seg1d

Release 0.1.2

Mathew Schwartz

Jul 30, 2021

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Check dependencies

Currently tested on `Python 3.8+`.

For the package: `numpy>=1.15`, `scipy>=1.0.0`, `sklearn>=0.2`, `numba>=0.40`

For building documentation: `sphinx`, `scipy` html_theme, `numpydoc`, `matplotlib`

1.1.2 Get seg1d

```
pip install seg1d
```

1.1.3 Building Documentation

Documentation is built using sphinx with the scipy html_theme in the `docs` folder.

To get the requirements for building documentation you can run:

```
pip install sphinx numpydoc matplotlib
```

If you have not installed `seg1d` you can then run the following to get the remaining requirements

```
pip install numpy scipy sklearn numba
```

Finally, the following two lines will build the documentation.

```
make clean
make html
```

1.1.4 Testing

Docstrings examples should be compliant with `doctest`. Navigate to the `docs` folder and run:

```
make doctest
```

A summary report will be generated and should not have any failed conditions. Note, this should be done after building the documentation with `make html`.

End-users can test the installation with a similar method provided through the examples.

After installation, start a python interactive console and import:

`from seg1d.examples import test`

Then run the tests with:

`test.run()`

A series of tests will be performed to check for intended output. While `matplotlib` is not a required dependency, if it is not installed, some of the tests will fail. This will be obvious as the test will show in the traceback that it failed due to `No module named 'matplotlib'`.

Due to variations in OS, many of the doctests use `np.around` to ensure matching. However, sorting of correlation values is dependent on a few platform variables, so tests that output multiple array values that are identical may show as failed. The documentation is always tested on Windows 10, Python 3.8 for ensuring compliance.

1.2 Getting Started

In this section, a few simple examples are given to ensure the installation is working and you are able to segment data. These examples call a method provided that uses default parameters for the algorithms within the segmentation process but allows user input for the data related parameters such as scaling and step size for the matching process.

1.2.1 Sample Data

A basic example of checking the installation using the included sample data.

Example using included sample data

```
>>> import seg1d
>>> import numpy as np
>>> #retrieve the sample reference, target, and weight data
>>> r,t,w = seg1d.sampleData()
>>> ### define some test parameters
>>> minW = 70 #minimum percent to scale down reference data
>>> maxW = 150 #maximum percent to scale up reference data
>>> step = 1 #step to use for correlating reference to target data
>>> #call the segmentation algorithm
>>> np.around( seg1d.segment_data(r,t,w,minW,maxW,step) , decimals=7 )
array([[207.       , 240.       ,   0.9124224],
       [342.       , 381.       ,   0.8801901],
       [ 72.       , 112.       ,   0.8776795]])
```

1.2.2 Sine Wave

Sample using sine wave

```
>>> import seg1d
>>> import numpy as np
>>> import matplotlib.pylab as plt
```

Data can be constructed as a numpy array

```
>>> # create an array of data
>>> x = np.linspace(-np.pi*2, np.pi*2, 2000)
```

(continues on next page)

(continued from previous page)

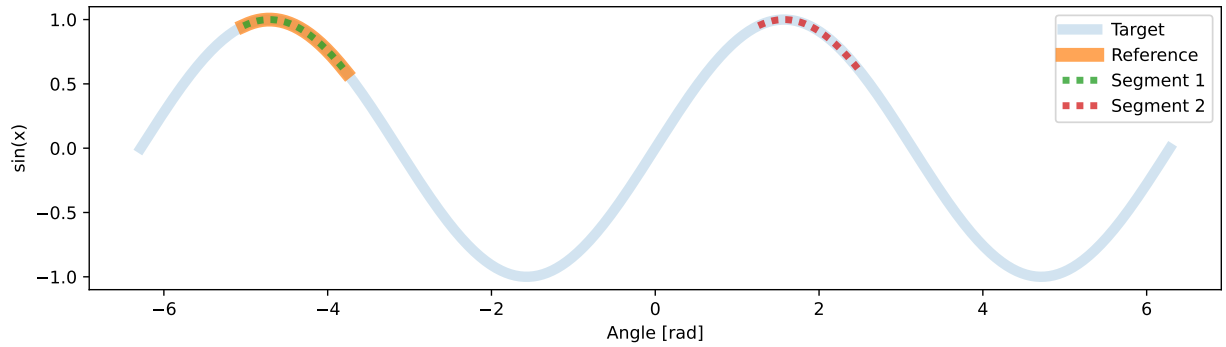
```
>>> # get an array of data from a sin function
>>> targ = np.sin(x)
```

To use the basic method interface, the data must be labeled

```
>>> # define a segment within the sine wave to use as reference
>>> t_s,t_e = 200,400
>>> # cut a segment out to use as a reference data
>>> refData = [ { '0' : targ[t_s:t_e] } ]
>>> targData = {'0' : targ}
>>> refWeights = {'0' : 1}
>>>
>>> ### define some test parameters
>>> minWin = 98 #minimum percent to scale down reference data
>>> maxWin = 105 #maximum percent to scale up reference data
>>> sizeStep = 1 #step to use for correlating reference to target data
>>>
>>> #call the segmentation algorithm
>>> segments = seg1d.segment_data(refData,targData,refWeights,minWin,maxWin,sizeStep)
>>> np.around(segments, decimals=7)
array([[2.000000e+02, 4.000000e+02, 1.000000e+00],
       [1.200000e+03, 1.398000e+03, 9.999999e-01]])
```

