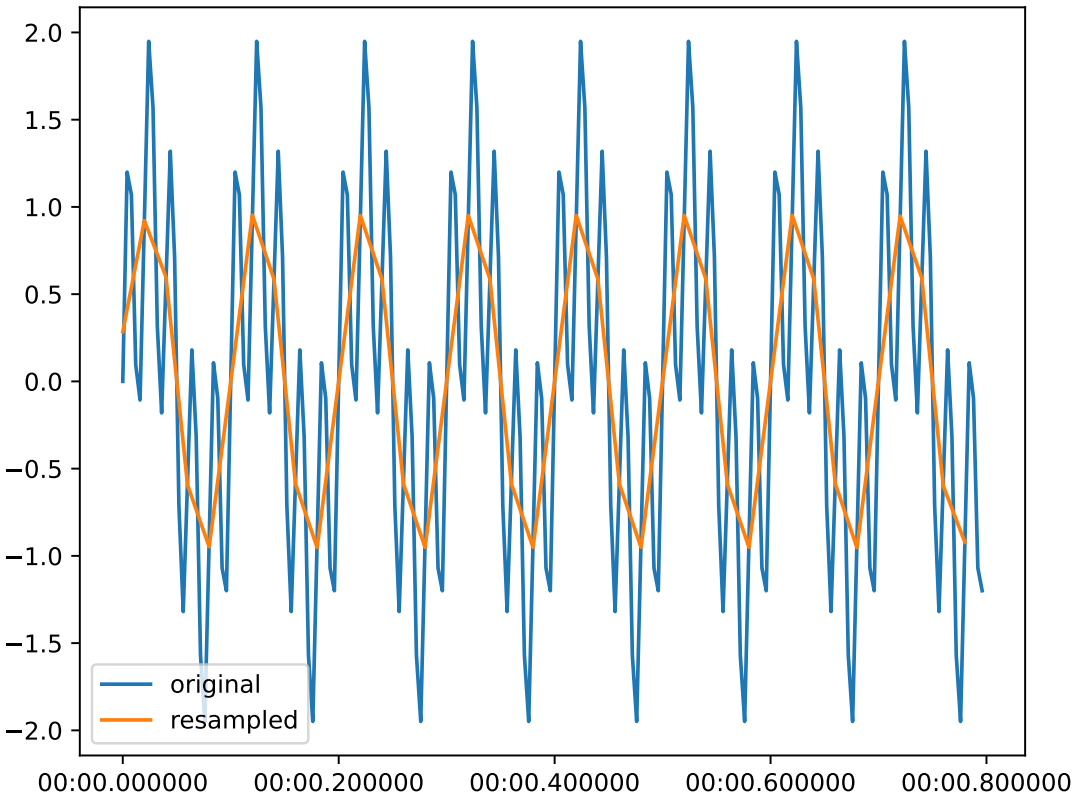
Examples

Resample the data from 250 Hz to 50 Hz

```
>>> import matplotlib.pyplot as plt
>>> from resistics.testing import time_data_periodic
>>> from resistics.time import Resample
>>> time_data = time_data_periodic([10, 50], fs=250, n_samples=200)
>>> print(time_data.metadata.n_samples, time_data.metadata.first_time, time_data.
↪metadata.last_time)
200 2020-01-01 00:00:00 2020-01-01 00:00:00.796
>>> process = Resample(new_fs=50)
>>> resampled = process.run(time_data)
>>> print(resampled.metadata.n_samples, resampled.metadata.first_time, resampled.
↪metadata.last_time)
40 2020-01-01 00:00:00 2020-01-01 00:00:00.78
>>> plt.plot(time_data.get_timestamps(), time_data["chan1"], label="original")
>>> plt.plot(resampled.get_timestamps(), resampled["chan1"], label="resampled")
>>> plt.legend(loc=3)
>>> plt.tight_layout()
>>> plt.show()
```

```
{
   "title": "Resample",
   "description": "Resample TimeData\n\nNote that resampling is done on np.float64␣
↪data and this will lead to a\ntemporary increase in memory usage. Once resampling␣
↪is complete, the data is\nconverted back to its original data type.\n\nParameters\
↪n----------\nnew_fs : int\n    The new sampling frequency\n\nExamples\n--------\
↪nResample the data from 250 Hz to 50 Hz\n\n.. plot::\n    :width: 90%\n\n    >>>␣
↪import matplotlib.pyplot as plt\n    >>> from resistics.testing import time_data_
↪periodic\n    >>> from resistics.time import Resample\n    >>> time_data = time_
↪data_periodic([10, 50], fs=250, n_samples=200)\n    >>> print(time_data.metadata.
↪n_samples, time_data.metadata.first_time, time_data.metadata.last_time)\n    200␣
↪2020-01-01 00:00:00 2020-01-01 00:00:00.796\n    >>> process = Resample(new_
↪fs=50)\n    >>> resampled = process.run(time_data)\n    >>> print(resampled.
↪metadata.n_samples, resampled.metadata.first_time, resampled.metadata.last_time)\
↪n    40 2020-01-01 00:00:00 2020-01-01 00:00:00.78\n    >>> plt.plot(time_data.
↪get_timestamps(), time_data[\"chan1\"], label=\"original\") # doctest: +SKIP\n    ␣
↪>>> plt.plot(resampled.get_timestamps(), resampled[\"chan1\"], label=\"resampled\
↪") # doctest: +SKIP\n    >>> plt.legend(loc=3) # doctest: +SKIP\n    >>> plt.
```

(continues on next page)



(continued from previous page)

```
↪tight_layout() # doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "new_fs": {
         "title": "New Fs",
         "type": "number"
      }
   },
   "required": [
      "new_fs"
   ]
}
```

field new_fs: float **[Required]**

run(*time_data:* TimeData) → *TimeData*

Resample TimeData

Resampling uses the polyphase method which does not assume periodicity Calculate the upsample rate and

the downsampling rate and using polyphase filtering, the final sample rate is:

$$(up/down) * originalsamplerate$$

Therefore, to get a sampling frequency of resampFreq, want:

$$(resampFreq/sampleFreq) * sampleFreq$$

Use the fractions library to get up and down as integers which they are required to be.

Parameters
time_data (TimeData) – Input TimeData

Returns
Resampled TimeData

Return type
TimeData

pydantic model resistics.time.**Decimate**

Bases: *TimeProcess*

Decimate TimeData

Warning: Data is converted to np.float64 prior to decimation. This is going to cause a temporary increase in memory usage, but decimating np.float64 delivers improved results.

The decimated data is converted back to its original data type prior to being returned.

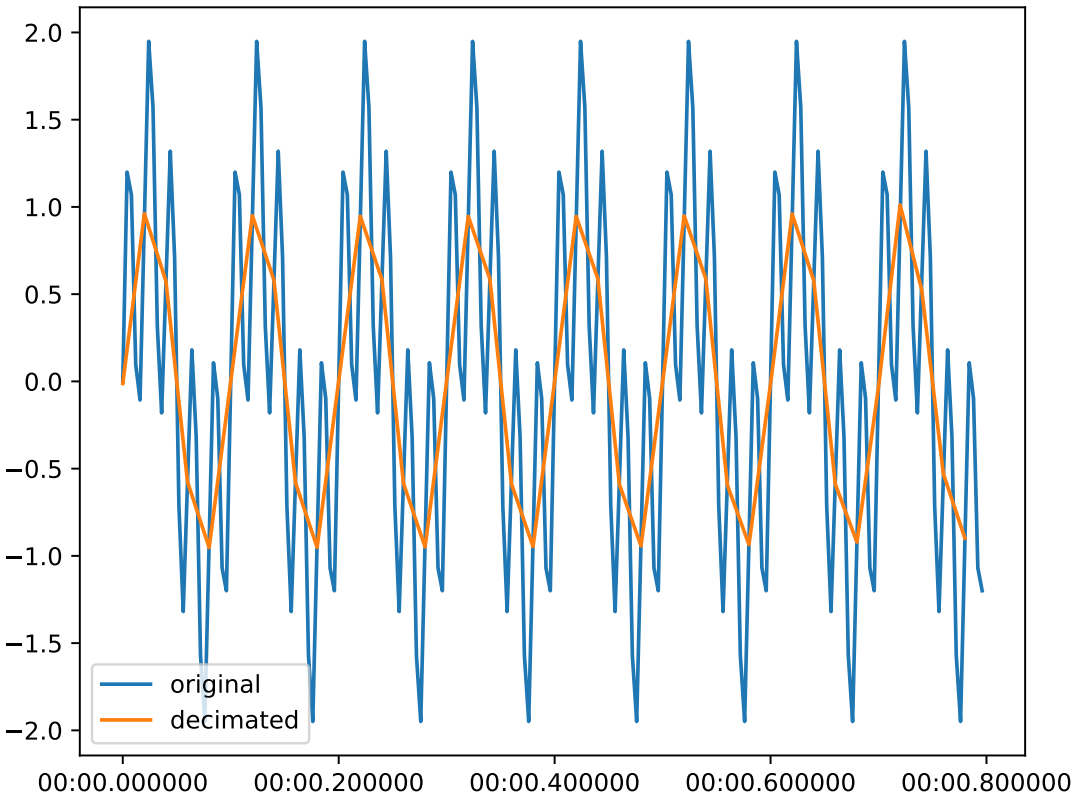
The max_factor for a single decimation step is by default set as 3. When using np.float64 data, it is possible to use a larger decimation factor, up to 13, but this does again have an impact on results.

For more information, see https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.decimate.html

Parameters
factor (*int*) – The decimation factor

Examples

```
>>> import matplotlib.pyplot as plt
>>> from resistics.testing import time_data_periodic
>>> from resistics.time import Decimate
>>> time_data = time_data_periodic([10, 50], fs=250, n_samples=200)
>>> print(time_data.metadata.n_samples, time_data.metadata.first_time, time_data.
↪metadata.last_time)
200 2020-01-01 00:00:00 2020-01-01 00:00:00.796
>>> process = Decimate(factor=5)
>>> decimated = process.run(time_data)
>>> print(decimated.metadata.n_samples, decimated.metadata.first_time, decimated.
↪metadata.last_time)
40 2020-01-01 00:00:00 2020-01-01 00:00:00.78
>>> plt.plot(time_data.get_timestamps(), time_data["chan1"], label="original")
>>> plt.plot(decimated.get_timestamps(), decimated["chan1"], label="decimated")
>>> plt.legend(loc=3)
>>> plt.tight_layout()
>>> plt.show()
```




```
{
   "title": "Decimate",
   "description": "Decimate TimeData\n\n.. warning::\n\n    Data is converted to np.
↪float64 prior to decimation. This is going to\n    cause a temporary increase in␣
↪memory usage, but decimating np.float64\n    delivers improved results.\n\n    ␣
↪The decimated data is converted back to its original data type prior\n    to␣
↪being returned.\n\n    The max_factor for a single decimation step is by default␣
↪set as 3.\n    When using np.float64 data, it is possible to use a larger␣
↪decimation\n    factor, up to 13, but this does again have an impact on results.\
↪n\n    For more information, see\n    https://docs.scipy.org/doc/scipy/reference/
↪generated/scipy.signal.decimate.html\n\nParameters\n----------\nfactor : int\n    ␣
↪The decimation factor\n\nExamples\n--------\n.. plot::\n    :width: 90%\n\n    >>>
↪ import matplotlib.pyplot as plt\n    >>> from resistics.testing import time_data_
↪periodic\n    >>> from resistics.time import Decimate\n    >>> time_data = time_
↪data_periodic([10, 50], fs=250, n_samples=200)\n    >>> print(time_data.metadata.
↪n_samples, time_data.metadata.first_time, time_data.metadata.last_time)\n    200␣
↪2020-01-01 00:00:00 2020-01-01 00:00:00.796\n    >>> process = Decimate(factor=5)\
↪n    >>> decimated = process.run(time_data)\n    >>> print(decimated.metadata.n_
↪samples, decimated.metadata.first_time, decimated.metadata.last_time)\n    40␣
↪2020-01-01 00:00:00 2020-01-01 00:00:00.78\n    >>> plt.plot(time_data.get_
↪timestamps(), time_data[\"chan1\"], label=\"original\") # doctest: +SKIP\n    >>>␣
↪plt.plot(decimated.get_timestamps(), decimated[\"chan1\"], label=\"decimated\") #␣
↪doctest: +SKIP\n    >>> plt.legend(loc=3) # doctest: +SKIP\n    >>> plt.tight_
↪layout() # doctest: +SKIP\n    >>> plt.show() # doctest: +SKIP",
```


(continues on next page)

(continued from previous page)

```
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "factor": {
         "title": "Factor",
         "minimum": 1,
         "type": "integer"
      },
      "max_single_factor": {
         "title": "Max Single Factor",
         "default": 3,
         "minimum": 2,
         "type": "integer"
      }
   },
   "required": [
      "factor"
   ]
}
```

field factor: ConstrainedIntValue [Required]

Constraints

- **minimum** = 1

field max_single_factor: ConstrainedIntValue = 3

Constraints

- **minimum** = 2

run(*time_data:* TimeData) → *TimeData*

Decimate TimeData

Parameters
time_data (TimeData) – Input TimeData

Returns
Decimated TimeData

Return type
TimeData

pydantic model resistics.time.**ShiftTimestamps**

Bases: *TimeProcess*

Shift timestamps. This method is usually used when there is an offset on the sampling, so that instead of coinciding with a second or an hour, they are offset from this.

