demo

December 27, 2021

1 Demo

Import libraries

[1]:
```
import src.isthmuslib as isli
import numpy as np
import pandas as pd
from typing import List, Dict
import pathlib
```

Disable scrolling

[2]:
```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

<IPython.core.display.Javascript object>

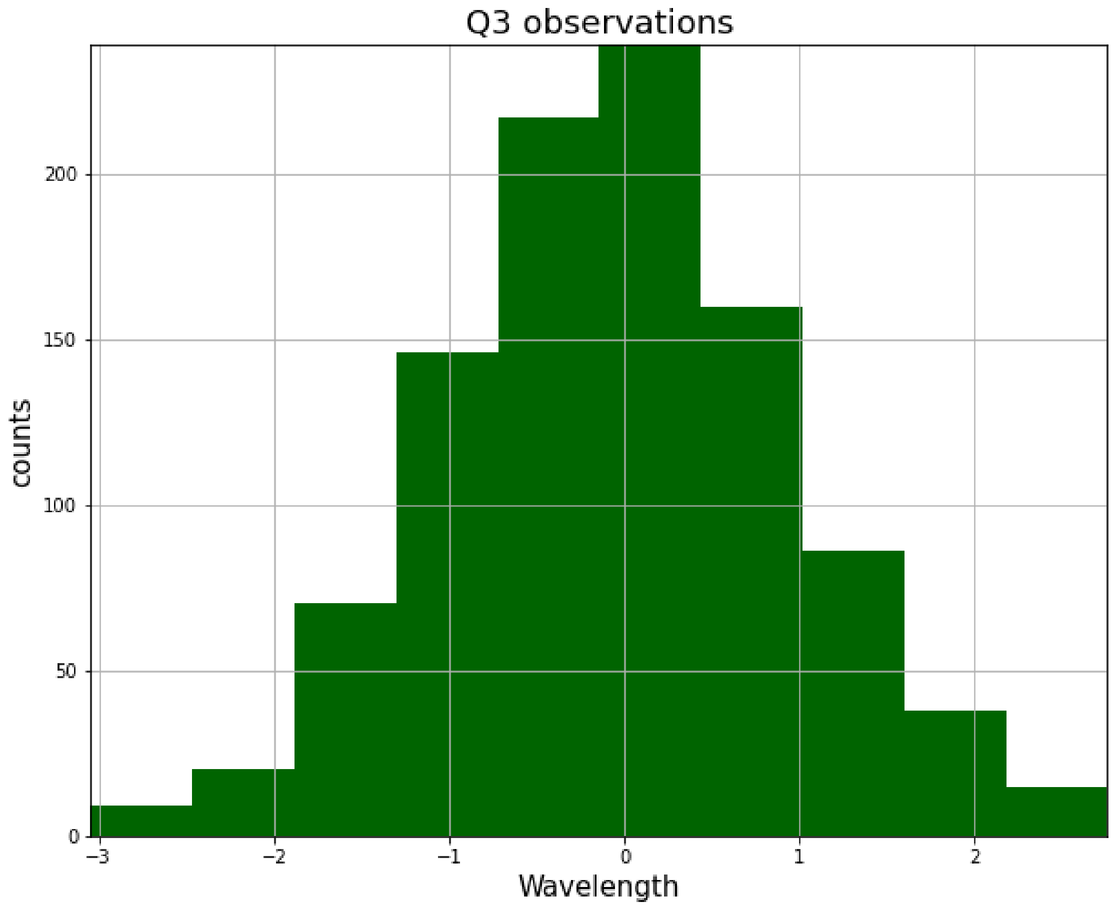
Make up some random sample data

[3]:
```
np.random.seed(0)
data_1: np.ndarray = np.random.normal(size=1000)
data_2: np.ndarray = [1 + x / 2 for x in data_1]
data_3: np.ndarray = np.random.normal(size=1000)
data_4: np.ndarray = np.random.normal(size=1000)
```

2 Visualize 1D Histogram

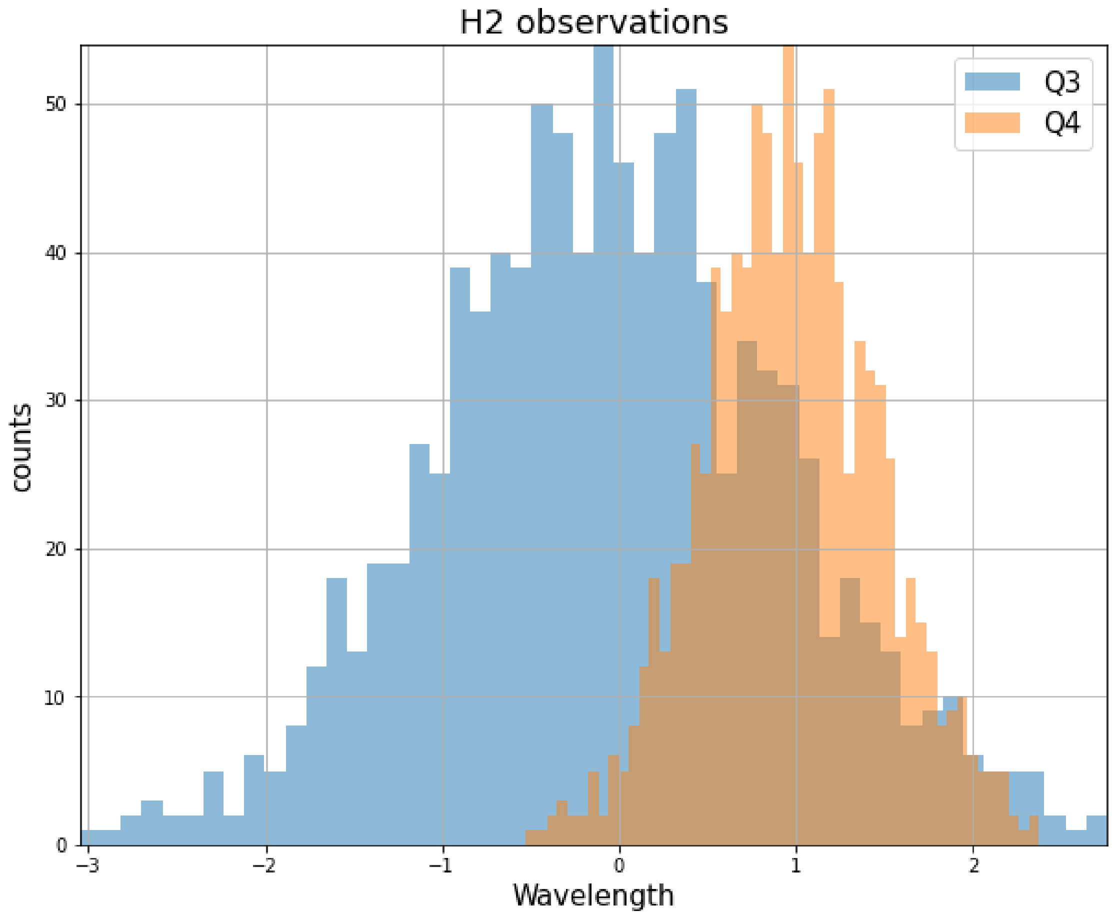
Single distribution

[4]:
```
isli.hist(data_1, xlabel='Wavelength', title='Q3 observations');
```



Multiple distributions

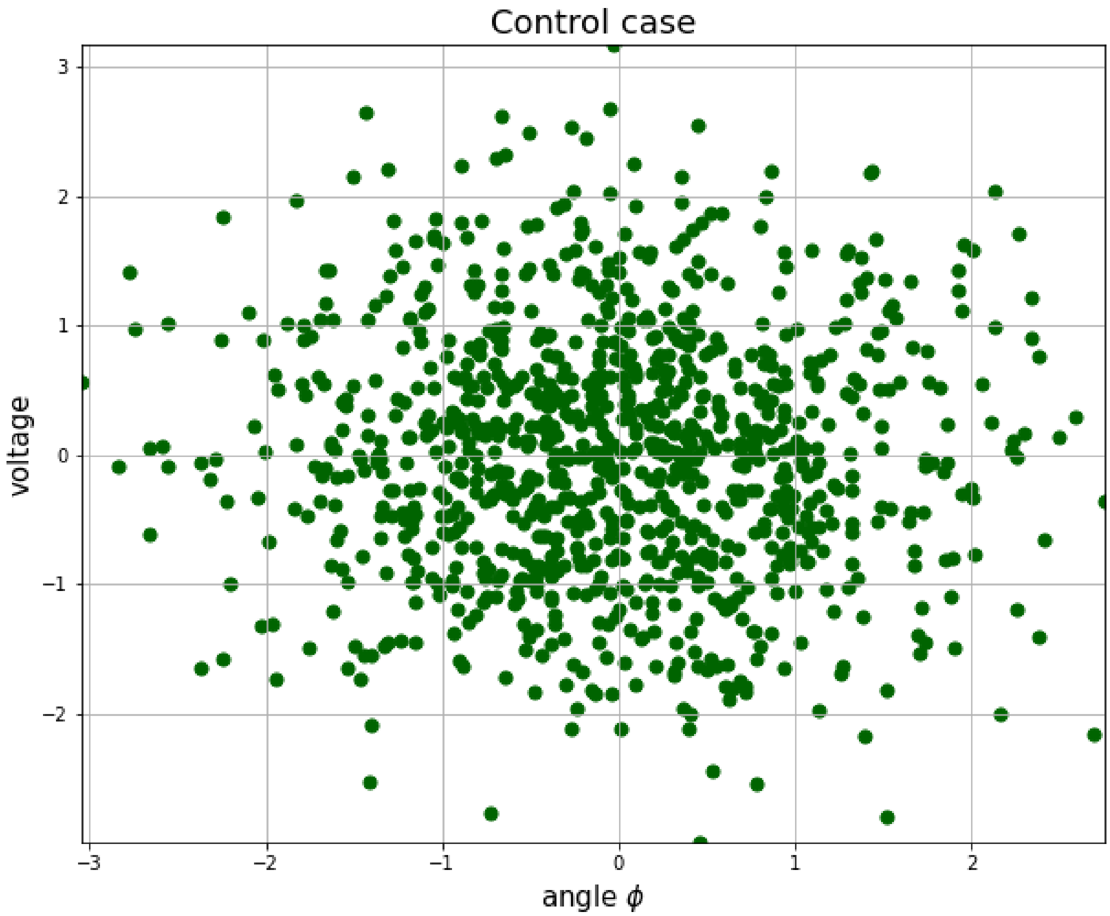
```
[5]: isli.hist([data_1, data_2], xlabel='Wavelength', title='H2 observations',␣
      ↪legend_strings=["Q3", "Q4"], bins=50);
```



2.1 Visualize 2D x & y