Using matplotlib we can visualize the results

```
>>> plt.figure(figsize=(10,3))
>>> # plot the full sine wave
>>> plt.plot(x, targ,linewidth=6,alpha=0.2,label='Target')
>>> # plot the original reference segment
>>> plt.plot(x[t_s:t_e], targ[t_s:t_e],linewidth=8,alpha=0.7,label='Reference')
>>>
>>> # plot all segments found
>>> seg_num = 1
>>> for s,e,c in segments:
...     plt.plot(x[s:e], targ[s:e],dashes=[1,1],linewidth=4,alpha=0.8,
...     label='Segment {}'.format(seg_num))
...     seg_num += 1
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



1.2.3 Gauss

In this example a Gaussian pulse is used to show segmentation on the varying shape of different amplitude. As the center arc is given as reference, the multiple extending arcs are found as well. Through the output of the segments, the correlation values can be seen to decrease, although still clustered within the group.

```
>>> import seg1d
>>> import numpy as np
>>> import matplotlib.pylab as plt
>>> import scipy.signal as signal
```

```
>>> # create an array of data
>>> x = np.linspace(-1, 1, 2000)
>>> # get an array of data from a Gaussian pulse
>>> targ = signal.gausspulse(x, fc=5)
```

```
>>> # define a segment within the sine wave to use as reference
>>> t_s,t_e    = 950,1050
>>> # cut a segment out to use as a reference data
>>> refData    = [ { 'gauss' : targ[t_s:t_e] } ]
>>> targData   =   { 'gauss' : targ }
>>> refWeights =   { 'gauss' : 1 }
```

```
>>> ### define some test parameters
>>> minWin   = 98 #minimum percent to scale down reference data
>>> maxWin   = 105 #maximum percent to scale up reference data
>>> sizeStep = 1 #step to use for correlating reference to target data
```

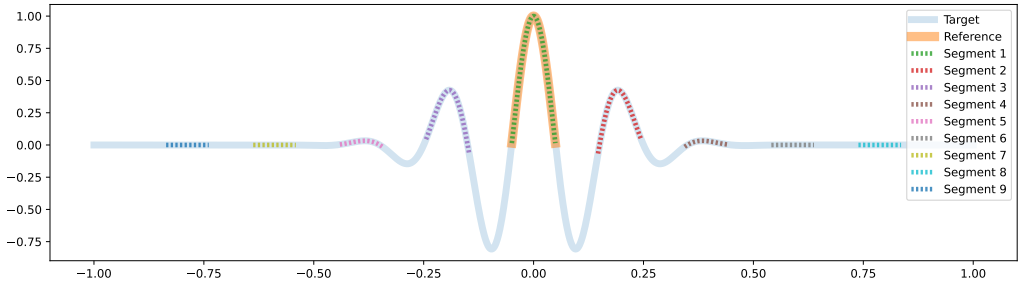
```
>>> # call the segmentation algorithm
>>> segments = seg1d.segment_data(refData,targData,refWeights,minWin,maxWin,sizeStep)
>>> print(np.around(segments,decimals=7))
[[9.500000e+02 1.050000e+03 1.000000e+00]
 [1.146000e+03 1.245000e+03 9.867665e-01]
 [7.550000e+02 8.540000e+02 9.867665e-01]
 [1.343000e+03 1.441000e+03 9.498135e-01]
 [5.590000e+02 6.570000e+02 9.498135e-01]
 [1.540000e+03 1.638000e+03 8.949109e-01]
 [3.620000e+02 4.600000e+02 8.949109e-01]
 [1.738000e+03 1.836000e+03 8.301899e-01]
```

(continues on next page)

(continued from previous page)

```
 [1.640000e+02 2.620000e+02 8.301899e-01]]
```

```
>>> plt.figure(figsize=(15,4))
>>> # plot the full pulse
>>> plt.plot(x, targ,linewidth=6,alpha=0.2,label='Target')
>>> # plot the original reference segment
>>> plt.plot(x[t_s:t_e], targ[t_s:t_e],linewidth=8,alpha=0.5,label='Reference')
>>> # plot all segments found
>>> seg_num = 1
>>> for s,e,c in segments:
...     plt.plot(x[s:e], targ[s:e],dashes=[0.5,0.5],linewidth=4,alpha=0.8,
...     label='Segment {}'.format(seg_num))
...     seg_num += 1
>>> plt.legend()
>>> plt.show()
```



1.3 API Examples

seg1d can be used through both the routines and through the `Segmenter` class, which provides various methods for constructing a segmenter by setting parameters and data.

This section contains the more advanced usage of the package by creating an instance of the class. The usage is explained through different sample datasets, both of real and generated data. Each example emphasizes a different aspect of the segmentation process.

API Examples

1.3.1 Basic Use

An example of instancing the Segmenter class to use the convenience methods on array data