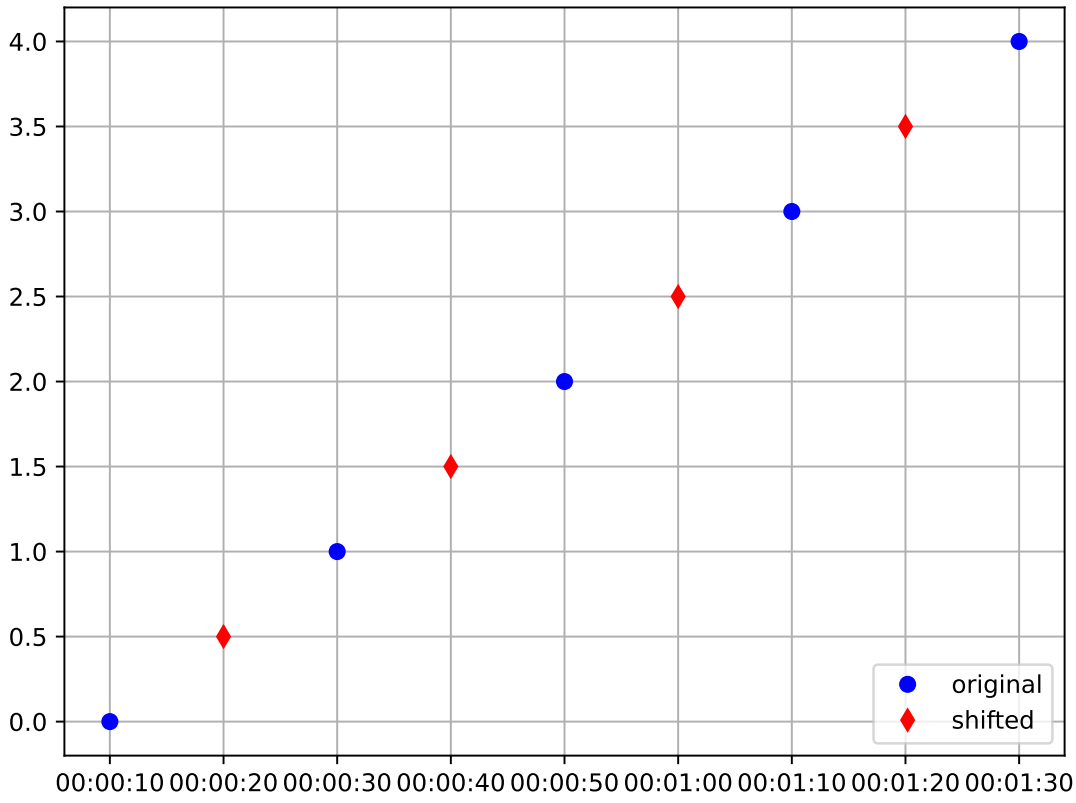
The function interpolates the original data onto the shifted timestamps.

Parameters
shift (*float*) – The shift in seconds. This must be positive as data is never extrapolated

Examples

An example shifting timestamps for TimeData with a sample period of 20 seconds (fs = 1/20 = 0.05 Hz) but with an offset of 10 seconds on the timestamps

```
>>> from resistics.testing import time_data_with_offset
>>> from resistics.time import ShiftTimestamps
>>> time_data = time_data_with_offset(offset=10, fs=1/20, n_samples=5)
>>> [x.time().strftime('%H:%M:%S') for x in time_data.get_timestamps()]
['00:00:10', '00:00:30', '00:00:50', '00:01:10', '00:01:30']
>>> process = ShiftTimestamps(shift=10, style="linear")
>>> result = process.run(time_data)
>>> [x.time().strftime('%H:%M:%S') for x in result.get_timestamps()]
['00:00:20', '00:00:40', '00:01:00', '00:01:20']
>>> plt.plot(time_data.get_timestamps(), time_data["chan1"], "bo", label="original")
>>> plt.plot(result.get_timestamps(), result["chan1"], "rd", label="shifted")
>>> plt.legend(loc=4)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```



See also:

CropTimestamps
: Crop timestamps to a specified time unit

```
{
   "title": "ShiftTimestamps",
   "description": "Shift timestamps. This method is usually used when there is an␣
↪offset on the\nsampling, so that instead of coinciding with a second or an hour,␣
↪they are\noffset from this.\n\nThe function interpolates the original data onto␣
↪the shifted timestamps.\n\nParameters\n----------\nshift : float\n    The shift␣
↪in seconds. This must be positive as data is never\n    extrapolated\n\nExamples\
↪n--------\nAn example shifting timestamps for TimeData with a sample period of 20\
↪nseconds (fs = 1/20 = 0.05 Hz) but with an offset of 10 seconds on the\
↪ntimestamps\n\n.. plot::\n    :width: 90%\n\n    >>> from resistics.testing␣
↪import time_data_with_offset\n    >>> from resistics.time import ShiftTimestamps\
↪n    >>> time_data = time_data_with_offset(offset=10, fs=1/20, n_samples=5)\n    >
↪>> [x.time().strftime('%H:%M:%S') for x in time_data.get_timestamps()]\n    [
↪'00:00:10', '00:00:30', '00:00:50', '00:01:10', '00:01:30']\n    >>> process =␣
↪ShiftTimestamps(shift=10, style=\"linear\")\n    >>> result = process.run(time_
↪data)\n    >>> [x.time().strftime('%H:%M:%S') for x in result.get_timestamps()]\n␣
↪   ['00:00:20', '00:00:40', '00:01:00', '00:01:20']\n    >>> plt.plot(time_data.
↪get_timestamps(), time_data[\"chan1\"], \"bo\", label=\"original\") # doctest:␣
↪+SKIP\n    >>> plt.plot(result.get_timestamps(), result[\"chan1\"], \"rd\",␣
↪label=\"shifted\") # doctest: +SKIP\n    >>> plt.legend(loc=4) # doctest: +SKIP\n␣
↪   >>> plt.grid() # doctest: +SKIP\n    >>> plt.tight_layout() # doctest: +SKIP\n␣
↪   >>> plt.show() # doctest: +SKIP\n\nSee Also\n--------\nCropTimestamps : Crop␣
↪timestamps to a specified time unit",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "shift": {
         "title": "Shift",
         "exclusiveMinimum": 0,
         "type": "number"
      },
      "style": {
         "title": "Style",
         "default": "spline",
         "enum": [
            "spline",
            "linear"
         ],
         "type": "string"
      }
   },
   "required": [
      "shift"
   ]
}
```


field shift: PositiveFloat [Required]

Constraints

- **exclusiveMinimum** = 0

field style: `Literal['spline', 'linear'] = 'spline'`

run(*time_data:* TimeData) → *TimeData*

Shift timestamps and interpolate data

Parameters
time_data (TimeData) – Input TimeData

Returns
TimeData with shifted timestamps and data interpolated

Return type
TimeData

Raises
ProcessRunError – If the shift is greater than the sampling frequency. This method is not supposed to be used for resampling, but simply for removing an offset from timestamps

pydantic model resistics.time.**CropTimestamps**

Bases: *TimeProcess*

Crop timestamps to make them begin at the next second or minute or hour and end one sample before the nearest second, minute or hour.

Input timestamps should be sampled coincidentally with the time unit. This can be achieved with *ShiftTimestamps*.

Cropping works as follows:

- First times are cropped to the next time unit or maintained if they are a whole time unit already
- Last times are cropped to the previous time unit or maintained if they are a whole time unit already

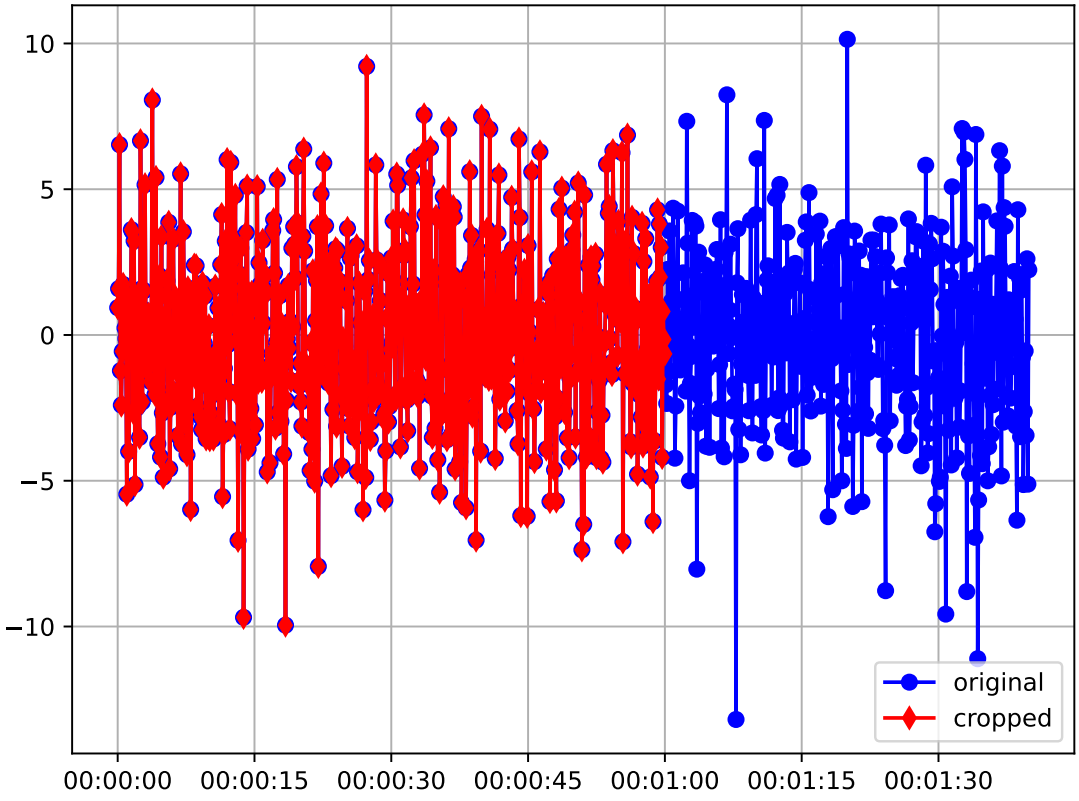
Parameters
time_unit (*str*) – The time unit to crop to given in pandas style time freq

Examples

An example cropping timestamps to the nearest minute.

```
>>> from resistics.testing import time_data_random
>>> from resistics.time import CropTimestamps
>>> time_data = time_data_random(n_samples=1000)
>>> print(time_data.metadata.first_time, time_data.metadata.last_time)
2020-01-01 00:00:00 2020-01-01 00:01:39.9
>>> process = CropTimestamps(time_unit="T")
>>> result = process.run(time_data)
>>> print(result.metadata.first_time, result.metadata.last_time)
2020-01-01 00:00:00 2020-01-01 00:01:00
>>> plt.plot(time_data.get_timestamps(), time_data["Ex"], "bo-", label="original")
>>> plt.plot(result.get_timestamps(), result["Ex"], "rd-", label="cropped")
>>> plt.legend(loc=4)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

See also:



ShiftTimestamps

Calculate data values on shifted timestamps

```
{
   "title": "CropTimestamps",
   "description": "Crop timestamps to make them begin at the next second or minute␣
↪or hour\nand end one sample before the nearest second, minute or hour.\n\nInput␣
↪timestamps should be sampled coincidentally with the time\nunit. This can be␣
↪achieved with :class:`~ShiftTimestamps`.\n\nCropping works as follows:\n\n- First␣
↪times are cropped to the next time unit or maintained if they are a\n  whole time␣
↪unit already\n- Last times are cropped to the previous time unit or maintained if␣
↪they are\n  a whole time unit already\n\nParameters\n----------\ntime_unit : str\
↪n    The time unit to crop to given in pandas style time freq\n\nExamples\n-------
↪-\nAn example cropping timestamps to the nearest minute.\n\n.. plot::\n    ␣
↪:width: 90%\n\n    >>> from resistics.testing import time_data_random\n    >>>␣
↪from resistics.time import CropTimestamps\n    >>> time_data = time_data_random(n_
↪samples=1000)\n    >>> print(time_data.metadata.first_time, time_data.metadata.
↪last_time)\n    2020-01-01 00:00:00 2020-01-01 00:01:39.9\n    >>> process =␣
↪CropTimestamps(time_unit=\"T\")\n    >>> result = process.run(time_data)\n    >>>␣
↪print(result.metadata.first_time, result.metadata.last_time)\n    2020-01-01␣
↪00:00:00 2020-01-01 00:01:00\n    >>> plt.plot(time_data.get_timestamps(), time_
↪data[\"Ex\"], \"bo-\", label=\"original\") # doctest: +SKIP\n    >>> plt.
↪plot(result.get_timestamps(), result[\"Ex\"], \"rd-\", label=\"cropped\") #␣
↪doctest: +SKIP\n    >>> plt.legend(loc=4) # doctest: +SKIP\n    >>> plt.grid() #␣
↪doctest: +SKIP\n    >>> plt.tight_layout() # doctest: +SKIP\n    >>> plt.show() #␣
```

(continues on next page)

(continued from previous page)

```
↪doctest: +SKIP\n\nSee Also\n--------\nShiftTimestamps : Calculate data values on␣
↪shifted timestamps",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "time_unit": {
         "title": "Time Unit",
         "default": "s",
         "type": "string"
      }
   }
}
```

field time_unit: `str = 's'`

run(*time_data:* TimeData) → *TimeData*

Crop timestamps to the next time unit

Parameters
time_data (TimeData) – The TimeData to crop

Returns
The cropped TimeData

Return type
TimeData

resistics.time.**serialize_custom_fnc**(*fnc: Callable*) → str

Serialize the custom functions

This is not really reversible and recovering parameters from ApplyFunction is not supported

Parameters
fnc (*Callable*) – Function to serialize

Returns
serialized output

Return type
str

pydantic model resistics.time.**ApplyFunction**

Bases: *TimeProcess*

Apply a generic functions to the time data

To be used with single argument functions that take the channel data array and a perform transformation on the data.