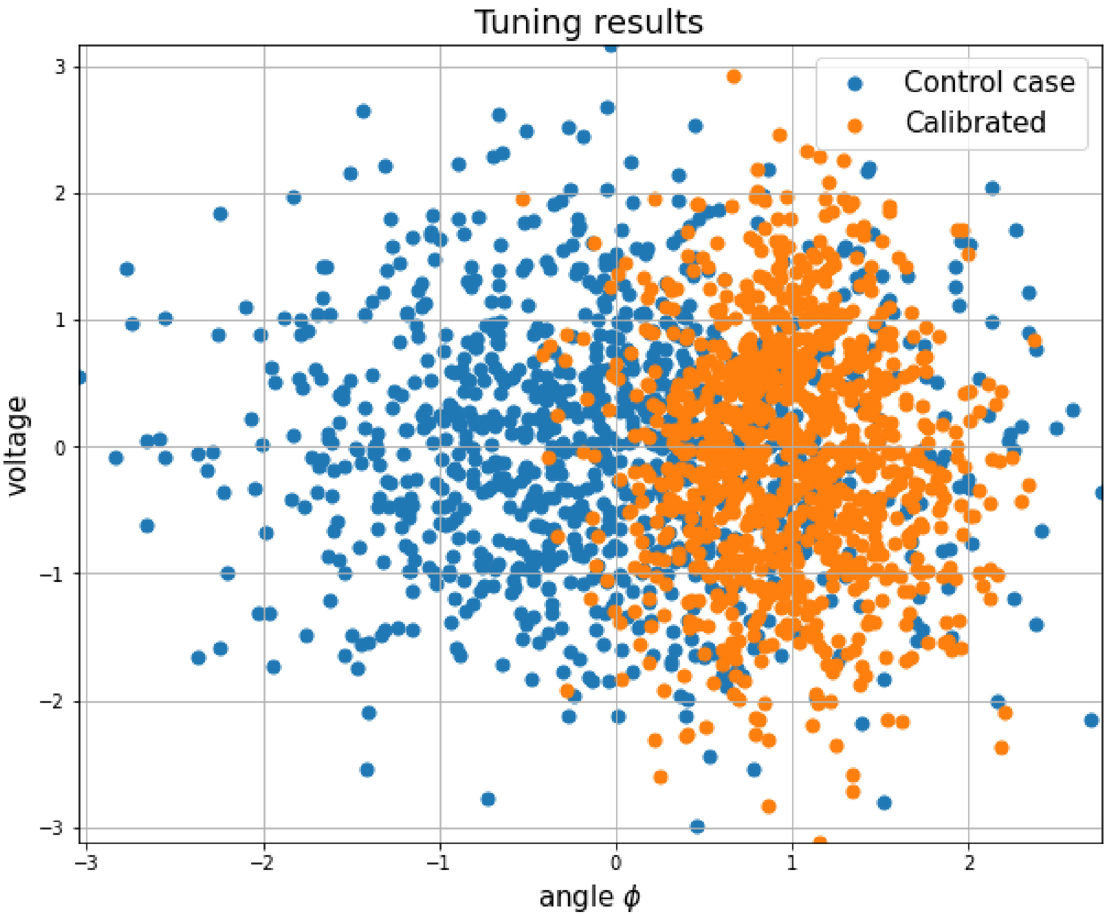
One data set

[6]:
```
isli.scatter(data_1, data_3, xlabel='angle $\phi$', ylabel='voltage',␣
 ↪title='Control case');
```



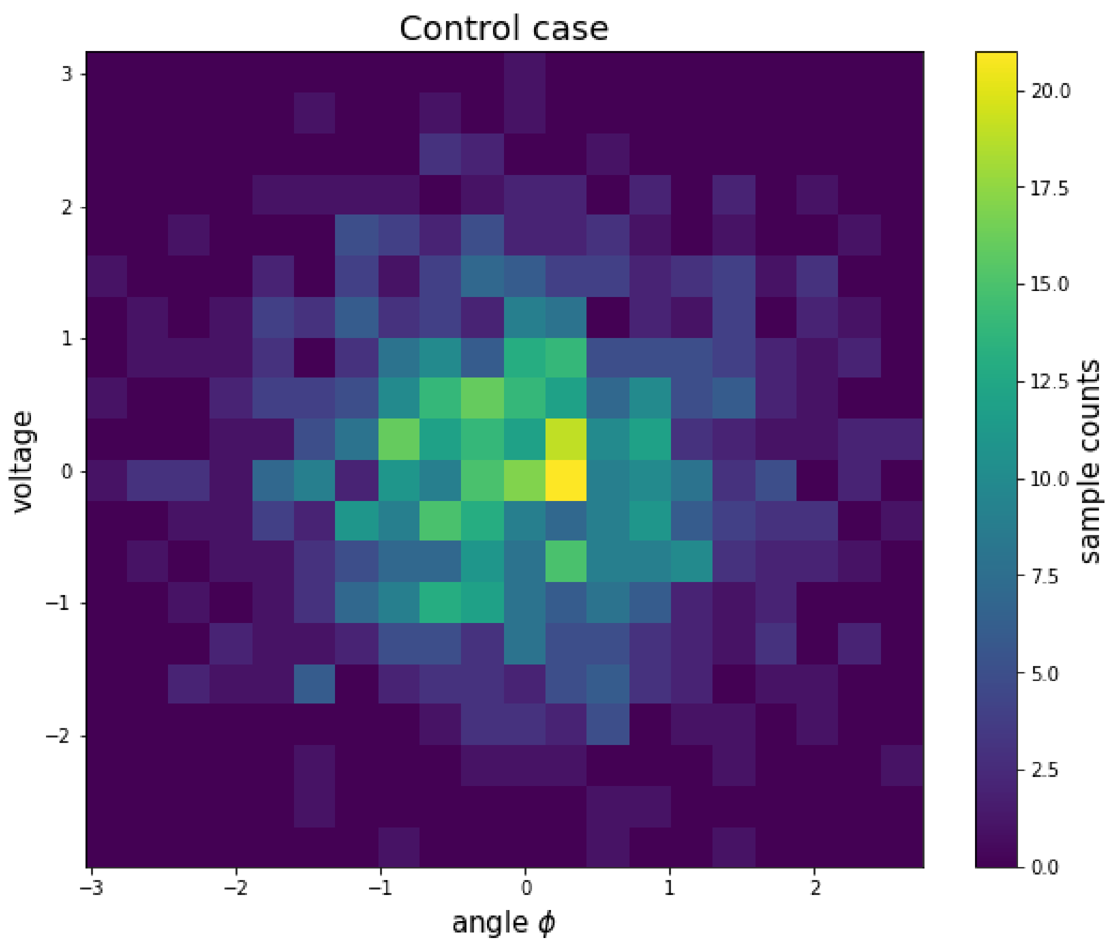
Multiple data sets

[7]:
```
isli.scatter([data_1, data_2], [data_3, data_4], xlabel='angle $\phi$',␣
↪ylabel='voltage', title='Tuning results',
              legend_strings=['Control case', 'Calibrated']);
```



We can also cast a single x & y vector pair into a 2D histogram (essentially a surface with height [color] showing bin counts)

[8]:
```
isli.hist2d(data_1, data_3, xlabel='angle $\phi$', ylabel='voltage',␣
↪title='Control case', bins=(20, 20),
             colorbar_label='sample counts');
```



2.1.1 Plotting surfaces

To-do: add some demos for plotting surfaces

2.2 Working with Vector Sequences

The timeseries-like `VectorSequence` class has `.read_csv()` method for easy import (don't forget to set the `basis_col_name`)

[9]:
```
timeseries: isli.VectorSequence = isli.VectorSequence().read_csv(