```
>>> import seg1d
>>> import numpy as np
>>> import matplotlib.pylab as plt
```

Then we generate some data

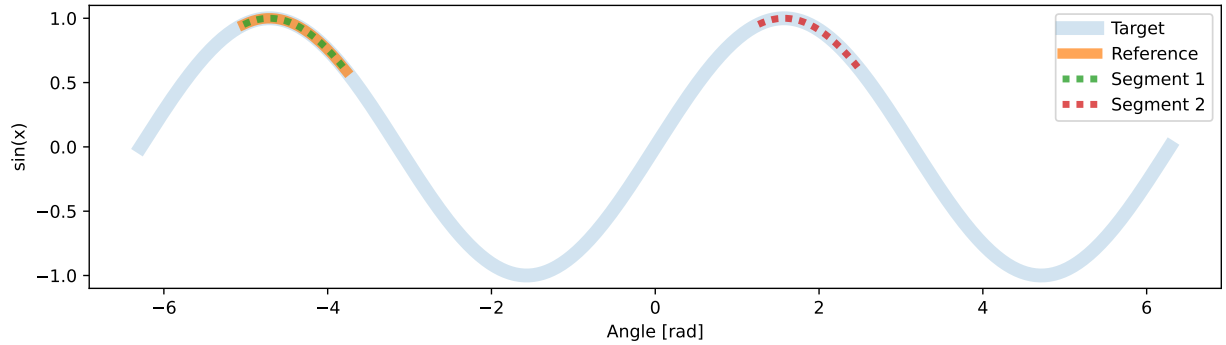
```
>>> x = np.linspace(-np.pi*2, np.pi*2, 2000) #create an array of data
>>> targ = np.sin(x)  # target data from a sin function
>>> t_s,t_e = 200,400 # define a sub-series
```

To assign the data to the Segmenter, first we create an instance of it and then use the `set_target()` and `add_reference()` methods.

```
>>> s = seg1d.Segmenter()  # instance of the segmenter
>>> s.minW, s.maxW, s.step = 98, 105, 1  # scaling parameters
>>> s.set_target(targ) # set target and reference data
>>> s.add_reference(targ[t_s:t_e])
>>> segments = s.segment() # run segmentation algorithm
>>> np.around(segments, decimals=7)
array([[2.000000e+02, 4.000000e+02, 1.000000e+00],
       [1.200000e+03, 1.398000e+03, 9.999999e-01]])
```

Using matplotlib we can visualize the results

```
>>> plt.figure(figsize=(10,3))
>>> #plot the full sine wave
>>> plt.plot(x, targ,linewidth=8,alpha=0.2,label='Target')
>>> #plot the original reference segment
>>> plt.plot(x[t_s:t_e], targ[t_s:t_e],linewidth=6,alpha=0.7,label='Reference')
>>>
>>> #plot all segments found
>>> seg_num = 1
>>> for s,e,c in segments:
...     plt.plot(x[s:e], targ[s:e],dashes=[1,1],linewidth=4,alpha=0.8,
...     label='Segment {}'.format(seg_num))
...     seg_num += 1
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



1.3.2 ECG

In this example we use the ECG data included with scipy signal module. The references roughly includes the Q-T interval (https://en.wikipedia.org/wiki/Electrocardiography). In the first portion, two sample segments are used. While the segments are not aligned, they are able to find some segments correctly. In the second portion of the example, only one segment is used for the reference data.

```
>>> import random
>>> import numpy as np
>>> from scipy.misc import electrocardiogram
>>> import matplotlib.pyplot as plt
>>> import seg1d
```

After imports, the scipy signal ECG data is called and some segments are taken.

```
>>> ecg = electrocardiogram() #get the scipy sample data
>>> ref_slices = [[927, 1057],[1111, 1229]] #pick sample endpoints
```

```
>>> s = seg1d.Segmenter()  #create the segmenter
```

```
>>> refs = [ ecg[x[0]:x[1]] for x in ref_slices ]
>>> for r in refs: s.add_reference(r) #set reference data
```

```
>>> s.set_target(ecg[1500:3500]) #set the target data to the ecg after ref
>>> segments = s.segment()  # run segmenter with defaults
```

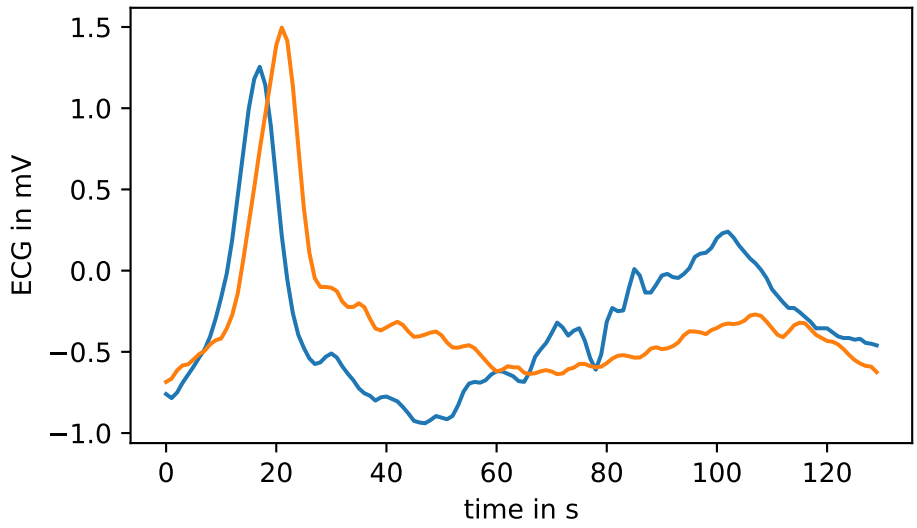
```
>>> print(np.around(segments,decimals=7))
[[1.607000e+03 1.729000e+03 8.169533e-01]
 [7.380000e+02 8.220000e+02 8.123868e-01]
 [9.190000e+02 1.003000e+03 8.120505e-01]
 [1.439000e+03 1.552000e+03 8.092366e-01]
 [3.600000e+02 4.930000e+02 8.077664e-01]
 [1.091000e+03 1.213000e+03 8.043364e-01]
 [1.775000e+03 1.895000e+03 7.998723e-01]
 [1.720000e+02 3.000000e+02 7.926582e-01]
 [1.268000e+03 1.340000e+03 7.847107e-01]
 [5.540000e+02 6.280000e+02 7.802931e-01]]
```

The reference data is automatically scaled to the largest reference in the dataset when the `segment` method is called. Therefore, by retrieving this attribute we can plot what the reference set looks like when the lengths are normalized.