Parameters
fncs (*Dict[str, Callable]*) – Dictionary of channel to callable

Examples

```
>>> import numpy as np
>>> from resistics.testing import time_data_ones
>>> from resistics.time import ApplyFunction
>>> time_data = time_data_ones()
>>> process = ApplyFunction(fncs={"Ex": lambda x: 2*x, "Hy": lambda x: 3*x*x - 5*x␣
↪+ 1})
>>> result = process.run(time_data)
>>> time_data["Ex"]
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32)
>>> result["Ex"]
array([2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
>>> time_data["Hy"]
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32)
>>> result["Hy"]
array([-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.])
```


```
{
   "title": "ApplyFunction",
   "description": "Apply a generic functions to the time data\n\nTo be used with␣
↪single argument functions that take the channel data array\nand a perform␣
↪transformation on the data.\n\nParameters\n----------\nfncs : Dict[str, Callable]\
↪n    Dictionary of channel to callable\n\nExamples\n--------\n>>> import numpy as␣
↪np\n>>> from resistics.testing import time_data_ones\n>>> from resistics.time␣
↪import ApplyFunction\n>>> time_data = time_data_ones()\n>>> process =␣
↪ApplyFunction(fncs={\"Ex\": lambda x: 2*x, \"Hy\": lambda x: 3*x*x - 5*x + 1})\n>>
↪> result = process.run(time_data)\n>>> time_data[\"Ex\"]\narray([1., 1., 1., 1.,␣
↪1., 1., 1., 1., 1., 1.], dtype=float32)\n>>> result[\"Ex\"]\narray([2., 2., 2., 2.
↪, 2., 2., 2., 2., 2., 2.])\n>>> time_data[\"Hy\"]\narray([1., 1., 1., 1., 1., 1.,␣
↪1., 1., 1., 1.], dtype=float32)\n>>> result[\"Hy\"]\narray([-1., -1., -1., -1., -
↪1., -1., -1., -1., -1., -1.])",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```


field fncs: `Dict[str, Callable]` **[Required]**

run(*time_data:* TimeData) → *TimeData*

Apply functions to channel data

Parameters
time_data (TimeData) – Input TimeData

Returns
Transformed TimeData

Return type
TimeData

resistics.transfunc module

Module defining transfer functions

pydantic model resistics.transfunc.**Component**

Bases: *Metadata*

Data class for a single component in a Transfer function

Example

```
>>> from resistics.transfunc import Component
>>> component = Component(real=[1, 2, 3, 4, 5], imag=[-5, -4, -3, -2 , -1])
>>> component.get_value(0)
(1-5j)
>>> component.to_numpy()
array([1.-5.j, 2.-4.j, 3.-3.j, 4.-2.j, 5.-1.j])
```

```
{
   "title": "Component",
   "description": "Data class for a single component in a Transfer function\n\
↪nExample\n-------\n>>> from resistics.transfunc import Component\n>>> component =␣
↪Component(real=[1, 2, 3, 4, 5], imag=[-5, -4, -3, -2 , -1])\n>>> component.get_
↪value(0)\n(1-5j)\n>>> component.to_numpy()\narray([1.-5.j, 2.-4.j, 3.-3.j, 4.-2.j,
↪ 5.-1.j])",
   "type": "object",
   "properties": {
      "real": {
         "title": "Real",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "imag": {
         "title": "Imag",
         "type": "array",
         "items": {
            "type": "number"
         }
      }
   },
   "required": [
      "real",
      "imag"
   ]
}
```

field real: List[float] **[Required]**

The real part of the component

field imag: List[float] **[Required]**

The complex part of the component

get_value(*eval_idx: int*) → complex

Get the value for an evaluation frequency

to_numpy() → ndarray

Get the component as a numpy complex array

resistics.transfunc.**get_component_key**(*out_chan: str*, *in_chan: str*) → str

Get key for out channel and in channel combination in the solution

Parameters

- **out_chan** (*str*) – The output channel
- **in_chan** (*str*) – The input channel

Returns

The component key

Return type

str

Examples

```
>>> from resistics.regression import get_component_key
>>> get_component_key("Ex", "Hy")
'ExHy'
```

pydantic model resistics.transfunc.**TransferFunction**

Bases: *Metadata*

Define a generic transfer function

This class is a describes generic transfer function, including:

- The output channels for the transfer function
- The input channels for the transfer function
- The cross channels for the transfer function

The cross channels are the channels that will be used to calculate out the cross powers for the regression.

This generic parent class has no implemented plotting function. However, child classes may have a plotting function as different transfer functions may need different types of plots.

Note: Users interested in writing a custom transfer function should inherit from this generic Transfer function

---

See also:

ImpandanceTensor

Transfer function for the MT impedance tensor

Tipper

Transfer function for the MT tipper

Examples

A generic example

```
>>> from resistics.transfunc import TransferFunction
>>> tf = TransferFunction(variation="example", out_chans=["bye", "see you", "ciao"],
↪ in_chans=["hello", "hi_there"])
>>> print(tf.to_string())
| bye      |   | bye_hello         bye_hi_there      | | hello    |
| see you  | = | see you_hello     see you_hi_there  | | hi_there |
| ciao     |   | ciao_hello        ciao_hi_there     |
```

Combining the impedance tensor and the tipper into one TransferFunction

```
>>> tf = TransferFunction(variation="combined", out_chans=["Ex", "Ey"], in_chans=[
↪"Hx", "Hy", "Hz"])
>>> print(tf.to_string())
| Ex |   | Ex_Hx Ex_Hy Ex_Hz | | Hx |
| Ey | = | Ey_Hx Ey_Hy Ey_Hz | | Hy |
                               | Hz |
```

```
{
   "title": "TransferFunction",
   "description": "Define a generic transfer function\n\nThis class is a describes␣
↪generic transfer function, including:\n\n- The output channels for the transfer␣
↪function\n- The input channels for the transfer function\n- The cross channels␣
↪for the transfer function\n\nThe cross channels are the channels that will be␣
↪used to calculate out the\ncross powers for the regression.\n\nThis generic␣
↪parent class has no implemented plotting function. However,\nchild classes may␣
↪have a plotting function as different transfer functions\nmay need different␣
↪types of plots.\n\n.. note::\n\n    Users interested in writing a custom transfer␣
↪function should inherit\n    from this generic Transfer function\n\nSee Also\n----
↪----\nImpandanceTensor : Transfer function for the MT impedance tensor\nTipper :␣
↪Transfer function for the MT tipper\n\nExamples\n--------\nA generic example\n\n>>
↪> from resistics.transfunc import TransferFunction\n>>> tf =␣
↪TransferFunction(variation=\"example\", out_chans=[\"bye\", \"see you\", \"ciao\
↪"], in_chans=[\"hello\", \"hi_there\"])\n>>> print(tf.to_string())\n| bye      | ␣
↪ | bye_hello         bye_hi_there      | | hello    |\n| see you  | = | see you_
↪hello     see you_hi_there  | | hi_there |\n| ciao     |   | ciao_hello       ␣
↪ciao_hi_there     |\n\nCombining the impedance tensor and the tipper into one␣
↪TransferFunction\n\n>>> tf = TransferFunction(variation=\"combined\", out_chans=[\
↪"Ex\", \"Ey\"], in_chans=[\"Hx\", \"Hy\", \"Hz\"])\n>>> print(tf.to_string())\n|␣
↪Ex |   | Ex_Hx Ex_Hy Ex_Hz | | Hx |\n| Ey | = | Ey_Hx Ey_Hy Ey_Hz | | Hy |\n     ␣
↪                           | Hz |",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "variation": {
         "title": "Variation",
         "default": "generic",
```

(continues on next page)

(continued from previous page)


```
            "maxLength": 16,
            "type": "string"
         },
         "out_chans": {
            "title": "Out Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "in_chans": {
            "title": "In Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "cross_chans": {
            "title": "Cross Chans",
            "type": "array",
            "items": {
               "type": "string"
            }
         },
         "n_out": {
            "title": "N Out",
            "type": "integer"
         },
         "n_in": {
            "title": "N In",
            "type": "integer"
         },
         "n_cross": {
            "title": "N Cross",
            "type": "integer"
         }
      },
      "required": [
         "out_chans",
         "in_chans"
      ]
}
```


field name: str | None = None

The name of the transfer function, this will be set automatically

Validated by

- validate_name

field variation: ConstrainedStrValue = 'generic'

A short additional bit of information about this variation

Constraints

- **maxLength** = 16

field out_chans: List[str] [Required]

The output channels

field in_chans: List[str] [Required]

The input channels

field cross_chans: List[str] | None = None

The channels to use for calculating the cross spectra

Validated by

- validate_cross_chans

field n_out: int | None = None

The number of output channels

Validated by

- validate_n_out

field n_in: int | None = None

The number of input channels

Validated by

- validate_n_in

field n_cross: int | None = None

The number of cross power channels

Validated by

- validate_n_cross

classmethod validate(*value:* TransferFunction | *Dict[str, Any]*) → *TransferFunction*

Validate a TransferFunction

Parameters

value (*Union[TransferFunction, Dict[str, Any]]*) – A TransferFunction child class or a dictionary

Returns

A TransferFunction or TransferFunction child class

Return type

TransferFunction

Raises

- **ValueError** – If the value is neither a TransferFunction or a dictionary
- **KeyError** – If name is not in the dictionary
- **ValueError** – If initialising from dictionary fails

Examples

The following example will show how a child TransferFunction class can be instantiated using a dictionary and the parent TransferFunction (but only as long as that child class has been imported).

```
>>> from resistics.transfunc import TransferFunction
```

Show known TransferFunction types in built into resistics

```
>>> for entry in TransferFunction._types.items():
...     print(entry)
('ImpedanceTensor', <class 'resistics.transfunc.ImpedanceTensor'>)
('Tipper', <class 'resistics.transfunc.Tipper'>)
```

Now let's initialise an ImpedanceTensor from the base TransferFunction and a dictionary.

```
>>> mytf = {"name": "ImpedanceTensor", "variation": "ecross", "cross_chans": [
↪"Ex", "Ey"]}
>>> test = TransferFunction(**mytf)
Traceback (most recent call last):
...
KeyError: 'out_chans'
```

This is not quite what we were expecting. The generic TransferFunction requires out_chans to be defined, but they are not in the dictionary as the ImpedanceTensor child class defaults these. To get this to work, instead use the validate class method. This is the class method used by pydantic when instantiating.

```
>>> mytf = {"name": "ImpedanceTensor", "variation": "ecross", "cross_chans": [
↪"Ex", "Ey"]}
>>> test = TransferFunction.validate(mytf)
>>> test.summary()
{
    'name': 'ImpedanceTensor',
    'variation': 'ecross',
    'out_chans': ['Ex', 'Ey'],
    'in_chans': ['Hx', 'Hy'],
    'cross_chans': ['Ex', 'Ey'],
    'n_out': 2,
    'n_in': 2,
    'n_cross': 2
}
```

That's more like it. This will raise errors if an unknown type of TransferFunction is received.

```
>>> mytf = {"name": "NewTF", "cross_chans": ["Ex", "Ey"]}
>>> test = TransferFunction.validate(mytf)
Traceback (most recent call last):
...
ValueError: Unable to initialise NewTF from dictionary
```

Or if the dictionary does not have a name key

```
>>> mytf = {"cross_chans": ["Ex", "Ey"]}
>>> test = TransferFunction.validate(mytf)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
KeyError: 'No name provided for initialisation of TransferFunction'
```

Unexpected inputs will also raise an error

```
>>> test = TransferFunction.validate(5)
Traceback (most recent call last):
...
ValueError: TransferFunction unable to initialise from <class 'int'>
```

n_eqns_per_output() → int

Get the number of equations per output

n_regressors() → int

Get the number of regressors

solution_components() → List[str]

Get the components of the solution based on the input and output channels

Returns
The solution components

Return type
List[str]

Examples

```
>>> from resistics.transfunc import TransferFunction
>>> tf = TransferFunction(
...     name="example",
...     variation="a",
...     in_chans=["a", "b", "c"],
...     out_chans=["x", "y"]
... )
>>> tf.solution_components()
['xa', 'xb', 'xc', 'ya', 'yb', 'yc']
```

to_string()

Get the transfer function as as string

pydantic model resistics.transfunc.**ImpedanceTensor**

Bases: *TransferFunction*

Standard magnetotelluric impedance tensor

Notes

Information about data units

- Magnetic permeability in nT . m / A
- Electric (E) data is in mV/m
- Magnetic (H) data is in nT
- Z = E/H is in mV / m . nT
- Units of resistance = Ohm = V / A

Examples

```
>>> from resistics.transfunc import ImpedanceTensor
>>> tf = ImpedanceTensor()
>>> print(tf.to_string())
| Ex | = | Ex_Hx Ex_Hy | | Hx |
| Ey |   | Ey_Hx Ey_Hy | | Hy |
```