    pathlib.Path.cwd() / 'data' / 'version_controlled' /␣
↪'example_vector_sequence_data.csv', inplace=False,
    basis_col_name='timestamp', name_root='Experiment gamma')
```

The data is stored in a dataframe, so any pandas style calls and commands are available, for example:

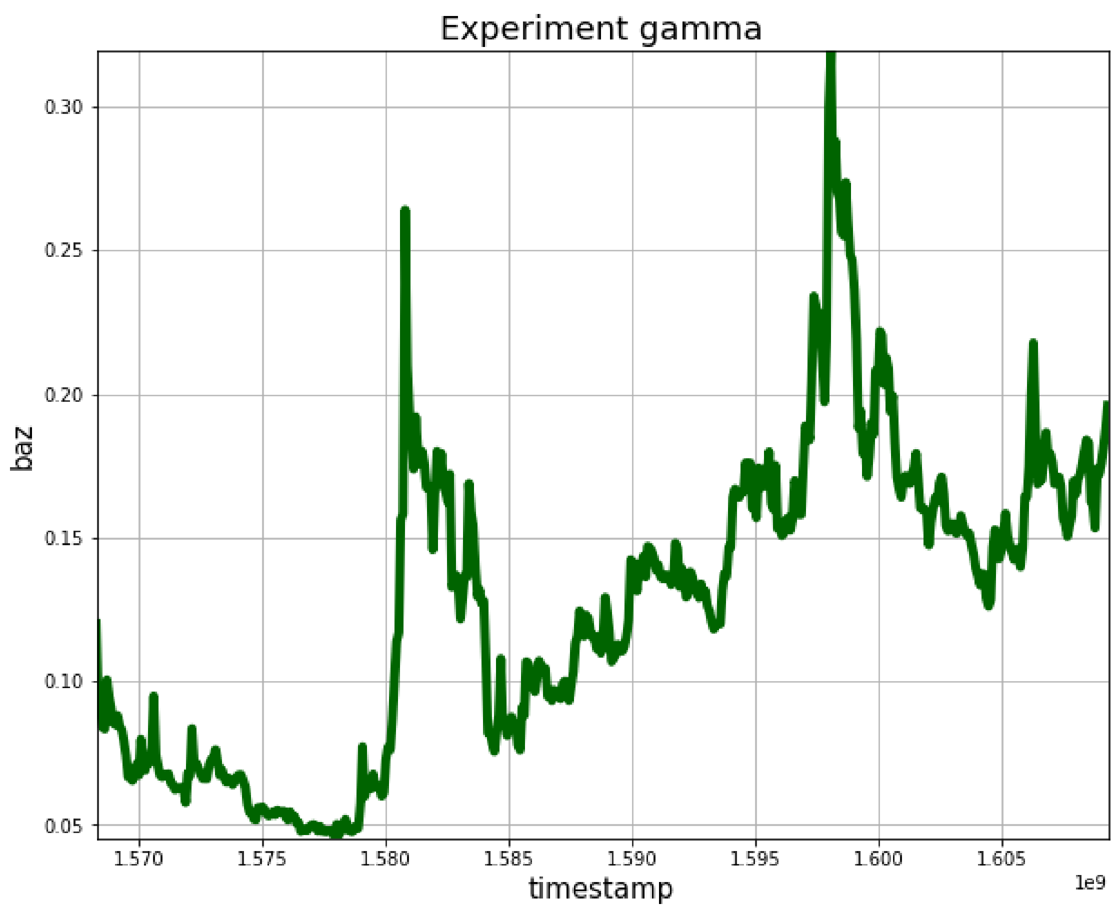
[10]:
```
timeseries.data.sort_values(by='foo', ascending=True, inplace=False).head(15)
```

[10]:

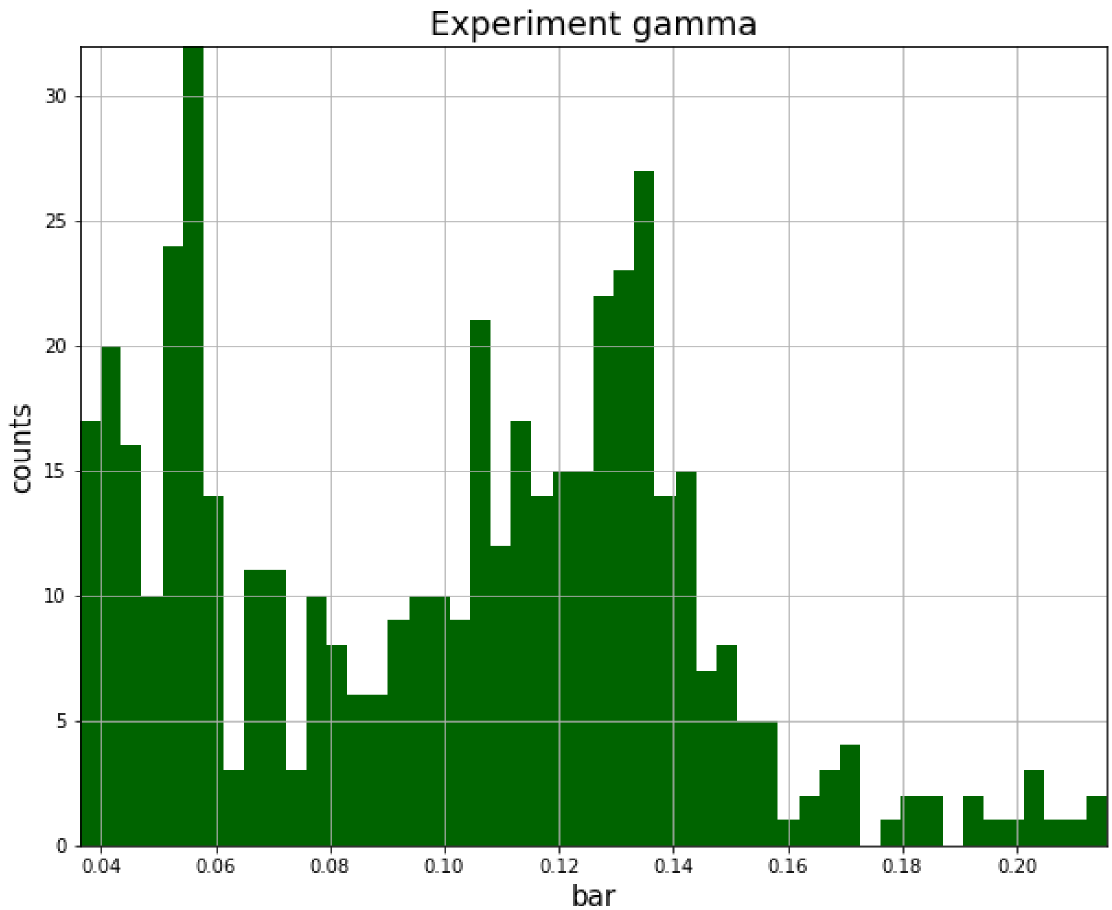
	timestamp	foo	bar	baz
112	1578034800	0.05380	0.038067	0.05024
97	1576652400	0.05460	0.039700	0.04852
110	1577862000	0.05650	0.040167	0.04820
111	1577948400	0.05665	0.036333	0.04536
115	1578294000	0.05715	0.041900	0.05028
114	1578207600	0.05735	0.038233	0.04800
99	1576825200	0.05800	0.039600	0.04800
113	1578121200	0.05815	0.038033	0.04652
118	1578553200	0.05830	0.038600	0.04800
119	1578639600	0.05850	0.039633	0.04756
106	1577430000	0.05895	0.039767	0.04772
96	1576566000	0.05910	0.036567	0.04772
108	1577602800	0.05940	0.039600	0.04752