In the example, it is clear the peaks of the reference segments are not aligned. This discrepency, due to the averaging of all reference data items, will be seen in the final segments of the target data later.

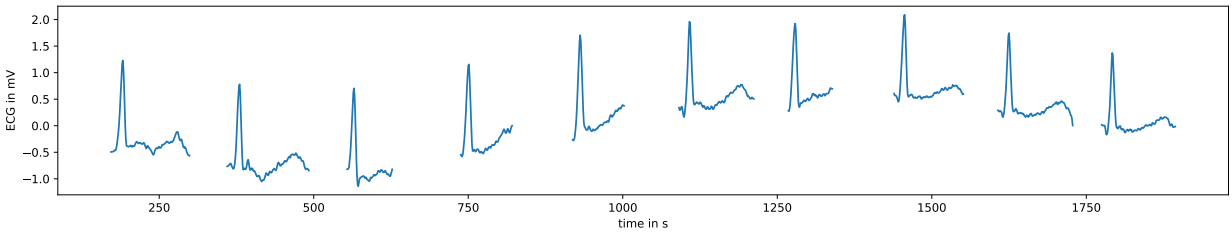
```
>>> refs = s.r
>>> refs = np.asarray( [ x[y] for x in refs for y in x ] )
```

```
>>> plt.figure(figsize=(5,3))
>>> plt.plot(refs.T)
>>> plt.xlabel("time in s")
>>> plt.ylabel("ECG in mV")
>>> plt.tight_layout()
>>> plt.show()
```



The final segments are shown by calling the property `t_masked` which returns the target data as an ndarray with NaN values for areas not found to be segments.

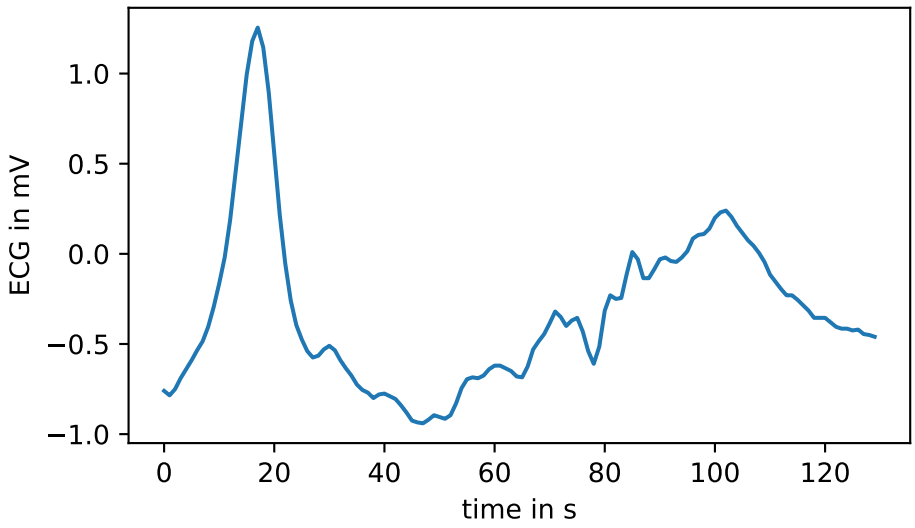
```
>>> plt.figure(figsize=(15,3))
>>> plt.plot(s.t_masked.T)
>>> plt.xlabel("time in s")
>>> plt.ylabel("ECG in mV")
>>> plt.tight_layout()
>>> plt.show()
```




```
>>> #use only 1 reference
>>> s.clear_reference()
>>> s.add_reference( ecg[927:1057] )
```

```
>>> refs = s.r
>>> refs = np.asarray( [ x[y] for x in refs for y in x ] )
```

```
>>> plt.figure(figsize=(5,3))
>>> plt.plot(refs.T)
>>> plt.xlabel("time in s")
>>> plt.ylabel("ECG in mV")
>>> plt.tight_layout()
>>> plt.show()
```