```
{
   "title": "ImpedanceTensor",
   "description": "Standard magnetotelluric impedance tensor\n\nNotes\n-----\
↪nInformation about data units\n\n- Magnetic permeability in nT . m / A\n-␣
↪Electric (E) data is in mV/m\n- Magnetic (H) data is in nT\n- Z = E/H is in mV /␣
↪m . nT\n- Units of resistance = Ohm = V / A\n\nExamples\n--------\n>>> from␣
↪resistics.transfunc import ImpedanceTensor\n>>> tf = ImpedanceTensor()\n>>>␣
↪print(tf.to_string())\n| Ex | = | Ex_Hx Ex_Hy | | Hx |\n| Ey |   | Ey_Hx Ey_Hy |␣
↪| Hy |",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "variation": {
         "title": "Variation",
         "default": "default",
         "maxLength": 16,
         "type": "string"
      },
      "out_chans": {
         "title": "Out Chans",
         "default": [
            "Ex",
            "Ey"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
```

(continues on next page)

(continued from previous page)

```
      "in_chans": {
         "title": "In Chans",
         "default": [
            "Hx",
            "Hy"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "cross_chans": {
         "title": "Cross Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "n_out": {
         "title": "N Out",
         "type": "integer"
      },
      "n_in": {
         "title": "N In",
         "type": "integer"
      },
      "n_cross": {
         "title": "N Cross",
         "type": "integer"
      }
   }
}
```

field variation: ConstrainedStrValue = 'default'

A short additional bit of information about this variation

Constraints

- **maxLength** = 16

field out_chans: List[str] = ['Ex', 'Ey']

The output channels

field in_chans: List[str] = ['Hx', 'Hy']

The input channels

static get_resistivity(*periods: ndarray*, *component:* Component) → ndarray

Get apparent resistivity for a component

Parameters

- **periods** (*np.ndarray*) – The periods of the component
- **component** (Component) – The component values

Returns
Apparent resistivity

Return type
np.ndarray

static get_phase(*key: str*, *component:* Component) → ndarray

Get the phase for the component

Note: Components ExHx and ExHy are wrapped around in [0,90]

Parameters

- **key** (*str*) – The component name
- **component** (Component) – The component values

Returns
The phase values

Return type
np.ndarray

static get_fig(*x_lim: List[float] | None = None*, *res_lim: List[float] | None = None*, *phs_lim: List[float] | None = None*) → Figure

Get a figure for plotting the ImpedanceTensor

Parameters

- **x_lim** (*Optional[List[float]], optional*) – The x limits, to be provided as powers of 10, by default None. For example, for 0.001, use -3
- **res_lim** (*Optional[List[float]], optional*) – The y limits for resistivity, to be provided as powers of 10, by default None. For example, for 1000, use 3
- **phs_lim** (*Optional[List[float]], optional*) – The phase limits, by default None

Returns
Plotly figure

Return type
go.Figure

static plot(*freqs: List[float]*, *components: Dict[str,* Component*]*, *fig: Figure | None = None*, *to_plot: List[str] | None = None*, *legend: str = 'Impedance tensor'*, *x_lim: List[float] | None = None*, *res_lim: List[float] | None = None*, *phs_lim: List[float] | None = None*, *symbol: str | None = 'circle'*) → Figure

Plot the Impedance tensor

Parameters

- **freqs** (*List[float]*) – The frequencies where the impedance tensor components have been calculated
- **components** (*Dict[str,* Component*]*) – The component data
- **fig** (*Optional[go.Figure], optional*) – Figure to add to, by default None
- **to_plot** (*Optional[List[str]], optional*) – The components to plot, by default all of the components of the impedance tensor

- **legend** (*str, optional*) – Legend prefix for the components, by default "Impedance tensor"
- **x_lim** (*Optional[List[float]], optional*) – The x limits, to be provided as powers of 10, by default None. For example, for 0.001, use -3. Only used when a figure is not provided.
- **res_lim** (*Optional[List[float]], optional*) – The y limits for resistivity, to be provided as powers of 10, by default None. For example, for 1000, use 3. Only used when a figure is not provided.
- **phs_lim** (*Optional[List[float]], optional*) – The phase limits, by default None. Only used when a figure is not provided.
- **symbol** (*Optional[str], optional*) – The marker symbol to use, by default "circle"

Returns
[description]

Return type
go.Figure

pydantic model resistics.transfunc.**Tipper**

Bases: *TransferFunction*

Magnetotelluric tipper

The tipper components are Tx = HzHx and Ty = HzHy

The tipper length is sqrt(Re(Tx)^2 + Re(Ty)^2)

The tipper angle is arctan (Re(Ty)/Re(Tx))

Notes

Information about units

- Tipper T = H/H is dimensionless

Examples

```
>>> from resistics.transfunc import Tipper
>>> tf = Tipper()
>>> print(tf.to_string())
| Hz | = | Hz_Hx Hz_Hy | | Hx |
                         | Hy |
```


```
{
   "title": "Tipper",
   "description": "Magnetotelluric tipper\n\nThe tipper components are Tx = HzHx␣
↪and Ty = HzHy\n\nThe tipper length is sqrt(Re(Tx)^2 + Re(Ty)^2)\n\nThe tipper␣
↪angle is arctan (Re(Ty)/Re(Tx))\n\nNotes\n-----\nInformation about units\n\n-␣
↪Tipper T = H/H is dimensionless\n\nExamples\n--------\n>>> from resistics.
↪transfunc import Tipper\n>>> tf = Tipper()\n>>> print(tf.to_string())\n| Hz | = |␣
↪Hz_Hx Hz_Hy | | Hx |\n                         | Hy |",
   "type": "object",
   "properties": {
```


(continues on next page)

(continued from previous page)

```
      "name": {
         "title": "Name",
         "type": "string"
      },
      "variation": {
         "title": "Variation",
         "default": "default",
         "maxLength": 16,
         "type": "string"
      },
      "out_chans": {
         "title": "Out Chans",
         "default": [
            "Hz"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "in_chans": {
         "title": "In Chans",
         "default": [
            "Hx",
            "Hy"
         ],
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "cross_chans": {
         "title": "Cross Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "n_out": {
         "title": "N Out",
         "type": "integer"
      },
      "n_in": {
         "title": "N In",
         "type": "integer"
      },
      "n_cross": {
         "title": "N Cross",
         "type": "integer"
      }
   }
}
```

field variation: ConstrainedStrValue = 'default'

A short additional bit of information about this variation

Constraints

- **maxLength** = 16

field out_chans: List[str] = ['Hz']

The output channels

field in_chans: List[str] = ['Hx', 'Hy']

The input channels

get_length(*components: Dict[str, Component]*) → ndarray

Get the tipper length

get_real_angle(*components: Dict[str, Component]*) → ndarray

Get the real angle

get_imag_angle(*components: Dict[str, Component]*) → ndarray

Get the imaginary angle

plot(*freqs: List[float]*, *components: Dict[str, Component]*, *x_lim: List[float] | None = None*, *len_lim: List[float] | None = None*, *ang_lim: List[float] | None = None*) → Figure

Plot the impedance tensor

> **Warning:** This probably needs further checking and verification

Parameters

- **freqs** (*List[float]*) – The x axis frequencies
- **components** (*Dict[str, Component]*) – The component data
- **x_lim** (*Optional[List[float]], optional*) – The x limits, to be provided as powers of 10, by default None. For example, for 0.001, use -3
- **len_lim** (*Optional[List[float]], optional*) – The y limits for tipper length, to be provided as powers of 10, by default None. For example, for 1000, use 3
- **ang_lim** (*Optional[List[float]], optional*) – The angle limits, by default None

Returns

Plotly figure

Return type

go.Figure

resistics.window module

Module for calculating window related data. Windows can be indexed relative to two starting indices.

- Local window index
  - Window index relative to the TimeData is called "local_win"
  - Local window indices always start at 0
- Global window index

  - The global window index is relative to the project reference time
  - The 0 index window begins at the reference time
  - This window indexing is to synchronise data across sites

The global window index is considered the default and sometimes referred to as the window. Local windows should be explicitly referred to as local_win in all cases.

The window module includes functionality to do the following:

- Windowing utility functions to calculate window and overlap sizes
- Functions to map windows to samples in TimeData
- Converting a global index array to datetime

Usually with windowing, there is a window size and windows overlap with each other for a set number of samples. As an illustrative examples, consider a signal sampled at 10 Hz (dt=0.1 seconds) with 24 samples. This will be windowed using a window size of 8 samples per window and a 2 sample overlap.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> fs = 10
>>> n_samples = 24
>>> win_size = 8
>>> olap_size = 2
>>> times = np.arange(0, n_samples) * (1/fs)
```

The first window

```
>>> start_win1 = 0
>>> end_win1 = win_size
>>> win1_times = times[start_win1:end_win1]
```

The second window

```
>>> start_win2 = end_win1 - olap_size
>>> end_win2 = start_win2 + win_size
>>> win2_times = times[start_win2:end_win2]
```

The third window

```
>>> start_win3 = end_win2 - olap_size
>>> end_win3 = start_win3 + win_size
>>> win3_times = times[start_win3:end_win3]
```

The fourth window

```
>>> start_win4= end_win3 - olap_size
>>> end_win4 = start_win4 + win_size
>>> win4_times = times[start_win4:end_win4]
```

Let's look at the actual window times for each window

```
>>> win1_times
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7])
>>> win2_times
```

(continues on next page)

(continued from previous page)

```
array([0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2, 1.3])
>>> win3_times
array([1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
>>> win4_times
array([1.8, 1.9, 2. , 2.1, 2.2, 2.3])
```

The duration and increments of windows can be calculated using provided methods

```
>>> from resistics.window import win_duration, inc_duration
>>> print(win_duration(win_size, fs))
0:00:00.7
>>> print(inc_duration(win_size, olap_size, fs))
0:00:00.6
```

Plot the windows to give an illustration of how it works

```
>>> plt.plot(win1_times, np.ones_like(win1_times), "bo", label="window1")
>>> plt.plot(win2_times, np.ones_like(win2_times)*2, "ro", label="window2")
>>> plt.plot(win3_times, np.ones_like(win3_times)*3, "go", label="window3")
>>> plt.plot(win4_times, np.ones_like(win4_times)*4, "co", label="window4")
>>> plt.xlabel("Time [s]")
>>> plt.legend()
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

resistics.window.**win_duration**(*win_size: int*, *fs: float*) → attotimedelta

Get the window duration

Parameters

- **win_size** (*int*) – Window size in samples
- **fs** (*float*) – Sampling frequency Hz

Returns

Duration

Return type

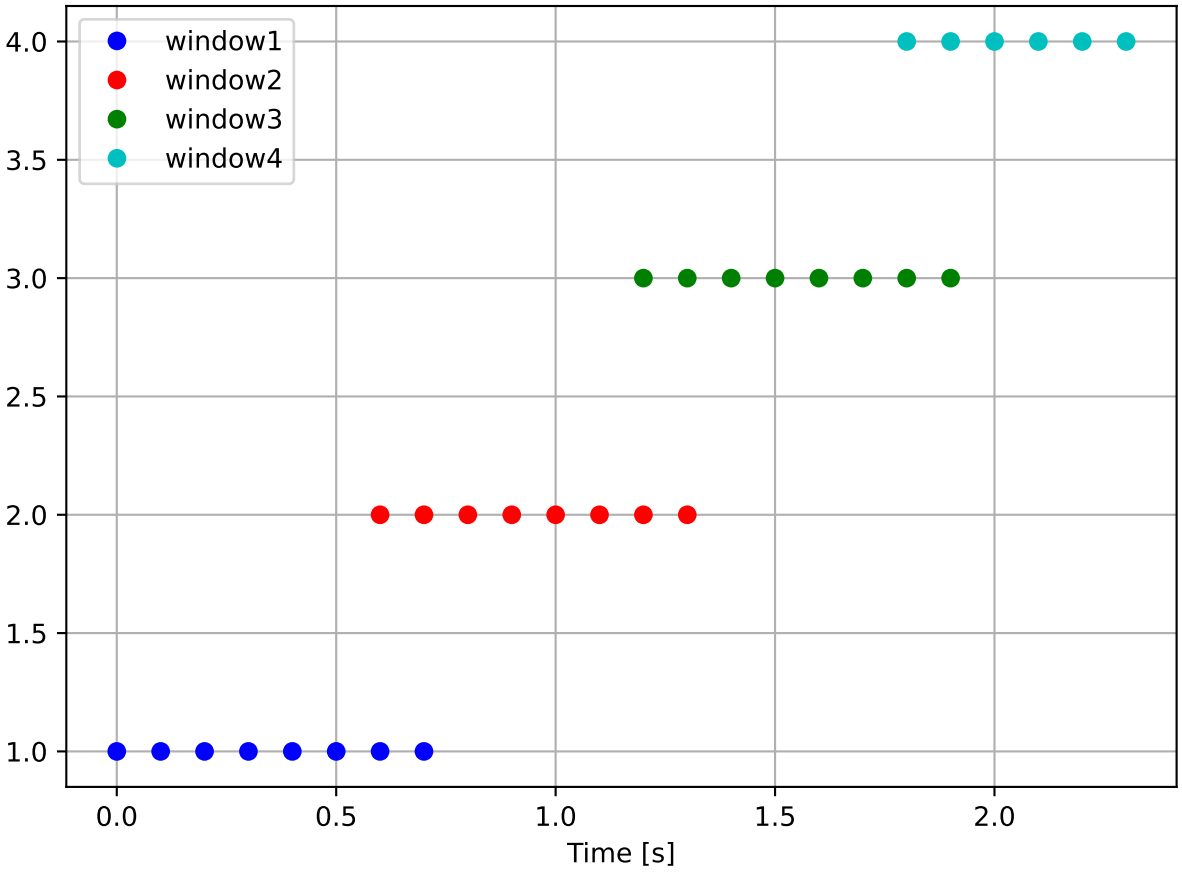
RSTimeDelta

Examples

A few examples with different sampling frequencies and window sizes

```
>>> from resistics.window import win_duration
>>> duration = win_duration(512, 512)
>>> print(duration)
0:00:00.998046875
>>> duration = win_duration(520, 512)
>>> print(duration)
0:00:01.013671875
>>> duration = win_duration(4096, 16_384)
>>> print(duration)
```

(continues on next page)


window1
window2
window3
window4
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.5
1.0
1.5
2.0
Time [s]

(continued from previous page)

```
0:00:00.24993896484375
>>> duration = win_duration(200, 0.05)
>>> print(duration)
1:06:20
```

resistics.window.**inc_duration**(*win_size: int*, *olap_size: int*, *fs: float*) → attotimedelta

Get the increment between window start times

If the overlap size = 0, then the time increment between windows is simply the window duration. However, when there is an overlap, the increment between window start times has to be adjusted by the overlap size

Parameters

- **win_size** (*int*) – The window size in samples
- **olap_size** (*int*) – The overlap size in samples
- **fs** (*float*) – The sample frequency Hz

Returns
The duration of the window

Return type
RSTimeDelta

Examples

```
>>> from resistics.window import inc_duration
>>> increment = inc_duration(128, 32, 128)
>>> print(increment)
0:00:00.75
>>> increment = inc_duration(128*3600, 128*60, 128)
>>> print(increment)
0:59:00
```

resistics.window.**win_to_datetime**(*ref_time: attodatetime*, *global_win: int*, *increment: attotimedelta*) → attodatetime

Convert reference window index to start time of window

Parameters

- **ref_time** (*RSDateTime*) – Reference time
- **global_win** (*int*) – Window index relative to reference time
- **increment** (*RSTimeDelta*) – The increment duration

Returns
Start time of window

Return type
RSDateTime

Examples

An example with sampling at 1 Hz, a window size of 100 and an overlap size of 25.