104	1577257200	0.05980	0.039600	0.04784
105	1577343600	0.05980	0.039600	0.04948

Plotting methods in the above style are attached to the timeseries itself. If we call that object's `.plot()` or `.hist()` (etc) methods, it will automatically create consistently styled and labeled plots

[11]:
```
timeseries.plot('baz');
```



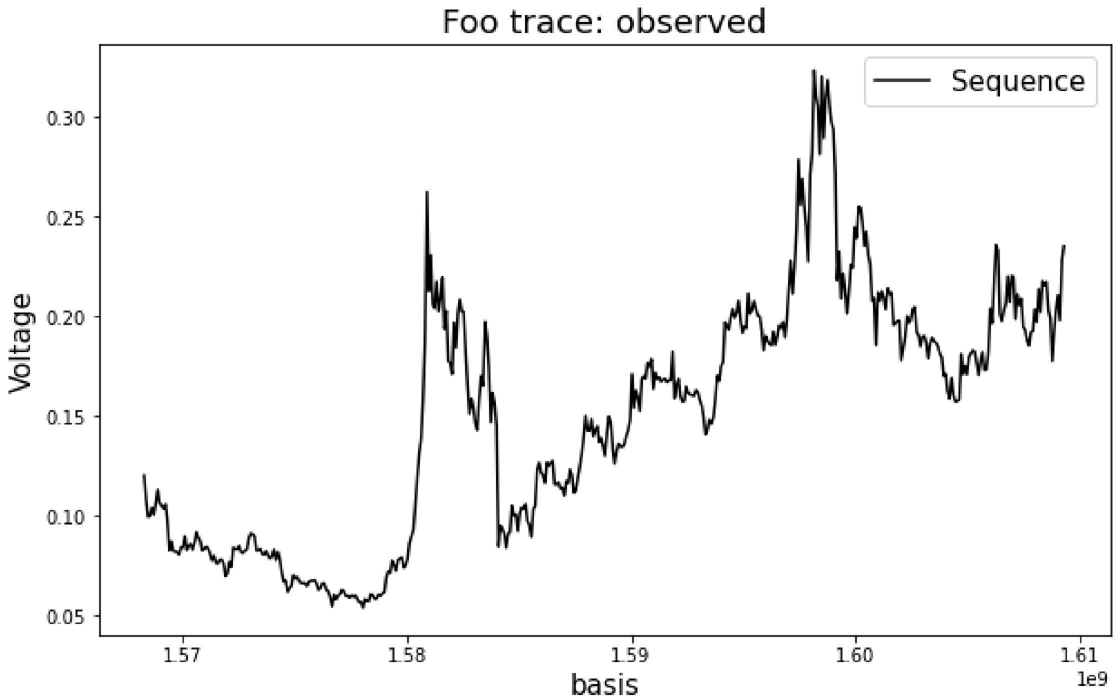
[12]:
```
timeseries.hist('bar', bins=50);
```

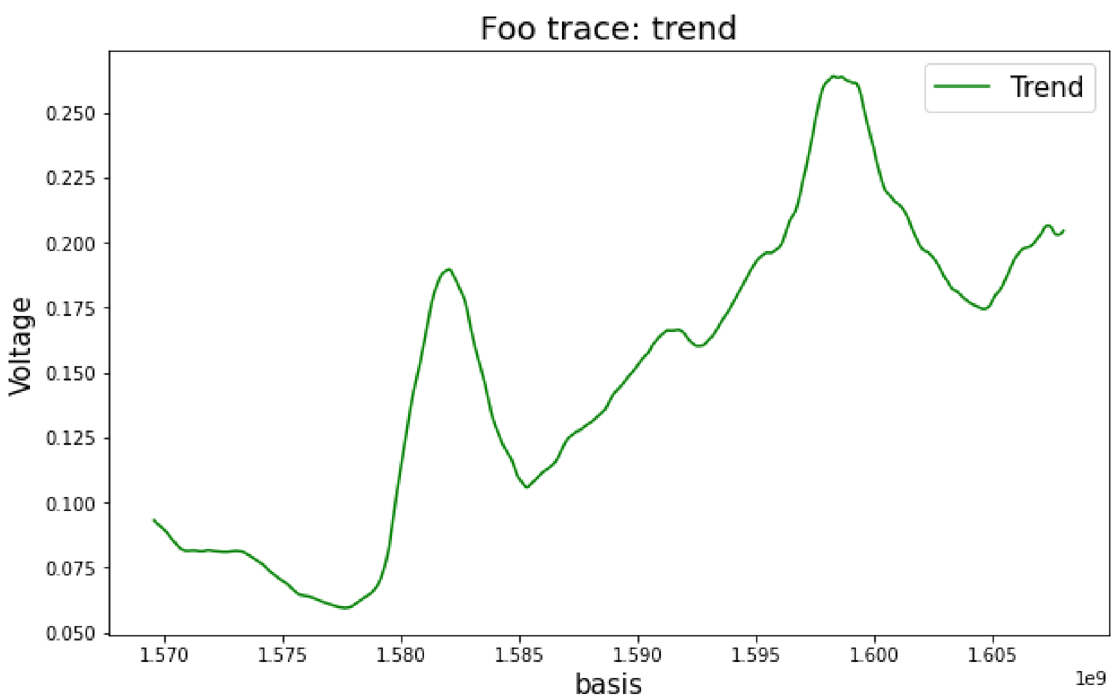


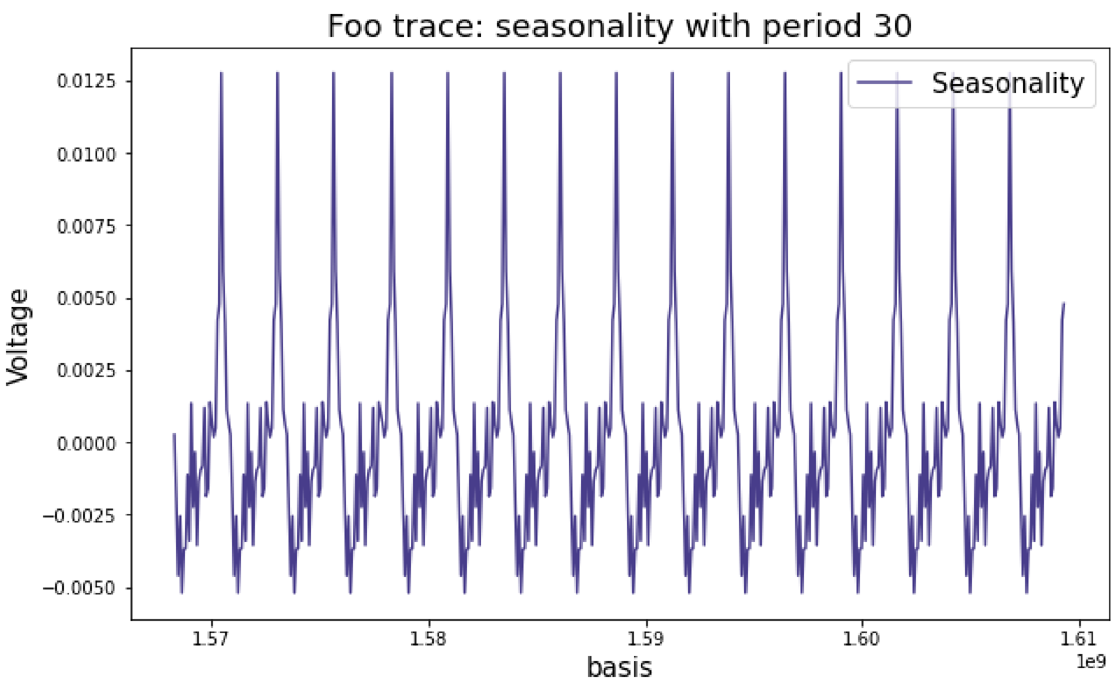
2.2.1 Seasonal decomposition

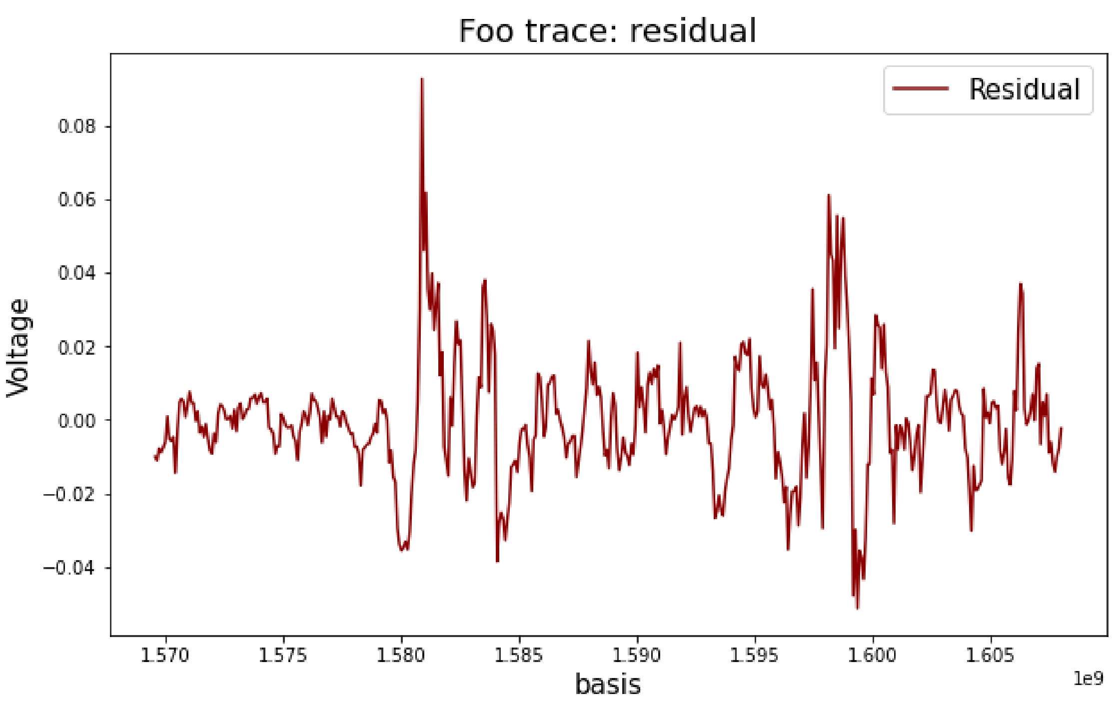
We can visualize seasonal decomposition analyses with a single line, wrapping `statsmodel.tsa`

[13]:
```
timeseries.plot_decomposition('foo', 30, figsize=(10, 6), title='Foo trace: ',␣
↪ylabel='Voltage');
```


Foo trace: observed
Sequence
Voltage
0.30
0.25
0.20
0.15
0.10
0.05
1.57
1.58
1.59
1.60
1.61
basis
1e9


Foo trace: trend
Trend
Voltage
0.250
0.225
0.200
0.175
0.150
0.125
0.100
0.075
0.050
1.570
1.575
1.580
1.585
1.590
1.595
1.600
1.605
basis