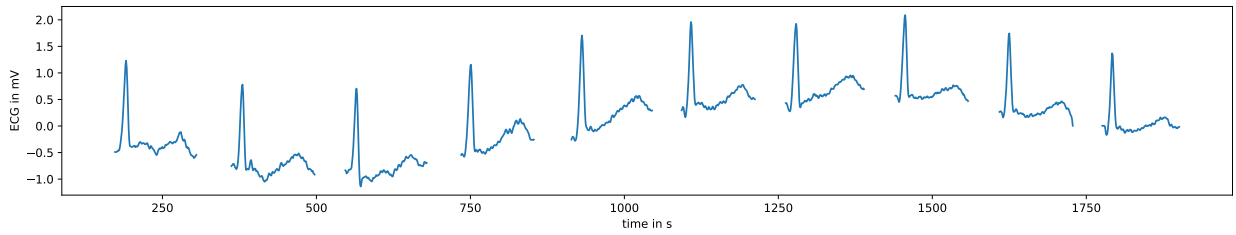

```
>>> #remove first part of data (contains reference)
>>> s.set_target(ecg[1500:3500])
>>> s.nC = 2
>>> s.cMin = 0.7
```

```
>>> segments = s.segment()
```

```
>>> print(np.around(segments,decimals=7))
[[7.350000e+02 8.540000e+02 9.462850e-01]
 [1.093000e+03 1.213000e+03 9.242974e-01]
 [9.140000e+02 1.046000e+03 9.059727e-01]
 [3.620000e+02 4.980000e+02 9.009127e-01]
 [5.470000e+02 6.800000e+02 8.940106e-01]
 [1.262000e+03 1.390000e+03 8.868629e-01]
 [1.776000e+03 1.902000e+03 8.771139e-01]
 [1.609000e+03 1.729000e+03 8.689476e-01]
 [1.440000e+03 1.559000e+03 8.646669e-01]
 [1.730000e+02 3.060000e+02 8.029426e-01]]
```

```
>>> res = s.t_masked
```

```
>>> plt.figure(figsize=(15,3))
>>> plt.plot(res.T)
>>> plt.xlabel("time in s")
>>> plt.ylabel("ECG in mV")
>>> plt.tight_layout()
>>> plt.show()
```



1.3.3 Feature Inclusion

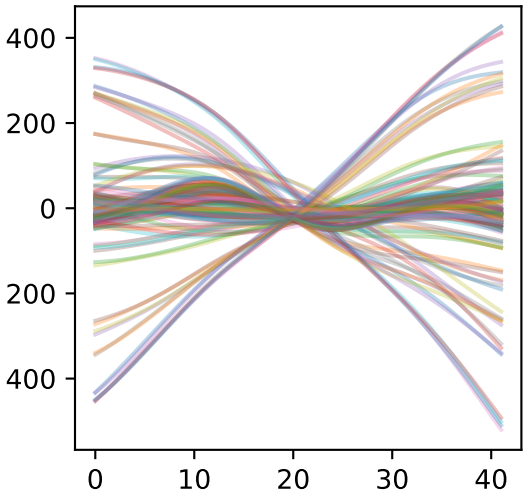
There may be a case where an original dataset has multiple features, but only a subset of these features are wanted to be included in the segmentation process.

```
>>> import numpy as np
>>> import matplotlib.pylab as plt
>>> import seg1d
```

```
>>> #retrieve the sample reference, target, and weight data
>>> r,t,w = seg1d.sampleData(c=0.5)
```

Note: The reference data shown here is centered at 0 on the y axis (vertical). As the algorithm process is based on the shape of the curve, it is irrelevant what this offset is.

```
>>> # plot reference data
>>> plt_r = np.asarray( [ x for y in r for x in y.values()  ]  ).T
>>> plt.figure(figsize=(3,3))
>>> plt.plot(plt_r,alpha=0.3)
>>> plt.show()
```

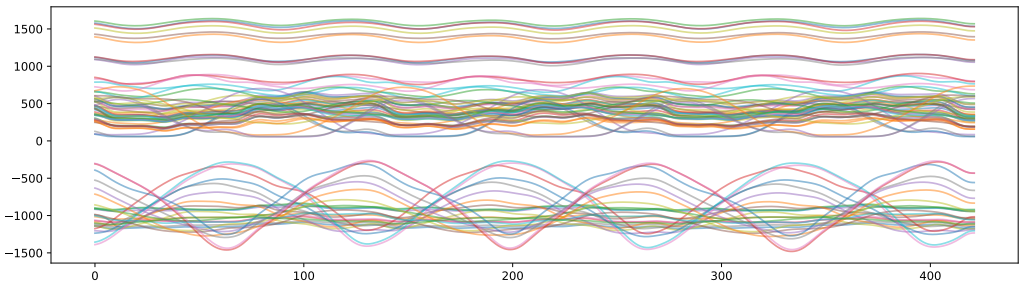


```
>>> # plot target data
>>> plt_t = np.asarray( [ x for x in t.values() ] )
>>> plt.figure(figsize=(15,4))
```

(continues on next page)

(continued from previous page)

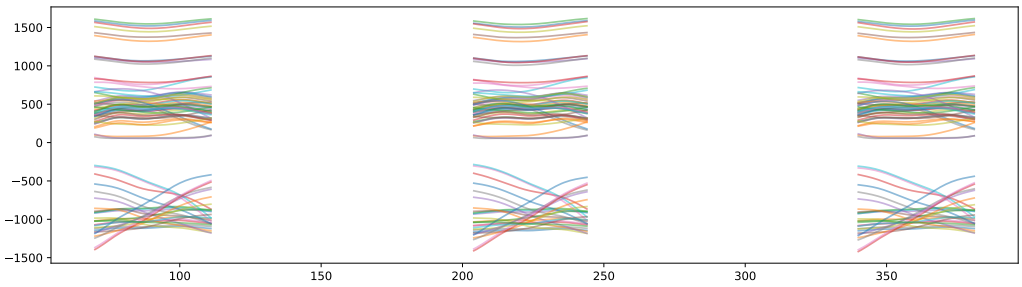
```
>>> plt.plot(plt_t.T,alpha=0.5)
>>> plt.show()
```




```
>>> #Make an instance of the segmenter
>>> s = seg1d.Segmenter()
>>> #set scaling parameters
>>> s.minW,s.maxW,s.step = 98, 105, 1
>>> #Set target and reference data
>>> s.t, s.r, s.w = t,r,w
>>> #call the segmentation algorithm
>>> segments = s.segment()
>>> print(np.around(segments,decimals=7))
[[204.        245.          0.7128945]
 [ 70.        112.          0.6670482]
 [340.        382.          0.6630886]]
```

```
>>> plt_t = s.t_masked #get a NaN masked array of the target data
```

```
>>> # plot masked target
>>> plt.figure(figsize=(15,4))
>>> plt.plot(plt_t.T,alpha=0.5)
>>> plt.show()
```