```
>>> from resistics.sampling import to_datetime
>>> from resistics.window import inc_duration, win_to_datetime
>>> ref_time = to_datetime("2021-01-01 00:00:00")
>>> fs = 1
>>> win_size = 60
>>> olap_size = 15
>>> increment = inc_duration(win_size, olap_size, fs)
>>> print(increment)
0:00:45
```

The increment is the time increment between the start of time one window and the succeeding window.

```
>>> print(win_to_datetime(ref_time, 0, increment))
2021-01-01 00:00:00
>>> print(win_to_datetime(ref_time, 1, increment))
2021-01-01 00:00:45
>>> print(win_to_datetime(ref_time, 2, increment))
2021-01-01 00:01:30
>>> print(win_to_datetime(ref_time, 3, increment))
2021-01-01 00:02:15
```

resistics.window.**datetime_to_win**(*ref_time: attodatetime*, *time: attodatetime*, *increment: attotimedelta*, *method: str = 'round'*) → int

Convert a datetime to a global window index

Parameters

- **ref_time** (*RSDateTime*) – Reference time
- **time** (*RSDateTime*) – Datetime to convert
- **increment** (*RSTimeDelta*) – The increment duration
- **method** (*str, optional*) – Method for dealing with float results, by default "round"

Returns

The global window index i.e. the window index relative to the reference time

Return type

int

Raises

ValueError – If time < ref_time

Examples

A simple example to show the logic

```
>>> from resistics.sampling import to_datetime, to_timedelta
>>> from resistics.window import datetime_to_win, win_to_datetime, inc_duration
>>> ref_time = to_datetime("2021-01-01 00:00:00")
>>> time = to_datetime("2021-01-01 00:01:00")
>>> increment = to_timedelta(60)
>>> global_win = datetime_to_win(ref_time, time, increment)
>>> global_win
1
>>> print(win_to_datetime(ref_time, global_win, increment))
2021-01-01 00:01:00
```

A more complex logic with window sizes, overlap sizes and sampling frequencies

```
>>> fs = 128
>>> win_size = 256
>>> olap_size = 64
>>> ref_time = to_datetime("2021-03-15 00:00:00")
>>> time = to_datetime("2021-04-17 18:00:00")
>>> increment = inc_duration(win_size, olap_size, fs)
>>> print(increment)
0:00:01.5
>>> global_win = datetime_to_win(ref_time, time, increment)
>>> global_win
1944000
>>> print(win_to_datetime(ref_time, global_win, increment))
2021-04-17 18:00:00
```

In this scenario, explore the use of rounding

```
>>> time = to_datetime("2021-04-17 18:00:00.50")
>>> global_win = datetime_to_win(ref_time, time, increment, method = "floor")
>>> global_win
1944000
>>> print(win_to_datetime(ref_time, global_win, increment))
2021-04-17 18:00:00
>>> global_win = datetime_to_win(ref_time, time, increment, method = "ceil")
>>> global_win
1944001
>>> print(win_to_datetime(ref_time, global_win, increment))
2021-04-17 18:00:01.5
>>> global_win = datetime_to_win(ref_time, time, increment, method = "round")
>>> global_win
1944000
>>> print(win_to_datetime(ref_time, global_win, increment))
2021-04-17 18:00:00
```

Another example with a window duration of greater than a day

```
>>> fs = 4.8828125e-05
>>> win_size = 64
```

(continues on next page)

(continued from previous page)

```
>>> olap_size = 16
>>> ref_time = to_datetime("1985-07-18 01:00:20")
>>> time = to_datetime("1985-09-22 23:00:00")
>>> increment = inc_duration(win_size, olap_size, fs)
>>> print(increment)
11 days, 9:04:00
>>> global_win = datetime_to_win(ref_time, time, increment)
>>> global_win
6
>>> print(win_to_datetime(ref_time, global_win, increment))
1985-09-24 07:24:20
```

This time is greater than the time that was transformed to global window, 1985-09-22 23:00:00. Try again, this time with the floor option.

```
>>> global_win = datetime_to_win(ref_time, time, increment, method="floor")
>>> global_win
5
>>> print(win_to_datetime(ref_time, global_win, increment))
1985-09-12 22:20:20
```

resistics.window.**get_first_and_last_win**(*ref_time: attodatetime*, *metadata:* DecimatedLevelMetadata, *win_size: int*, *olap_size: int*) → Tuple[int, int]

Get first and last window for a decimated data level

Note: For the last window, on initial calculation this may be one or a maximum of two windows beyond the last time. The last window is adjusted in this function.

Two windows may occur when the time of the last sample is in the overlap of the final two windows.

Parameters

- **ref_time** (*RSDateTime*) – The reference time
- **metadata** (DecimatedLevelMetadata) – Metadata for the decimation level
- **win_size** (*int*) – Window size in samples
- **olap_size** (*int*) – Overlap size in samples

Returns

First and last global windows. This is window indices relative to the reference time

Return type

Tuple[int, int]

Raises

ValueError – If unable to calculate the last window correctly as this will result in an incorrect number of windows

Examples

Get the first and last window for the first decimation level in a decimated data instance.

```
>>> from resistics.testing import decimated_data_random
>>> from resistics.sampling import to_datetime
>>> from resistics.window import get_first_and_last_win, win_to_datetime
>>> from resistics.window import win_duration, inc_duration
>>> ref_time = to_datetime("2021-01-01 00:00:00")
>>> dec_data = decimated_data_random(fs=0.1, first_time="2021-01-01 00:05:10", n_
↪samples=100, factor=10)
```

Get the metadata for decimation level 0

```
>>> level_metadata = dec_data.metadata.levels_metadata[0]
>>> level_metadata.summary()
{
    'fs': 10.0,
    'n_samples': 10000,
    'first_time': '2021-01-01 00:05:10.000000_000000_000000_000000',
    'last_time': '2021-01-01 00:21:49.899999_999999_977300_000000'
}
```

Note: As a point of interest, note how the last time is actually slightly incorrect. This is due to machine precision issues described in more detail here https://docs.python.org/3/tutorial/floatingpoint.html. Whilst there is value in using the high resolution datetime format for high sampling rates, there is a tradeoff. Such are the perils of floating point arithmetic.

The next step is to calculate the first and last window, relative to the reference time

```
>>> win_size = 100
>>> olap_size = 25
>>> first_win, last_win = get_first_and_last_win(ref_time, level_metadata, win_size,
↪ olap_size)
>>> print(first_win, last_win)
42 173
```

These window indices can be converted to start times of the windows. The last window is checked to make sure it does not extend past the end of the time data. First get the window duration and increments.

```
>>> duration = win_duration(win_size, level_metadata.fs)
>>> print(duration)
0:00:09.9
>>> increment = inc_duration(win_size, olap_size, level_metadata.fs)
>>> print(increment)
0:00:07.5
```

Now calculate the times of the windows

```
>>> first_win_start_time = win_to_datetime(ref_time, 42, increment)
>>> last_win_start_time = win_to_datetime(ref_time, 173, increment)
>>> print(first_win_start_time, last_win_start_time)
```

(continues on next page)

(continued from previous page)

```
2021-01-01 00:05:15 2021-01-01 00:21:37.5
>>> print(last_win_start_time + duration)
2021-01-01 00:21:47.4
>>> print(level_metadata.last_time)
2021-01-01 00:21:49.8999999999999773
>>> level_metadata.last_time > last_win_start_time + increment
True
```

resistics.window.**get_win_starts**(*ref_time: attodatetime*, *win_size: int*, *olap_size: int*, *fs: float*, *n_wins: int*, *index_offset: int*) → DatetimeIndex

Get window start times

This is a useful for getting the timestamps for the windows in a dataset

Parameters

- **ref_time** (*RSDateTime*) – The reference time
- **win_size** (*int*) – The window size
- **olap_size** (*int*) – The overlap size
- **fs** (*float*) – The sampling frequency
- **n_wins** (*int*) – The number of windows
- **index_offset** (*int*) – The index offset from the reference time

Returns
The start times of the windows

Return type
pd.DatetimeIndex

Examples

```
>>> import pandas as pd
>>> from resistics.sampling import to_datetime
>>> from resistics.window import get_win_starts
>>> ref_time = to_datetime("2021-01-01 00:00:00")
>>> win_size = 100
>>> olap_size = 25
>>> fs = 10
>>> n_wins = 3
>>> index_offset = 480
>>> starts = get_win_starts(ref_time, win_size, olap_size, fs, n_wins, index_offset)
>>> pd.Series(starts)
0   2021-01-01 01:00:00.000
1   2021-01-01 01:00:07.500
2   2021-01-01 01:00:15.000
dtype: datetime64[ns]
```

resistics.window.**get_win_ends**(*starts: DatetimeIndex*, *win_size: int*, *fs: float*) → DatetimeIndex

Get window end times

Parameters

- **starts** (*RSDateTime*) – The start times of the windows
- **win_size** (*int*) – The window size
- **fs** (*float*) – The sampling frequency

Returns
The end times of the windows

Return type
pd.DatetimeIndex

Examples

```
>>> import pandas as pd
>>> from resistics.sampling import to_datetime
>>> from resistics.window import get_win_starts, get_win_ends
>>> ref_time = to_datetime("2021-01-01 00:00:00")
>>> win_size = 100
>>> olap_size = 25
>>> fs = 10
>>> n_wins = 3
>>> index_offset = 480
>>> starts = get_win_starts(ref_time, win_size, olap_size, fs, n_wins, index_offset)
>>> pd.Series(starts)
0   2021-01-01 01:00:00.000
1   2021-01-01 01:00:07.500
2   2021-01-01 01:00:15.000
dtype: datetime64[ns]
>>> ends = get_win_ends(starts, win_size, fs)
>>> pd.Series(ends)
0   2021-01-01 01:00:09.900
1   2021-01-01 01:00:17.400
2   2021-01-01 01:00:24.900
dtype: datetime64[ns]
```

resistics.window.**get_win_table**(*ref_time: attodatetime*, *metadata:* DecimatedLevelMetadata, *win_size: int*, *olap_size: int*) → DataFrame

Get a DataFrame with

Parameters

- **ref_time** (*RSDateTime*) – Reference
- **metadata** (DecimatedLevelMetadata) – Metadata for the decimation level
- **win_size** (*int*) – The window size
- **olap_size** (*int*) – The overlap size

Returns
A pandas DataFrame with details about each window

Return type
pd.DataFrame

Examples