1e9


Foo trace: seasonality with period 30
Seasonality
Voltage
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
−0.0025
−0.0050
1.57
1.58
1.59
1.60
1.61
1e9
basis


Foo trace: residual
Residual
Voltage
0.08
0.06
0.04
0.02
0.00
−0.02
−0.04
1.570
1.575
1.580
1.585
1.590
1.595
1.600
1.605
1e9
basis

2.2.2 Sliding window analyses

The VectorSequence timeseries class contains logic for sliding window analyses with arbitrary functions. Here we'll use a throwaway lambda `appreciation` to demonstrate

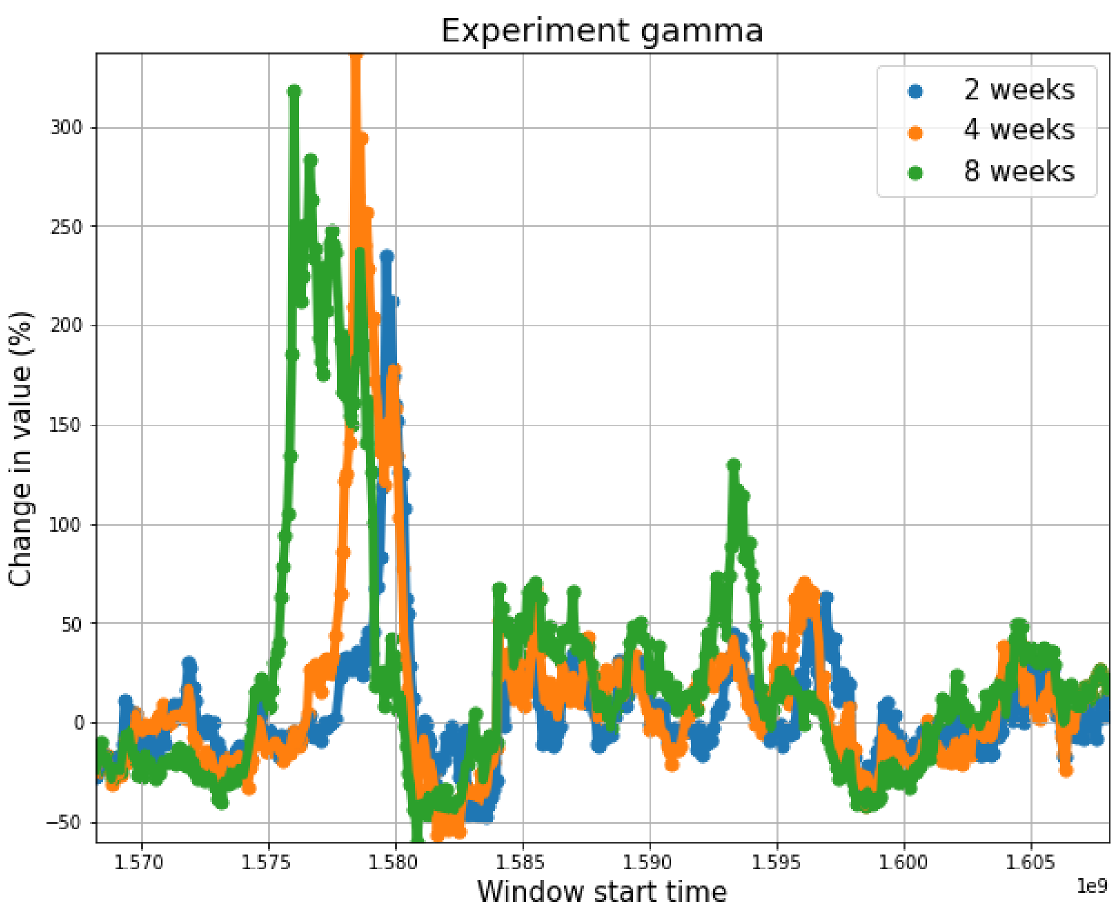
[14]:
```
appreciation = lambda o: {'Change in value (%)': 100 * (o.values('foo')[-1] / o.
↪values('foo')[0] - 1)}
```

Apply the function over sliding windows with 2, 4, and 8 week durations

[15]:
```
window_widths_weeks: List[float] = [2, 4, 8]
result: isli.SlidingWindowResults = timeseries.sliding_window(appreciation,
                                                              [x * 60 * 60 * 24␣
↪* 7 for x in window_widths_weeks],