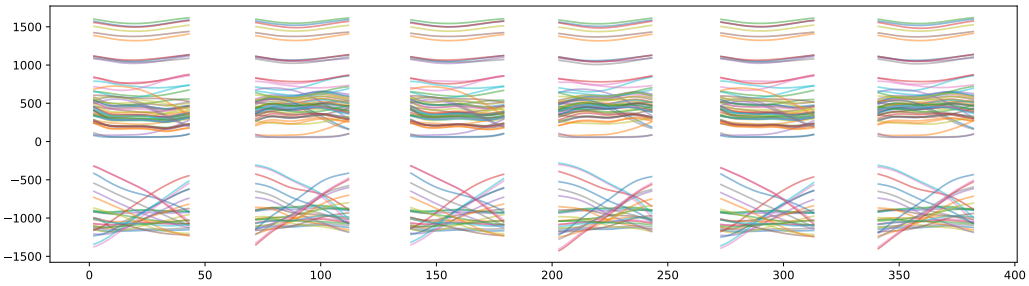
To use a subset of the features, the weights can be redefined, which may result in a different segmentation result

```
>>> sub = [('C7','z'),('T10','z'),('CLAV','z')]
>>> s.w = { x: w[x] for x in sub }
>>> segments = s.segment()
```

```
>>> print(np.around(segments,decimals=7))
[[  2.          44.           0.9648465]
 [341.         383.           0.9646419]
 [203.         244.           0.9644605]
 [273.         314.           0.9640178]
 [ 72.         113.           0.9632458]
 [139.         180.           0.9624551]]
```

```
>>> plt_t = s.t_masked #get a NaN masked array of the target data
```

```
>>> # plot masked target
>>> plt.figure(figsize=(15,4))
>>> plt.plot(plt_t.T,alpha=0.5)
>>> plt.show()
```



1.3.4 Parameter Tuning

In some cases, the default values used in the segmentation process do not result in the desired results. In this case, the various parameters that are invovled can be manually set by the user. These parameters are all defined and available through the `seg1d.Segmenter()` class.

In this example, the attributes of the segmentation algorithm will be demonstrated through a sine wave with added noise. In this example, the seed used for the random noise is the same in both the target and reference, although a different SNR is used.

First we import `seg1d`, a helper function for adding noise in the example called `segnoise`, and the plotting utils from `matplotlib`.

```
>>> import seg1d
>>> import numpy as np
>>> import matplotlib.pylab as plt
>>> import seg1d.examples.noise as segnoise
```

Next an array of data is generated and a sine wave is created. A signal-noise ratio of 30 is added to the sine wave.

```
>>> # create an array of data
>>> x = np.linspace(-np.pi*2, np.pi*2, 2000)
>>> # get an array of data from a sin function
>>> targ = np.sin(x)
>>> # add noise to the signal
```

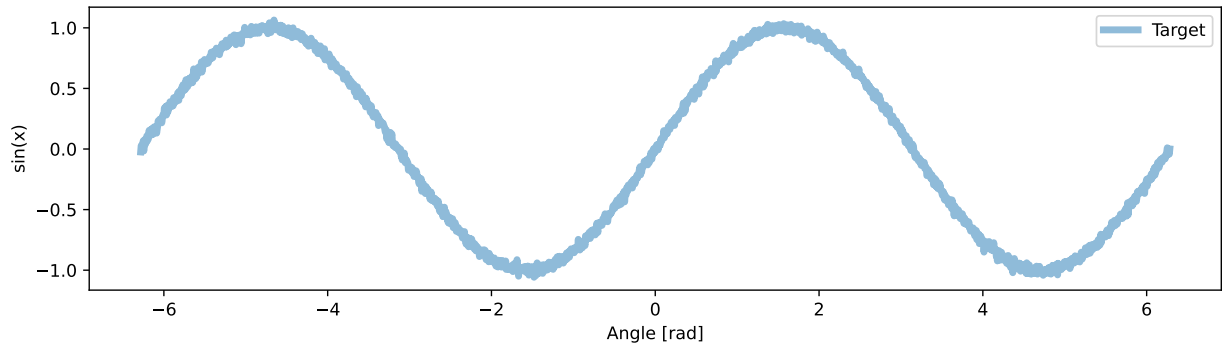
(continues on next page)

(continued from previous page)

```
>>> np.random.seed(123)
>>> targ = segnoise.add_noise(targ,snr=30)
```

The target data that is used for finding segments in looks like:

```
>>> # Plot the target
>>> plt.figure(figsize=(10,3))
>>> plt.plot(x, targ,linewidth=4,alpha=0.5,label='Target')
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



Now another noisy sine wave is created and a segment of it is cut out.