```
>>> import matplotlib.pyplot as plt
>>> from resistics.decimate import DecimatedLevelMetadata
>>> from resistics.sampling import to_datetime, to_timedelta
>>> from resistics.window import get_win_table
>>> ref_time = to_datetime("2021-01-01 00:00:00")
>>> fs = 10
>>> n_samples = 1000
>>> first_time = to_datetime("2021-01-01 01:00:00")
>>> last_time = first_time + to_timedelta((n_samples-1)/fs)
>>> metadata = DecimatedLevelMetadata(fs=10, n_samples=1000, first_time=first_time,␣
↪last_time=last_time)
>>> print(metadata.fs, metadata.first_time, metadata.last_time)
10.0 2021-01-01 01:00:00 2021-01-01 01:01:39.9
>>> win_size = 100
>>> olap_size = 25
>>> df = get_win_table(ref_time, metadata, win_size, olap_size)
>>> print(df.to_string())
    global  local  from_sample  to_sample               win_start                 ␣
↪win_end
0      480      0            0         99 2021-01-01 01:00:00.000 2021-01-01␣
↪01:00:09.900
1      481      1           75        174 2021-01-01 01:00:07.500 2021-01-01␣
↪01:00:17.400
2      482      2          150        249 2021-01-01 01:00:15.000 2021-01-01␣
↪01:00:24.900
3      483      3          225        324 2021-01-01 01:00:22.500 2021-01-01␣
↪01:00:32.400
4      484      4          300        399 2021-01-01 01:00:30.000 2021-01-01␣
↪01:00:39.900
5      485      5          375        474 2021-01-01 01:00:37.500 2021-01-01␣
↪01:00:47.400
6      486      6          450        549 2021-01-01 01:00:45.000 2021-01-01␣
↪01:00:54.900
7      487      7          525        624 2021-01-01 01:00:52.500 2021-01-01␣
↪01:01:02.400
8      488      8          600        699 2021-01-01 01:01:00.000 2021-01-01␣
↪01:01:09.900
9      489      9          675        774 2021-01-01 01:01:07.500 2021-01-01␣
↪01:01:17.400
10     490     10          750        849 2021-01-01 01:01:15.000 2021-01-01␣
↪01:01:24.900
11     491     11          825        924 2021-01-01 01:01:22.500 2021-01-01␣
↪01:01:32.400
12     492     12          900        999 2021-01-01 01:01:30.000 2021-01-01␣
↪01:01:39.900
```

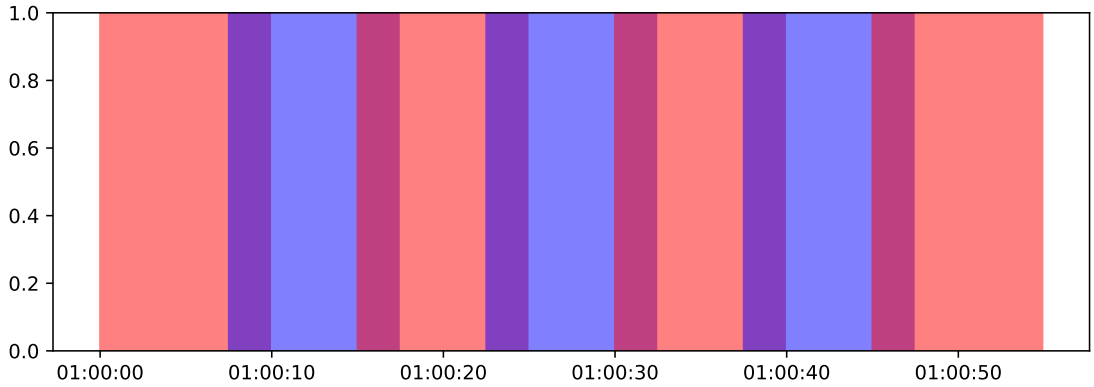
Plot six windows to illustrate the overlap

```
>>> plt.figure(figsize=(8, 3))
>>> for idx, row in df.iterrows():
...     color = "red" if idx%2 == 0 else "blue"
```

(continues on next page)

(continued from previous page)

```
...     plt.axvspan(row.loc["win_start"], row.loc["win_end"], alpha=0.5,␣
↪color=color)
...     if idx > 5:
...         break
>>> plt.tight_layout()
>>> plt.show()
```



pydantic model resistics.window.**WindowParameters**

Bases: *ResisticsModel*

Windowing parameters per decimation level

Windowing parameters are the window and overlap size for each decimation level.

Parameters

- **n_levels** (*int*) – The number of decimation levels
- **min_n_wins** (*int*) – Minimum number of windows
- **win_sizes** (*List[int]*) – The window sizes per decimation level
- **olap_sizes** (*List[int]*) – The overlap sizes per decimation level

Examples

Generate decimation and windowing parameters for data sampled at 4096 Hz. Note that requesting window sizes or overlap sizes for decimation levels that do not exist will raise a ValueError.

```
>>> from resistics.decimate import DecimationSetup
>>> from resistics.window import WindowSetup
>>> dec_setup = DecimationSetup(n_levels=3, per_level=3)
>>> dec_params = dec_setup.run(4096)
>>> dec_params.summary()
{
    'fs': 4096.0,
    'n_levels': 3,
    'per_level': 3,
    'min_samples': 256,
    'eval_freqs': [
```

(continues on next page)

(continued from previous page)

```
        1024.0,
        724.0773439350246,
        512.0,
        362.0386719675123,
        256.0,
        181.01933598375615,
        128.0,
        90.50966799187808,
        64.0
    ],
    'dec_factors': [1, 2, 8],
    'dec_increments': [1, 2, 4],
    'dec_fs': [4096.0, 2048.0, 512.0]
}
>>> win_params = WindowSetup().run(dec_params.n_levels, dec_params.dec_fs)
>>> win_params.summary()
{
    'n_levels': 3,
    'min_n_wins': 5,
    'win_sizes': [1024, 512, 128],
    'olap_sizes': [256, 128, 32]
}
>>> win_params.get_win_size(0)
1024
>>> win_params.get_olap_size(0)
256
>>> win_params.get_olap_size(3)
Traceback (most recent call last):
...
ValueError: Level 3 must be 0 <= level < 3
```

```
{
   "title": "WindowParameters",
   "description": "Windowing parameters per decimation level\n\nWindowing␣
↪parameters are the window and overlap size for each decimation\nlevel.\n\
↪nParameters\n----------\nn_levels : int\n    The number of decimation levels\nmin_
↪n_wins : int\n    Minimum number of windows\nwin_sizes : List[int]\n    The␣
↪window sizes per decimation level\nolap_sizes : List[int]\n    The overlap sizes␣
↪per decimation level\n\nExamples\n--------\nGenerate decimation and windowing␣
↪parameters for data sampled at 4096 Hz.\nNote that requesting window sizes or␣
↪overlap sizes for decimation levels\nthat do not exist will raise a ValueError.\n\
↪n>>> from resistics.decimate import DecimationSetup\n>>> from resistics.window␣
↪import WindowSetup\n>>> dec_setup = DecimationSetup(n_levels=3, per_level=3)\n>>>␣
↪dec_params = dec_setup.run(4096)\n>>> dec_params.summary()\n{\n    'fs': 4096.0,\
↪n    'n_levels': 3,\n    'per_level': 3,\n    'min_samples': 256,\n    'eval_freqs
↪': [\n        1024.0,\n        724.0773439350246,\n        512.0,\n        362.
↪0386719675123,\n        256.0,\n        181.01933598375615,\n        128.0,\n    ␣
↪    90.50966799187808,\n        64.0\n    ],\n    'dec_factors': [1, 2, 8],\n
↪'dec_increments': [1, 2, 4],\n    'dec_fs': [4096.0, 2048.0, 512.0]\n}\n>>> win_
↪params = WindowSetup().run(dec_params.n_levels, dec_params.dec_fs)\n>>> win_
↪params.summary()\n{\n    'n_levels': 3,\n    'min_n_wins': 5,\n    'win_sizes':␣
```

(continues on next page)

(continued from previous page)

```
↪[1024, 512, 128],\n    'olap_sizes': [256, 128, 32]\n}\n>>> win_params.get_win_
↪size(0)\n1024\n>>> win_params.get_olap_size(0)\n256\n>>> win_params.get_olap_
↪size(3)\nTraceback (most recent call last):\n...\nValueError: Level 3 must be 0
↪<= level < 3",
   "type": "object",
   "properties": {
      "n_levels": {
         "title": "N Levels",
         "type": "integer"
      },
      "min_n_wins": {
         "title": "Min N Wins",
         "type": "integer"
      },
      "win_sizes": {
         "title": "Win Sizes",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
      "olap_sizes": {
         "title": "Olap Sizes",
         "type": "array",
         "items": {
            "type": "integer"
         }
      }
   },
   "required": [
      "n_levels",
      "min_n_wins",
      "win_sizes",
      "olap_sizes"
   ]
}
```

field n_levels: `int` **[Required]**

field min_n_wins: `int` **[Required]**

field win_sizes: `List[int]` **[Required]**

field olap_sizes: `List[int]` **[Required]**

check_level(*level: int*)

Check the decimation level is within range

get_win_size(*level: int*) → int

Get window size for a decimation level

get_olap_size(*level: int*) → int

Get overlap size for a decimation level

pydantic model resistics.window.**WindowSetup**

Bases: *ResisticsProcess*

Setup WindowParameters

WindowSetup outputs the WindowParameters to use for windowing decimated time data.

Window parameters are simply the window and overlap sizes for each decimation level.

Parameters

- **min_size** (*int, optional*) – Minimum window size, by default 128
- **min_olap** (*int, optional*) – Minimum overlap size, by default 32
- **win_factor** (*int, optional*) – Window factor, by default 4. Window sizes are calculated by sampling frequency / 4 to ensure sufficient frequency resolution. If the sampling frequency is small, window size will be adjusted to min_size
- **olap_proportion** (*float, optional*) – The proportion of the window size to use as the overlap, by default 0.25. For example, for a window size of 128, the overlap would be 0.25 * 128 = 32
- **min_n_wins** (*int, optional*) – The minimum number of windows needed in a decimation level, by default 5
- **win_sizes** (*Optional[List[int]], optional*) – Explicit define window sizes, by default None. Must have the same length as number of decimation levels
- **olap_sizes** (*Optional[List[int]], optional*) – Explicitly define overlap sizes, by default None. Must have the same length as number of decimation levels

Examples

Generate decimation and windowing parameters for data sampled at 0.05 Hz or 20 seconds sampling period

```
>>> from resistics.decimate import DecimationSetup
>>> from resistics.window import WindowSetup
>>> dec_params = DecimationSetup(n_levels=3, per_level=3).run(0.05)
>>> dec_params.summary()
{
    'fs': 0.05,
    'n_levels': 3,
    'per_level': 3,
    'min_samples': 256,
    'eval_freqs': [
        0.0125,
        0.008838834764831844,
        0.00625,
        0.004419417382415922,
        0.003125,
        0.002209708691207961,
        0.0015625,
        0.0011048543456039805,
        0.00078125
    ],
    'dec_factors': [1, 2, 8],
    'dec_increments': [1, 2, 4],
```

(continues on next page)

(continued from previous page)

```
    'dec_fs': [0.05, 0.025, 0.00625]
}
>>> win_params = WindowSetup().run(dec_params.n_levels, dec_params.dec_fs)
>>> win_params.summary()
{
    'n_levels': 3,
    'min_n_wins': 5,
    'win_sizes': [128, 128, 128],
    'olap_sizes': [32, 32, 32]
}
```

Window parameters can also be explicitly defined

```
>>> from resistics.decimate import DecimationSetup
>>> from resistics.window import WindowSetup
>>> dec_setup = DecimationSetup(n_levels=3, per_level=3)
>>> dec_params = dec_setup.run(0.05)
>>> win_setup = WindowSetup(win_sizes=[1000, 578, 104])
>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)
>>> win_params.summary()
{
    'n_levels': 3,
    'min_n_wins': 5,
    'win_sizes': [1000, 578, 104],
    'olap_sizes': [250, 144, 32]
}
```

When providing explicit window and overlap sizes, the size of the overlap is expected to be less than the size of the window

```
>>> from resistics.decimate import DecimationSetup
>>> from resistics.window import WindowSetup
>>> dec_setup = DecimationSetup(n_levels=3, per_level=3)
>>> dec_params = dec_setup.run(0.05)
>>> win_setup = WindowSetup(win_sizes=[1000, 578, 104], olap_sizes=[1001, 600, 25])
>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)
Traceback (most recent call last):
...
ValueError: Invalid overlaps found [ True  True False]
```

```
{
   "title": "WindowSetup",
   "description": "Setup WindowParameters\n\nWindowSetup outputs the␣
↪WindowParameters to use for windowing decimated\ntime data.\n\nWindow parameters␣
↪are simply the window and overlap sizes for each\ndecimation level.\n\nParameters\
↪n----------\nmin_size : int, optional\n    Minimum window size, by default 128\
↪nmin_olap : int, optional\n    Minimum overlap size, by default 32\nwin_factor :␣
↪int, optional\n    Window factor, by default 4. Window sizes are calculated by␣
↪sampling\n    frequency / 4 to ensure sufficient frequency resolution. If the\n  ␣
↪ sampling frequency is small, window size will be adjusted to\n    min_size\nolap_
↪proportion : float, optional\n    The proportion of the window size to use as the␣
↪overlap, by default\n    0.25. For example, for a window size of 128, the overlap␣
```

(continues on next page)

(continued from previous page)