                                                              overlapping=True)
```

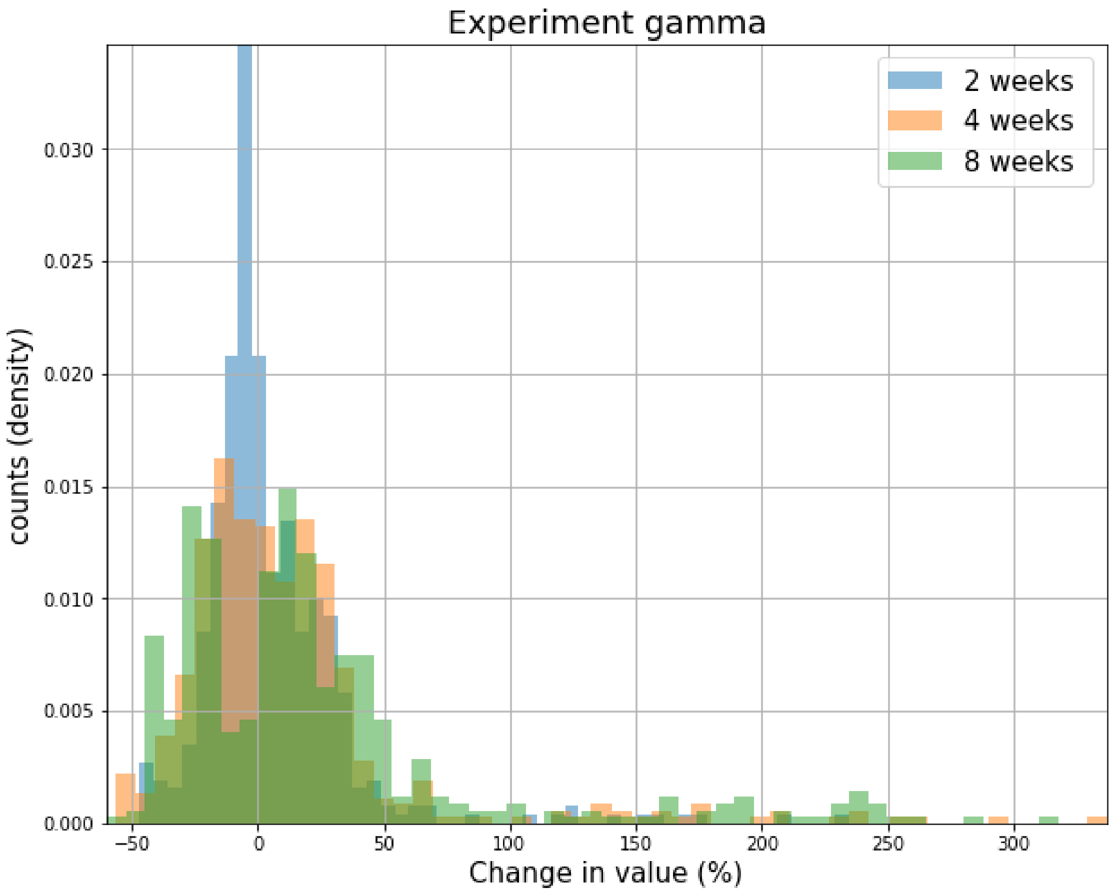
The `SlidingWindowResult.plot_results()` method automatically plots results separated by window width

[16]:
```
f = result.plot_results('Change in value (%)', legend_override=[f"{x} weeks "␣
↪for x in window_widths_weeks]);
```



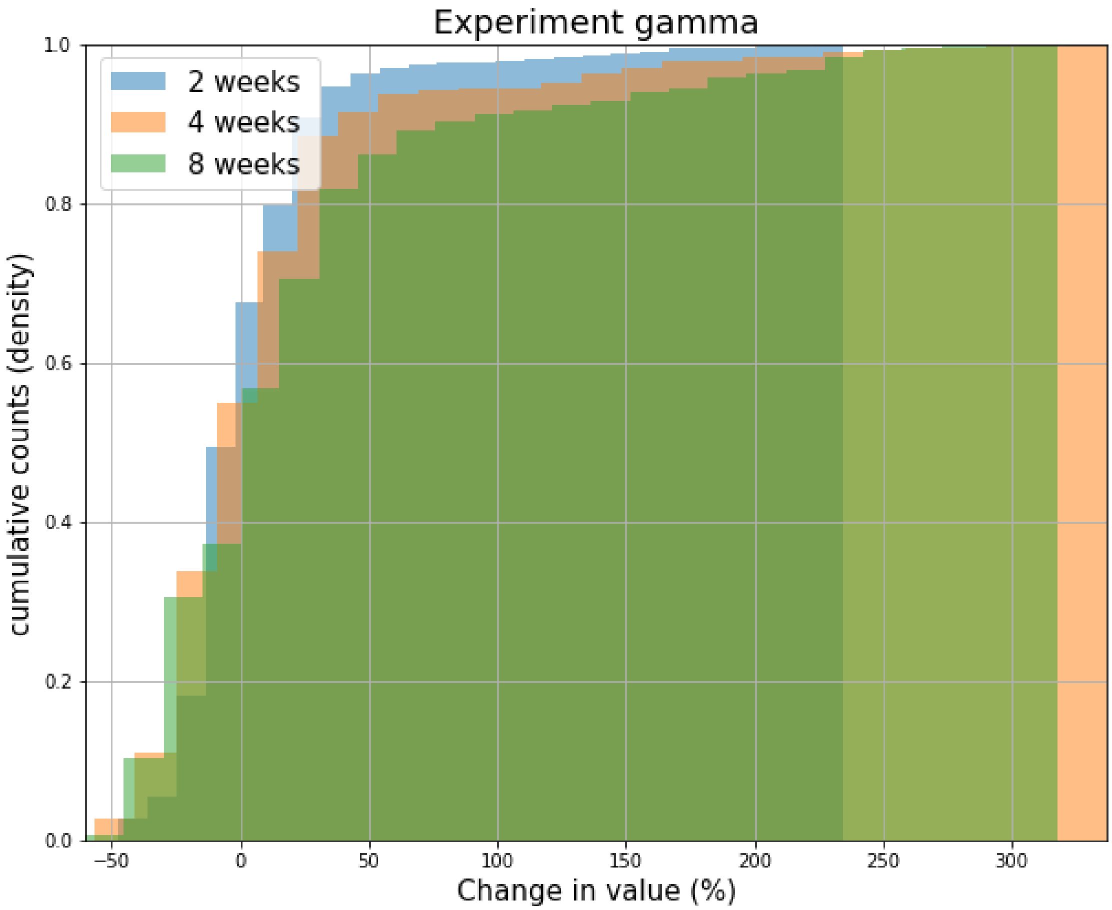
Likewise, the `sliding_window.plot_pdfs()` method plots distributions separated by window width

[17]:
```
result.plot_pdfs('Change in value (%)', density=True, bins=50,
                 legend_override=[f"{x} weeks " for x in window_widths_weeks]);
```



Adding `cumulative=True` produces the CDF

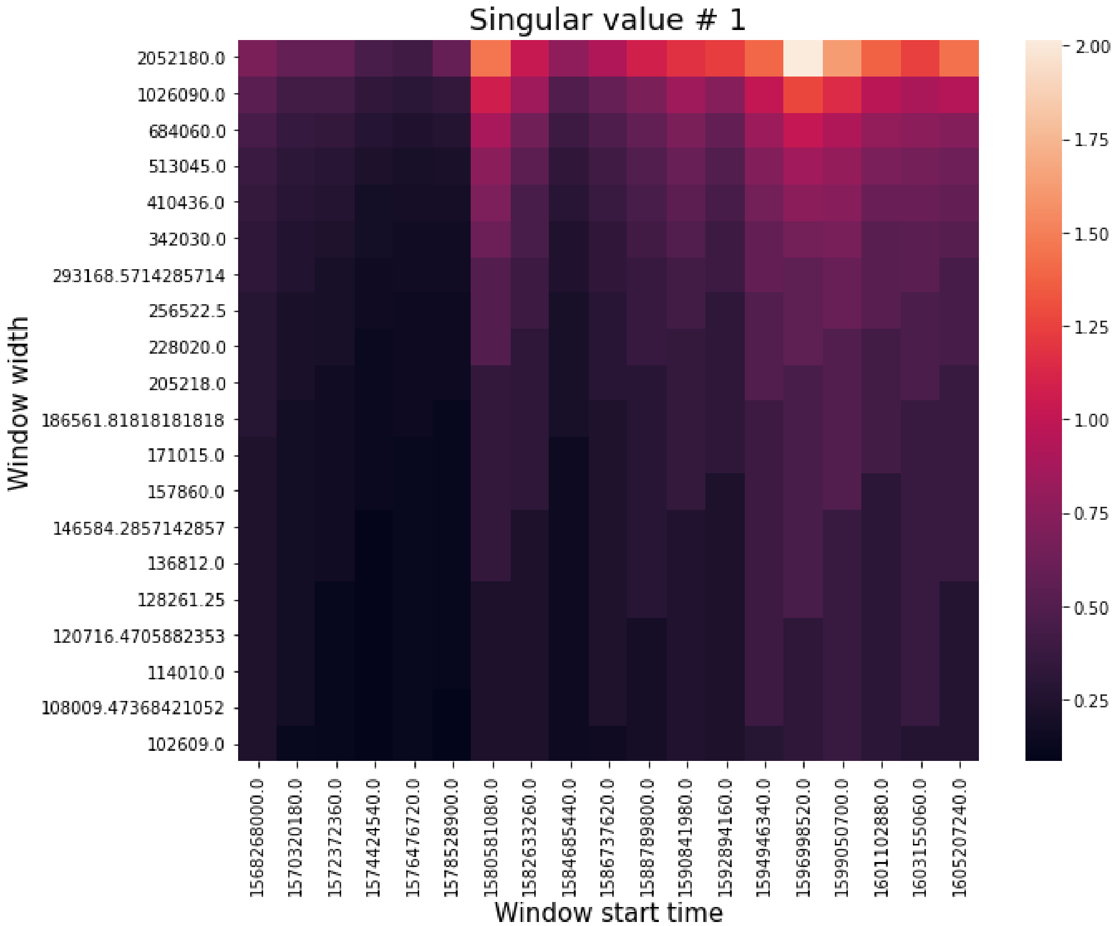
[18]:
```
result.plot_pdfs('Change in value (%)', density=True, cumulative=True, bins=25,
                 legend_override=[f"{x} weeks " for x in window_widths_weeks]);
```

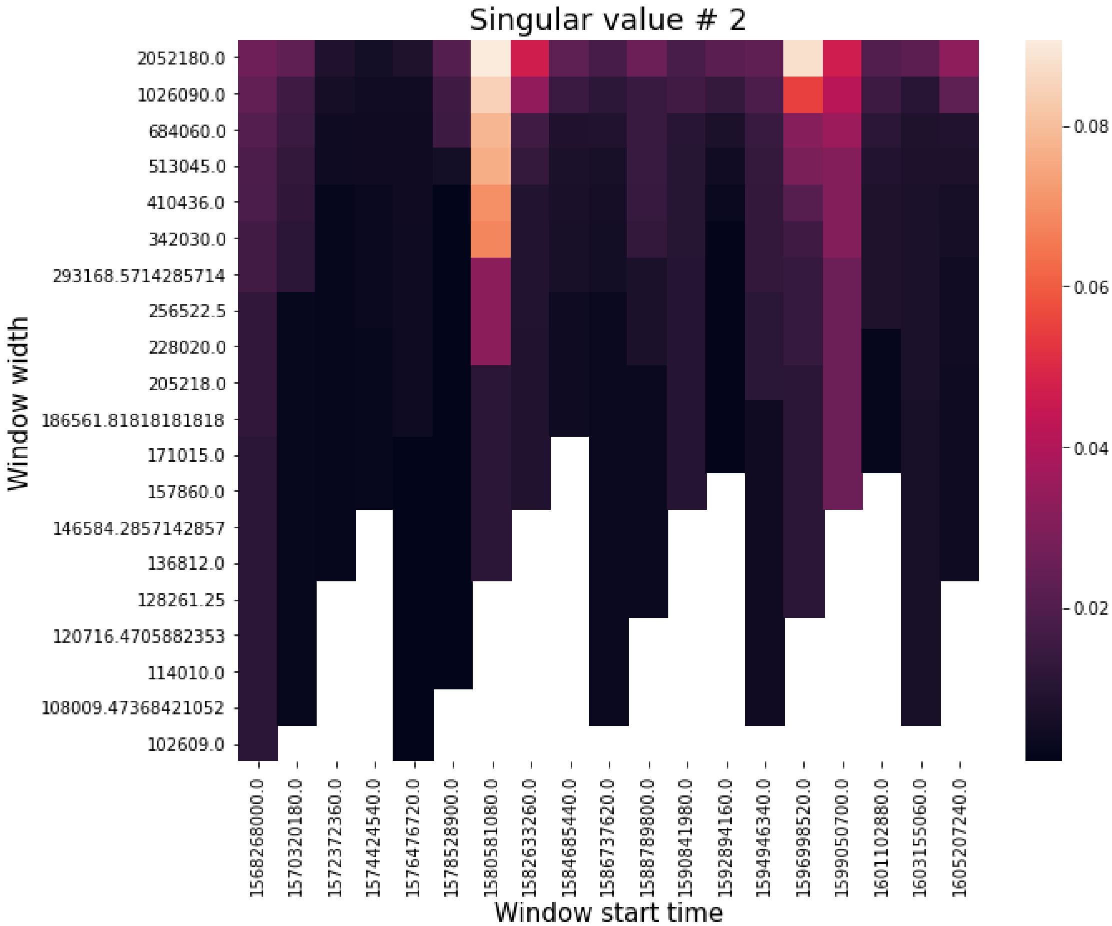


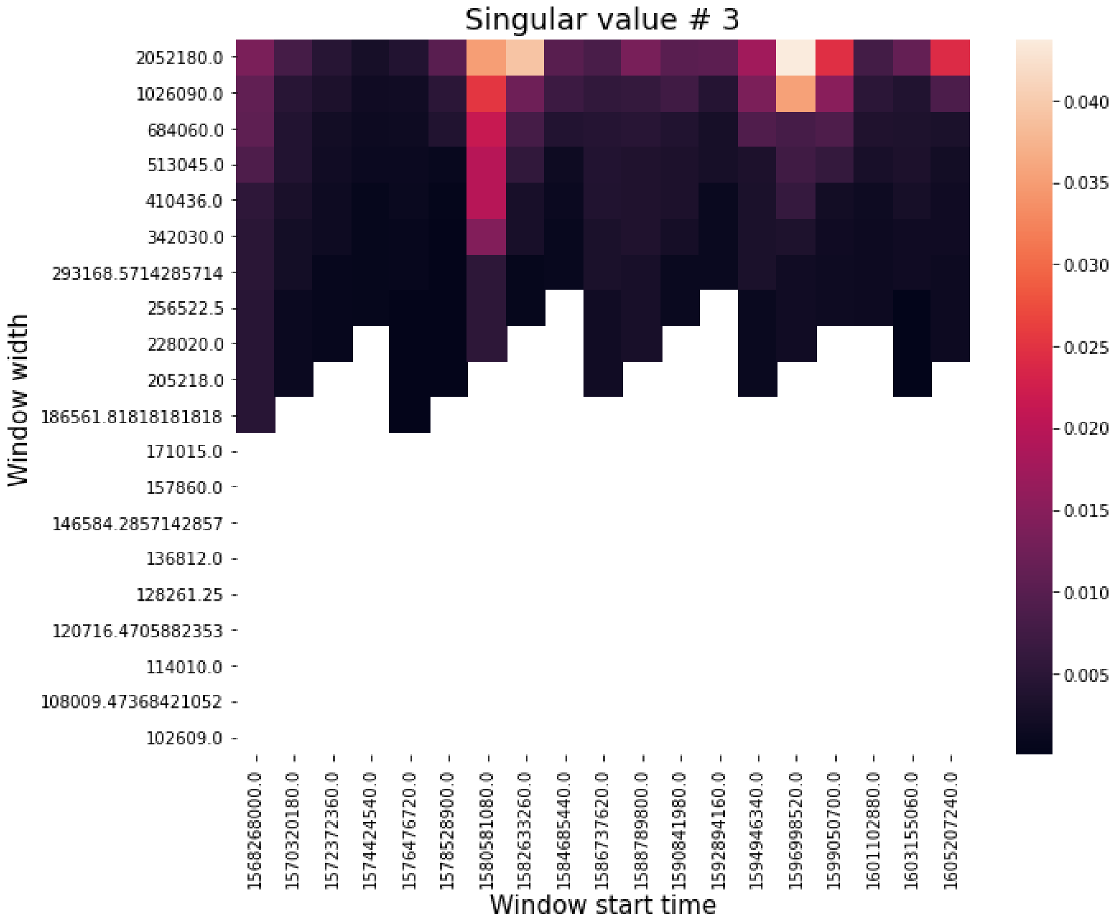
2.3 Dimensionality reduction and information content analyses