```
>>> # define a segment within the sine wave to use as reference
>>> t_s,t_e = 200,400
>>> # number of reference datasets to generate for the example
```

```
>>> # make reference data with different random noise on a segment of the original
>>> np.random.seed(123)
>>> refData = segnoise.add_noise(np.sin(x),snr=45)[t_s:t_e]
```

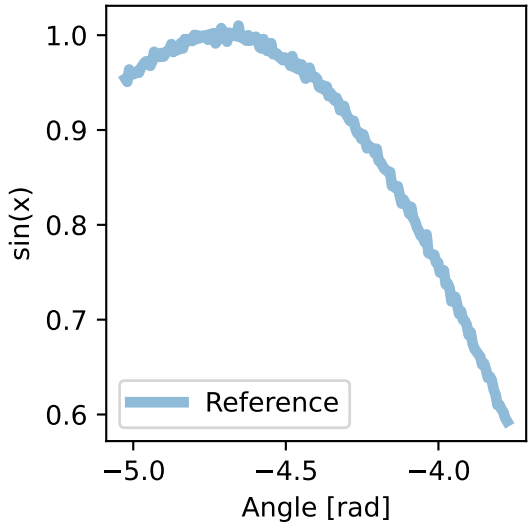
The reference data looks like:

```
>>> plt.figure(figsize=(3,3))
>>> # Plot the reference
>>> plt.plot(x[t_s:t_e], refData,linewidth=4,alpha=0.5,label='Reference')
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

To find the sub-series segment, an instance of the `Segmenter` class is created, basic scaling parameters, and the target and reference data are assigned.

```
>>> # Make an instance of the segmenter
>>> s = seg1d.Segmenter()
```

(continues on next page)



(continued from previous page)

```
>>> #set scaling parameters
>>> s.minW,s.maxW,s.step = 90, 110, 1
>>> #Set target and reference data
>>> s.set_target(targ)
>>> s.add_reference(refData)
>>> #call the segmentation algorithm
>>> segments = s.segment()
>>> np.around(segments, decimals=7)
array([[1.200000e+03, 1.420000e+03, 9.916268e-01],
       [2.000000e+02, 4.000000e+02, 9.904041e-01],
       [4.000000e+02, 5.820000e+02, 8.933443e-01],
       [1.421000e+03, 1.601000e+03, 8.833249e-01]])
```

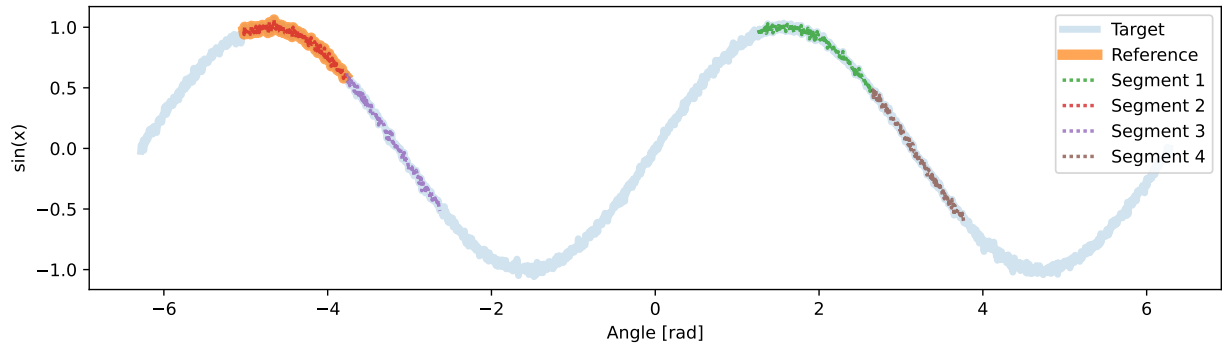
After running the segmentation algorithm, we plot the segment the reference data should be located, along with the segments that were found.

```
>>> plt.figure(figsize=(10,3))
>>> #plot the full sine wave
>>> plt.plot(x, targ,linewidth=4,alpha=0.2,label='Target')
>>> #plot the location of the original reference segment
>>> # NOTE this is just the location, the actual reference data is shown above
>>> plt.plot(x[t_s:t_e], targ[t_s:t_e],linewidth=6,alpha=0.7,label='Reference')
>>> #plot all segments found
>>> seg_num = 1
>>> for seg in segments:
...     st = seg[0]
...     e = seg[1]
...     plt.plot(x[st:e], targ[st:e],dashes=[1,1],linewidth=2,alpha=0.8,
...     label='Segment {}'.format(seg_num))
...     seg_num += 1
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
```

(continues on next page)

(continued from previous page)

```
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



From the plot, it is clear there are segments that do not belong. By accessing the `Segmenter` attributes, the algorithm and this error are better understood (and resolved).

```
>>> # First we look at the original segments before clustering
>>> np.around(s.groups, decimals=7)
array([[1.200000e+03, 1.420000e+03, 9.916268e-01],
       [2.000000e+02, 4.000000e+02, 9.904041e-01],
       [4.000000e+02, 5.820000e+02, 8.933443e-01],
       [1.421000e+03, 1.601000e+03, 8.833249e-01],
       [5.830000e+02, 7.650000e+02, 7.286635e-01],
       [1.602000e+03, 1.782000e+03, 6.541974e-01]])
```

As shown in the output, there are a total of 6 segments found before clustering.

As the distribution of segments is apporx. [0.99,0.99,0.89,0.88,0.72,0.65], the attribute, `Segmenter.cAdd`, (defaults to 0.5) that is added for forcing clusters only combines the last two values, 0.72 and 0.65 in the lower cluser.

Modifying this attribute would then change the clusters, for example:

```
>>> s.cAdd = 0.8
>>> np.around(s.segment(), decimals=7)
array([[1.200000e+03, 1.420000e+03, 9.916268e-01],
       [2.000000e+02, 4.000000e+02, 9.904041e-01]])
```

If the attribute is removed, then only the original segments are used in the clustering. However, this results in the same cluster as the original where the default of `cAdd` was 0.5.