```
↪would be\n    0.25 * 128 = 32\nmin_n_wins : int, optional\n    The minimum number␣
↪of windows needed in a decimation level, by\n    default 5\nwin_sizes :␣
↪Optional[List[int]], optional\n    Explicit define window sizes, by default None.␣
↪Must have the same\n    length as number of decimation levels\nolap_sizes :␣
↪Optional[List[int]], optional\n    Explicitly define overlap sizes, by default␣
↪None. Must have the same\n    length as number of decimation levels\n\nExamples\n-
↪-------\nGenerate decimation and windowing parameters for data sampled at 0.05 Hz␣
↪or\n20 seconds sampling period\n\n>>> from resistics.decimate import␣
↪DecimationSetup\n>>> from resistics.window import WindowSetup\n>>> dec_params =␣
↪DecimationSetup(n_levels=3, per_level=3).run(0.05)\n>>> dec_params.summary()\n{\n␣
↪   'fs': 0.05,\n    'n_levels': 3,\n    'per_level': 3,\n    'min_samples': 256,\
↪n    'eval_freqs': [\n        0.0125,\n        0.008838834764831844,\n        0.
↪00625,\n        0.004419417382415922,\n        0.003125,\n        0.
↪002209708691207961,\n        0.0015625,\n        0.0011048543456039805,\n       ␣
↪0.00078125\n    ],\n    'dec_factors': [1, 2, 8],\n    'dec_increments': [1, 2,␣
↪4],\n    'dec_fs': [0.05, 0.025, 0.00625]\n}\n>>> win_params = WindowSetup().
↪run(dec_params.n_levels, dec_params.dec_fs)\n>>> win_params.summary()\n{\n    'n_
↪levels': 3,\n    'min_n_wins': 5,\n    'win_sizes': [128, 128, 128],\n    'olap_
↪sizes': [32, 32, 32]\n}\n\nWindow parameters can also be explicitly defined\n\n>>>
↪ from resistics.decimate import DecimationSetup\n>>> from resistics.window import␣
↪WindowSetup\n>>> dec_setup = DecimationSetup(n_levels=3, per_level=3)\n>>> dec_
↪params = dec_setup.run(0.05)\n>>> win_setup = WindowSetup(win_sizes=[1000, 578,␣
↪104])\n>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)\n>>>
↪ win_params.summary()\n{\n    'n_levels': 3,\n    'min_n_wins': 5,\n    'win_sizes
↪': [1000, 578, 104],\n    'olap_sizes': [250, 144, 32]\n}\n\nWhen providing␣
↪explicit window and overlap sizes, the size of the overlap\nis expected to be␣
↪less than the size of the window\n\n>>> from resistics.decimate import␣
↪DecimationSetup\n>>> from resistics.window import WindowSetup\n>>> dec_setup =␣
↪DecimationSetup(n_levels=3, per_level=3)\n>>> dec_params = dec_setup.run(0.05)\n>>
↪> win_setup = WindowSetup(win_sizes=[1000, 578, 104], olap_sizes=[1001, 600, 25])\
↪n>>> win_params = win_setup.run(dec_params.n_levels, dec_params.dec_fs)\
↪nTraceback (most recent call last):\n...\nValueError: Invalid overlaps found [␣
↪True  True False]",
  "type": "object",
  "properties": {
    "name": {
      "title": "Name",
      "type": "string"
    },
    "min_size": {
      "title": "Min Size",
      "default": 128,
      "type": "integer"
    },
    "min_olap": {
      "title": "Min Olap",
      "default": 32,
      "type": "integer"
    },
    "win_factor": {
      "title": "Win Factor",
      "default": 4,
```


(continues on next page)

(continued from previous page)

```
            "type": "integer"
        },
        "olap_proportion": {
            "title": "Olap Proportion",
            "default": 0.25,
            "type": "number"
        },
        "min_n_wins": {
            "title": "Min N Wins",
            "default": 5,
            "type": "integer"
        },
        "win_sizes": {
            "title": "Win Sizes",
            "type": "array",
            "items": {
                "type": "integer"
            }
        },
        "olap_sizes": {
            "title": "Olap Sizes",
            "type": "array",
            "items": {
                "type": "integer"
            }
        }
    }
}
```

field min_size: *int* = 128

field min_olap: *int* = 32

field win_factor: *int* = 4

field olap_proportion: *float* = 0.25

field min_n_wins: *int* = 5

field win_sizes: *List[int]* | *None* = None

field olap_sizes: *List[int]* | *None* = None

run(*n_levels: int*, *dec_fs: List[float]*) → *WindowParameters*

Calculate window and overlap sizes for each decimation level based on decimation level sampling frequency and minimum allowable parameters

The window and overlap sizes (number of samples) are calculated based in the following way:

- window size = frequency at decimation level / window factor
- overlap size = window size * overlap proportion

This is to ensure good frequency resolution at high frequencies. At low sampling frequencies, this would result in very small window sizes, therefore, there a minimum allowable sizes for both windows and overlap defined by min_size and min_olap in the initialiser. If window sizes or overlaps size are calculated below these respecitively, they will be set to the minimum values.

Parameters

- **n_levels** (*int*) – The number of decimation levels
- **dec_fs** (*List[float]*) – The sampling frequencies for each decimation level

Returns
The window parameters, the window sizes and overlaps for each decimation level

Return type
WindowParameters

Raises

- **ValueError** – If the number of windows does not match the number of levels
- **ValueError** – If the number of overlaps does not match the number of levels
- **ValueError** – If any of the overlaps are bigger than the window sizes

pydantic model resistics.window.**WindowedLevelMetadata**

Bases: *Metadata*

Metadata for a windowed level


```
{
   "title": "WindowedLevelMetadata",
   "description": "Metadata for a windowed level",
   "type": "object",
   "properties": {
      "fs": {
         "title": "Fs",
         "type": "number"
      },
      "n_wins": {
         "title": "N Wins",
         "type": "integer"
      },
      "win_size": {
         "title": "Win Size",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "olap_size": {
         "title": "Olap Size",
         "exclusiveMinimum": 0,
         "type": "integer"
      },
      "index_offset": {
         "title": "Index Offset",
         "type": "integer"
      }
   },
   "required": [
      "fs",
      "n_wins",
      "win_size",
      "olap_size",
```


(continues on next page)

(continued from previous page)

```
      "index_offset"
   ]
}
```

field fs: *float* **[Required]**

The sampling frequency for the decimation level

field n_wins: *int* **[Required]**

The number of windows

field win_size: *PositiveInt* **[Required]**

The window size in samples

Constraints

- **exclusiveMinimum** = 0

field olap_size: *PositiveInt* **[Required]**

The overlap size in samples

Constraints

- **exclusiveMinimum** = 0

field index_offset: *int* **[Required]**

The global window offset for local window 0

property dt

pydantic model resistics.window.**WindowedMetadata**

Bases: *WriteableMetadata*

Metadata for windowed data

```
{
   "title": "WindowedMetadata",
   "description": "Metadata for windowed data",
   "type": "object",
   "properties": {
      "file_info": {
         "$ref": "#/definitions/ResisticsFile"
      },
      "fs": {
         "title": "Fs",
         "type": "array",
         "items": {
            "type": "number"
         }
      },
      "chans": {
         "title": "Chans",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
```

(continues on next page)

(continued from previous page)


```
    "n_chans": {
       "title": "N Chans",
       "type": "integer"
    },
    "n_levels": {
       "title": "N Levels",
       "type": "integer"
    },
    "first_time": {
       "title": "First Time",
       "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
       "examples": [
          "2021-01-01 00:00:00.000061_035156_250000_000000"
       ]
    },
    "last_time": {
       "title": "Last Time",
       "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
       "examples": [
          "2021-01-01 00:00:00.000061_035156_250000_000000"
       ]
    },
    "system": {
       "title": "System",
       "default": "",
       "type": "string"
    },
    "serial": {
       "title": "Serial",
       "default": "",
       "type": "string"
    },
    "wgs84_latitude": {
       "title": "Wgs84 Latitude",
       "default": -999.0,
       "type": "number"
    },
    "wgs84_longitude": {
       "title": "Wgs84 Longitude",
       "default": -999.0,
       "type": "number"
    },
    "easting": {
       "title": "Easting",
       "default": -999.0,
       "type": "number"
    },
    "northing": {
       "title": "Northing",
       "default": -999.0,
       "type": "number"
    },
```


(continues on next page)

(continued from previous page)

```
      "elevation": {
         "title": "Elevation",
         "default": -999.0,
         "type": "number"
      },
      "chans_metadata": {
         "title": "Chans Metadata",
         "type": "object",
         "additionalProperties": {
            "$ref": "#/definitions/ChanMetadata"
         }
      },
      "levels_metadata": {
         "title": "Levels Metadata",
         "type": "array",
         "items": {
            "$ref": "#/definitions/WindowedLevelMetadata"
         }
      },
      "ref_time": {
         "title": "Ref Time",
         "pattern": "%Y-%m-%d %H:%M:%S.%f_%o_%q_%v",
         "examples": [
            "2021-01-01 00:00:00.000061_035156_250000_000000"
         ]
      },
      "history": {
         "title": "History",
         "default": {
            "records": []
         },
         "allOf": [
            {
               "$ref": "#/definitions/History"
            }
         ]
      }
   },
   "required": [
      "fs",
      "chans",
      "n_levels",
      "first_time",
      "last_time",
      "chans_metadata",
      "levels_metadata",
      "ref_time"
   ],
   "definitions": {
      "ResisticsFile": {
         "title": "ResisticsFile",
         "description": "Required information for writing out a resistics file",
```

(continues on next page)

(continued from previous page)

```
            "type": "object",
            "properties": {
               "created_on_local": {
                  "title": "Created On Local",
                  "type": "string",
                  "format": "date-time"
               },
               "created_on_utc": {
                  "title": "Created On Utc",
                  "type": "string",
                  "format": "date-time"
               },
               "version": {
                  "title": "Version",
                  "type": "string"
               }
            }
         },
         "ChanMetadata": {
            "title": "ChanMetadata",
            "description": "Channel metadata",
            "type": "object",
            "properties": {
               "name": {
                  "title": "Name",
                  "type": "string"
               },
               "data_files": {
                  "title": "Data Files",
                  "type": "array",
                  "items": {
                     "type": "string"
                  }
               },
               "chan_type": {
                  "title": "Chan Type",
                  "type": "string"
               },
               "chan_source": {
                  "title": "Chan Source",
                  "type": "string"
               },
               "sensor": {
                  "title": "Sensor",
                  "default": "",
                  "type": "string"
               },
               "serial": {
                  "title": "Serial",
                  "default": "",
                  "type": "string"
               },
```

(continues on next page)

(continued from previous page)

```
            "gain1": {
               "title": "Gain1",
               "default": 1,
               "type": "number"
            },
            "gain2": {
               "title": "Gain2",
               "default": 1,
               "type": "number"
            },
            "scaling": {
               "title": "Scaling",
               "default": 1,
               "type": "number"
            },
            "chopper": {
               "title": "Chopper",
               "default": false,
               "type": "boolean"
            },
            "dipole_dist": {
               "title": "Dipole Dist",
               "default": 1,
               "type": "number"
            },
            "sensor_calibration_file": {
               "title": "Sensor Calibration File",
               "default": "",
               "type": "string"
            },
            "instrument_calibration_file": {
               "title": "Instrument Calibration File",
               "default": "",
               "type": "string"
            }
         },
         "required": [
            "name"
         ]
      },
      "WindowedLevelMetadata": {
         "title": "WindowedLevelMetadata",
         "description": "Metadata for a windowed level",
         "type": "object",
         "properties": {
            "fs": {
               "title": "Fs",
               "type": "number"
            },
            "n_wins": {
               "title": "N Wins",
               "type": "integer"
```

(continues on next page)

(continued from previous page)


```
            },
            "win_size": {
               "title": "Win Size",
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "olap_size": {
               "title": "Olap Size",
               "exclusiveMinimum": 0,
               "type": "integer"
            },
            "index_offset": {
               "title": "Index Offset",
               "type": "integer"
            }
         },
         "required": [
            "fs",
            "n_wins",
            "win_size",
            "olap_size",
            "index_offset"
         ]
      },
      "Record": {
         "title": "Record",
         "description": "Class to hold a record\n\nA record holds information about␣
↪a process that was run. It is intended to\ntrack processes applied to data,␣
↪allowing a process history to be saved\nalong with any datasets.\n\nExamples\n----
↪----\nA simple example of creating a process record\n\n>>> from resistics.common␣
↪import Record\n>>> messages = [\"message 1\", \"message 2\"]\n>>> record =␣
↪Record(\n...     creator={\"name\": \"example\", \"parameter1\": 15},\n...     ␣
↪messages=messages,\n...     record_type=\"example\"\n... )\n>>> record.summary()\n
↪{\n    'time_local': '...',\n    'time_utc': '...',\n    'creator': {'name':
↪'example', 'parameter1': 15},\n    'messages': ['message 1', 'message 2'],\n
↪'record_type': 'example'\n}",
         "type": "object",
         "properties": {
            "time_local": {
               "title": "Time Local",
               "type": "string",
               "format": "date-time"
            },
            "time_utc": {
               "title": "Time Utc",
               "type": "string",
               "format": "date-time"
            },
            "creator": {
               "title": "Creator",
               "type": "object"
            },
```


(continues on next page)

(continued from previous page)