Dimensionality reduction (SVD) logic over sliding windows is built into the `VectorSequence` class, allowing easy calculation and visualization of information surfaces (first 3 singular value surfaces shown below)

[19]:
```
timeseries.plot_info_surface(cols=['baz', 'foo', 'bar']);
```


Singular value # 1
Window width
Window start time
2052180.0
1026090.0
684060.0
513045.0
410436.0
342030.0
293168.5714285714
256522.5
228020.0
205218.0
186561.81818181818
171015.0
157860.0
146584.2857142857
136812.0
128261.25
120716.4705882353
114010.0
108009.47368421052
102609.0
1568268000.0
1570320180.0
1572372360.0
1574424540.0
1576476720.0
1578528900.0
1580581080.0
1582633260.0
1584685440.0
1586737620.0
1588789800.0
1590841980.0
1592894160.0
1594946340.0
1596998520.0
1599050700.0
1601102880.0
1603155060.0
1605207240.0
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25


Singular value # 2
Window width
2052180.0
1026090.0
684060.0
513045.0
410436.0
342030.0
293168.5714285714
256522.5
228020.0
205218.0
186561.81818181818
171015.0
157860.0
146584.2857142857
136812.0
128261.25
120716.4705882353
114010.0
108009.47368421052
102609.0
1568268000.0
1570320180.0
1572372360.0
1574424540.0
1576476720.0
1578528900.0
1580581080.0
1582633260.0
1584685440.0
1586737620.0
1588789800.0
1590841980.0
1592894160.0
1594946340.0
1596998520.0
1599050700.0
1601102880.0
1603155060.0
1605207240.0
Window start time
0.08
0.06
0.04
0.02


Singular value # 3
Window width
Window start time
2052180.0
1026090.0
684060.0
513045.0
410436.0
342030.0
293168.5714285714
256522.5
228020.0
205218.0
186561.81818181818
171015.0
157860.0
146584.2857142857
136812.0
128261.25
120716.4705882353
114010.0
108009.47368421052
102609.0
1568268000.0
1570320180.0
1572372360.0
1574424540.0
1576476720.0
1578528900.0
1580581080.0
1582633260.0
1584685440.0
1586737620.0
1588789800.0
1590841980.0
1592894160.0
1594946340.0
1596998520.0
1599050700.0
1601102880.0
1603155060.0
1605207240.0
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005