```
            "messages": {
               "title": "Messages",
               "type": "array",
               "items": {
                  "type": "string"
               }
            },
            "record_type": {
               "title": "Record Type",
               "type": "string"
            }
         },
         "required": [
            "creator",
            "messages",
            "record_type"
         ]
      },
      "History": {
         "title": "History",
         "description": "Class for storing processing history\n\nParameters\n-------
↪---\nrecords : List[Record], optional\n    List of records, by default []\n\
↪nExamples\n--------\n>>> from resistics.testing import record_example1, record_
↪example2\n>>> from resistics.common import History\n>>> record1 = record_
↪example1()\n>>> record2 = record_example2()\n>>> history =␣
↪History(records=[record1, record2])\n>>> history.summary()\n{\n    'records': [\n␣
↪       {\n            'time_local': '...',\n            'time_utc': '...',\n     ␣
↪      'creator': {\n                'name': 'example1',\n                'a': 5,\
↪n                'b': -7.0\n            },\n            'messages': ['Message 1',
↪'Message 2'],\n            'record_type': 'process'\n        },\n        {\n     ␣
↪      'time_local': '...',\n            'time_utc': '...',\n            'creator
↪': {\n                'name': 'example2',\n                'a': 'parzen',\n      ␣
↪         'b': -21\n            },\n            'messages': ['Message 5', 'Message␣
↪6'],\n            'record_type': 'process'\n        }\n    ]\n}",
         "type": "object",
         "properties": {
            "records": {
               "title": "Records",
               "default": [],
               "type": "array",
               "items": {
                  "$ref": "#/definitions/Record"
               }
            }
         }
      }
   }
}
```


field fs: List[float] [Required]

field chans: List[str] [Required]

field n_chans: int | None = None

Validated by

- validate_n_chans

field n_levels: int [Required]

field first_time: HighResDateTime [Required]

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field last_time: HighResDateTime [Required]

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field system: str = ''

field serial: str = ''

field wgs84_latitude: float = -999.0

field wgs84_longitude: float = -999.0

field easting: float = -999.0

field northing: float = -999.0

field elevation: float = -999.0

field chans_metadata: Dict[str, ChanMetadata] [Required]

field levels_metadata: List[WindowedLevelMetadata] [Required]

field ref_time: HighResDateTime [Required]

Constraints

- **pattern** = %Y-%m-%d %H:%M:%S.%f_%o_%q_%v
- **examples** = ['2021-01-01 00:00:00.000061_035156_250000_000000']

field history: History = History(records=[])

class resistics.window.**WindowedData**(*metadata:* WindowedMetadata, *data: Dict[int, ndarray]*)

Bases: ResisticsData

Windows of a DecimatedData object

The windowed data is stored in a dictionary attribute named data. This is a dictionary with an entry for each decimation level. The shape for a single decimation level is as follows:

n_wins x n_chans x n_samples

get_level(*level: int*) → ndarray

Get windows for a decimation level

Parameters
level (*int*) – The decimation level

Returns
The window array

Return type
np.ndarray

Raises
ValueError – If decimation level is not within range

get_local(*level: int*, *local_win: int*) → ndarray

Get window using local index

Parameters

- **level** (*int*) – The decimation level
- **local_win** (*int*) – Local window index

Returns
Window data

Return type
np.ndarray

Raises
ValueError – If local window index is out of range

get_global(*level: int*, *global_win: int*) → ndarray

Get window using global index

Parameters

- **level** (*int*) – The decimation level
- **global_win** (*int*) – Global window index

Returns
Window data

Return type
np.ndarray

get_chan(*level: int*, *chan: str*) → ndarray

Get all the windows for a channel

Parameters

- **level** (*int*) – The decimation level
- **chan** (*str*) – The channel

Returns
The data for the channels

Return type
np.ndarray

Raises
ChannelNotFoundError – If the channel is not found in the data

to_string() → str

Class information as a string

pydantic model resistics.window.**Windower**

Bases: *ResisticsProcess*

Windows DecimatedData

This is the primary window making process for resistics and should be used when alignment of windows with a site or across sites is required.

This method uses numpy striding to produce window views into the decimated data.

See also:

WindowerTarget
A windower to make a target number of windows

Examples

The Windower windows a DecimatedData object given a reference time and some window parameters.

There's quite a few imports needed for this example. Begin by doing the imports, defining a reference time and generating random decimated data.

```
>>> from resistics.sampling import to_datetime
>>> from resistics.testing import decimated_data_linear
>>> from resistics.window import WindowSetup, Windower
>>> dec_data = decimated_data_linear(fs=128)
>>> ref_time = dec_data.metadata.first_time
>>> print(dec_data.to_string())
<class 'resistics.decimate.DecimatedData'>
           fs        dt  n_samples           first_time                         last_
↪time
level
0      2048.0  0.000488      16384  2021-01-01 00:00:00  2021-01-01 00:00:07.
↪99951171875
1       512.0  0.001953       4096  2021-01-01 00:00:00    2021-01-01 00:00:07.
↪998046875
2       128.0  0.007812       1024  2021-01-01 00:00:00      2021-01-01 00:00:07.
↪9921875
```

Next, initialise the window parameters. For this example, use small windows, which will make inspecting them easier.

```
>>> win_params = WindowSetup(win_sizes=[16,16,16], min_olap=4).run(dec_data.
↪metadata.n_levels, dec_data.metadata.fs)
>>> win_params.summary()
{
    'n_levels': 3,
    'min_n_wins': 5,
    'win_sizes': [16, 16, 16],
    'olap_sizes': [4, 4, 4]
}
```

Perform the windowing. This actually creates views into the decimated data using the numpy.lib.stride_tricks.sliding_window_view function. The shape for a data array at a decimation level is: n_wins x n_chans x win_size. The information about each level is also in the levels_metadata attribute of WindowedMetadata.

```
>>> win_data = Windower().run(ref_time, win_params, dec_data)
>>> win_data.data[0].shape
(1365, 2, 16)
>>> for level_metadata in win_data.metadata.levels_metadata:
...     level_metadata.summary()
{
    'fs': 2048.0,
    'n_wins': 1365,
    'win_size': 16,
    'olap_size': 4,
    'index_offset': 0
}
{
    'fs': 512.0,
    'n_wins': 341,
    'win_size': 16,
    'olap_size': 4,
    'index_offset': 0
}
{
    'fs': 128.0,
    'n_wins': 85,
    'win_size': 16,
    'olap_size': 4,
    'index_offset': 0
}
```

Let's look at an example of data from the first decimation level for the first channel. This is simply a linear set of data ranging from 0...16_383.

```
>>> dec_data.data[0][0]
array([    0,     1,     2, ..., 16381, 16382, 16383])
```

Inspecting the first few windows shows they are as expected including the overlap.

```
>>> win_data.data[0][0, 0]
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> win_data.data[0][1, 0]
array([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27])
>>> win_data.data[0][2, 0]
array([24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39])
```

```
{
   "title": "Windower",
   "description": "Windows DecimatedData\n\nThis is the primary window making␣
↪process for resistics and should be used\nwhen alignment of windows with a site␣
↪or across sites is required.\n\nThis method uses numpy striding to produce window␣
↪views into the decimated\ndata.\n\nSee Also\n--------\nWindowerTarget : A␣
```

(continues on next page)

(continued from previous page)

```
↪windower to make a target number of windows\n\nExamples\n--------\nThe Windower␣
↪windows a DecimatedData object given a reference time and some\nwindow parameters.
↪\n\nThere's quite a few imports needed for this example. Begin by doing the\
↪nimports, defining a reference time and generating random decimated data.\n\n>>>␣
↪from resistics.sampling import to_datetime\n>>> from resistics.testing import␣
↪decimated_data_linear\n>>> from resistics.window import WindowSetup, Windower\n>>>
↪ dec_data = decimated_data_linear(fs=128)\n>>> ref_time = dec_data.metadata.first_
↪time\n>>> print(dec_data.to_string())\n<class 'resistics.decimate.DecimatedData'>\
↪n           fs        dt  n_samples           first_time                  ␣
↪last_time\nlevel\n0      2048.0  0.000488      16384  2021-01-01 00:00:00  2021-
↪01-01 00:00:07.99951171875\n1       512.0  0.001953       4096  2021-01-01␣
↪00:00:00    2021-01-01 00:00:07.998046875\n2       128.0  0.007812       1024 ␣
↪2021-01-01 00:00:00      2021-01-01 00:00:07.9921875\n\nNext, initialise the␣
↪window parameters. For this example, use small windows,\nwhich will make␣
↪inspecting them easier.\n\n>>> win_params = WindowSetup(win_sizes=[16,16,16], min_
↪olap=4).run(dec_data.metadata.n_levels, dec_data.metadata.fs)\n>>> win_params.
↪summary()\n{\n    'n_levels': 3,\n    'min_n_wins': 5,\n    'win_sizes': [16, 16,␣
↪16],\n    'olap_sizes': [4, 4, 4]\n}\n\nPerform the windowing. This actually␣
↪creates views into the decimated data\nusing the numpy.lib.stride_tricks.sliding_
↪window_view function. The shape\nfor a data array at a decimation level is: n_
↪wins x n_chans x win_size. The\ninformation about each level is also in the␣
↪levels_metadata attribute of\nWindowedMetadata.\n\n>>> win_data = Windower().
↪run(ref_time, win_params, dec_data)\n>>> win_data.data[0].shape\n(1365, 2, 16)\n>>
↪> for level_metadata in win_data.metadata.levels_metadata:\n...     level_
↪metadata.summary()\n{\n    'fs': 2048.0,\n    'n_wins': 1365,\n    'win_size': 16,
↪\n    'olap_size': 4,\n    'index_offset': 0\n}\n{\n    'fs': 512.0,\n    'n_wins
↪': 341,\n    'win_size': 16,\n    'olap_size': 4,\n    'index_offset': 0\n}\n{\n ␣
↪  'fs': 128.0,\n    'n_wins': 85,\n    'win_size': 16,\n    'olap_size': 4,\n
↪'index_offset': 0\n}\n\nLet's look at an example of data from the first␣
↪decimation level for the\nfirst channel. This is simply a linear set of data␣
↪ranging from 0...16_383.\n\n>>> dec_data.data[0][0]\narray([    0,     1,     2, .
↪.., 16381, 16382, 16383])\n\nInspecting the first few windows shows they are as␣
↪expected including the\noverlap.\n\n>>> win_data.data[0][0, 0]\narray([ 0,  1,  2,
↪  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])\n>>> win_data.data[0][1, 0]\
↪narray([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27])\n>>> win_
↪data.data[0][2, 0]\narray([24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
↪ 38, 39])",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```

run(*ref_time: attodatetime*, *win_params:* WindowParameters, *dec_data:* DecimatedData) → *WindowedData*

Perform windowing of DecimatedData

Parameters

- **ref_time** (*RSDateTime*) – The reference time
- **win_params** (WindowParameters) – The window parameters

- **dec_data** (DecimatedData) – The decimated data

Returns
Windows for decimated data

Return type
WindowedData

Raises
ProcessRunError – If the number of windows calculated in the window table does not match the size of the array views

pydantic model resistics.window.**WindowerTarget**

Bases: *Windower*

Windower that selects window sizes to meet a target number of windows

The minimum window size in window parameters will be respected even if the generated number of windows is below the target. This is to avoid situations where excessively small windows sizes are selected.

Warning: This process is primarily useful for quick processing of a single measurement and should not be used when any alignment of windows is required within a site or across sites.

Parameters

- **target** (*int*) – The target number of windows for each decimation level
- **olap_proportion** (*float*) – The overlap proportion of the window size

See also:

Windower
The window making process to use when alignment is required


```
{
   "title": "WindowerTarget",
   "description": "Windower that selects window sizes to meet a target number of␣
↪windows\n\nThe minimum window size in window parameters will be respected even if␣
↪the\ngenerated number of windows is below the target. This is to avoid situations\
↪nwhere excessively small windows sizes are selected.\n\n.. warning::\n\n    This␣
↪process is primarily useful for quick processing of a single\n    measurement and␣
↪should not be used when any alignment of windows is\n    required within a site␣
↪or across sites.\n\nParameters\n----------\ntarget : int\n    The target number␣
↪of windows for each decimation level\nolap_proportion : float\n    The overlap␣
↪proportion of the window size\n\nSee Also\n--------\nWindower : The window making␣
↪process to use when alignment is required",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "target": {
         "title": "Target",
         "default": 1000,
```


(continues on next page)

(continued from previous page)


```
            "type": "integer"
        },
        "min_size": {
            "title": "Min Size",
            "default": 64,
            "type": "integer"
        },
        "olap_proportion": {
            "title": "Olap Proportion",
            "default": 0.25,
            "type": "number"
        }
    }
}
```


field target: int = 1000

field min_size: int = 64

field olap_proportion: float = 0.25

run(*ref_time: attodatetime*, *win_params:* WindowParameters, *dec_data:* DecimatedData) → *WindowedData*

Perform windowing of DecimatedData

Parameters

- **ref_time** (*RSDateTime*) – The reference time
- **win_params** (WindowParameters) – The window parameters
- **dec_data** (DecimatedData) – The decimated data

Returns

Windows for decimated data

Return type

WindowedData

Raises

ProcessRunError – If the number of windows calculated in the window table does not match the size of the array views

pydantic model resistics.window.**WindowedDataWriter**

Bases: *ResisticsWriter*

Writer of resistics windowed data


```
{
   "title": "WindowedDataWriter",
   "description": "Writer of resistics windowed data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "overwrite": {
```


(continues on next page)

(continued from previous page)

```
            "title": "Overwrite",
            "default": true,
            "type": "boolean"
        }
    }
}
```

run(*dir_path: Path*, *win_data:* WindowedData) → None

Write out WindowedData

Parameters

- **dir_path** (*Path*) – The directory path to write to
- **win_data** (WindowedData) – Windowed data to write out

Raises

WriteError – If unable to write to the directory

pydantic model resistics.window.**WindowedDataReader**

Bases: *ResisticsProcess*

Reader of resistics windowed data

```
{
   "title": "WindowedDataReader",
   "description": "Reader of resistics windowed data",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      }
   }
}
```

run(*dir_path: Path*, *metadata_only: bool = False*) → *WindowedMetadata* | *WindowedData*

Read WindowedData

Parameters

- **dir_path** (*Path*) – The directory path to read from
- **metadata_only** (*bool, optional*) – Flag for getting metadata only, by default False

Returns

The WindowedData or WindowedMetadata if metadata_only is True

Return type

Union[*WindowedMetadata*, *WindowedData*]

Raises

ReadError – If the directory does not exist

4.3.2 Module contents

A package for the processing of magnetotelluric data

Resistics is a package for the robust processing of magnetotelluric data. It includes several features focussed on traceability and data investigation. For more information, visit the package website at: www.resistics.io

4.4 Literature references

CHAPTER

FIVE

INDEX

- genindex
- modindex
- search

BIBLIOGRAPHY

[Jones2009] Area selection for diamonds using magnetotellurics: Examples from southern Africa. Jones et al. (2009). Lithos, 112S, 83-92. doi: 10.1016/j.lithos.2009.06.011

PYTHON MODULE INDEX

r

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

Y