```
>>> s.cAdd = None
>>> np.around(s.segment(), decimals=7)
array([[1.200000e+03, 1.420000e+03, 9.916268e-01],
       [2.000000e+02, 4.000000e+02, 9.904041e-01],
       [4.000000e+02, 5.820000e+02, 8.933443e-01],
       [1.421000e+03, 1.601000e+03, 8.833249e-01]])
```

Alternatively, the minimum correlation for a given segment can be set with the `Segmenter.cMin` attribute.

```
>>> s.cMin = 0.9
>>> np.around(s.segment(),decimals=7)
array([[1.200000e+03, 1.420000e+03, 9.916268e-01]])
```

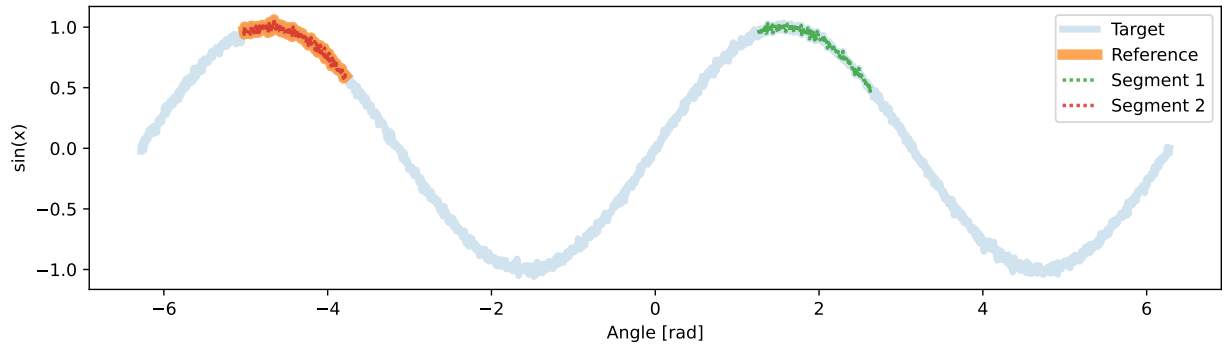
Since the `cAdd` was removed, the only segments available (higher than 0.9 correlation) were both 0.99, making the clustering result in a single segment.

If `cAdd` is set back to the default, the segment is correct.

```
>>> s.cAdd = 0.5
>>> segments = s.segment()
>>> np.around(segments, decimals=7)
array([[1.200000e+03, 1.420000e+03, 9.916268e-01],
       [2.000000e+02, 4.000000e+02, 9.904041e-01]])
```

Finally, plotting these segments shows the alignment and logical sub-series identification.

```
>>> plt.figure(figsize=(10,3))
>>> #plot the full sine wave
>>> plt.plot(x, targ,linewidth=4,alpha=0.2,label='Target')
>>> #plot the original reference segment
>>> plt.plot(x[t_s:t_e], targ[t_s:t_e],linewidth=6,alpha=0.7,label='Reference')
>>> #plot all segments found
>>> seg_num = 1
>>> for seg in segments:
...     st = seg[0]
...     e = seg[1]
...     plt.plot(x[st:e], targ[st:e],dashes=[1,1],linewidth=2,alpha=0.8,
...     label='Segment {}'.format(seg_num))
...     seg_num += 1
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```



1.3.5 Feature Processing

If there is a series of references that have different lengths, or the weighting is unknown, this example can help guide users to process the data.

In this example we use a few utilities provided by *seg1d* to process raw data. In many of the other examples, a reference set, target data, and weighting is assumed or provided as sample. If you have data from multiple sources, perhaps of different lengths of time (or samples of points) or varying features, the data must be rescaled and parsed for the matching features. This example also shows how to generate the feature weights, which will compare a set of references and find the features most simliar between them (as a normalized sum of all features).

First we import numpy and matplotlib (to plot for the example)

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Then import the base seg1d and the seg1d processing module

```
>>> import seg1d
>>> import seg1d.processing as proc
```

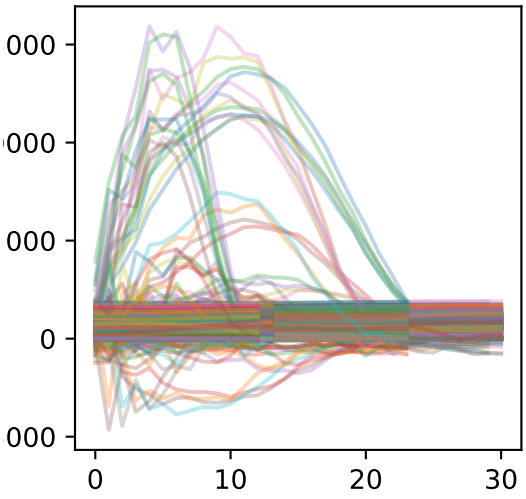
########################## SEG1D Processing ######################

To start, we load an array of dictionaries provided as sample data in seg1d. raw_r is the set of references, and raw_t is a set of targets

```
>>> raw_r, raw_t = seg1d.sample_input_data()
```

First we can take a look at the reference datasets, noticing they are all different lengths

```
>>> plt.figure(figsize=(3,3))
>>> for r in raw_r:
...     plt_r = np.asarray( [ x for x in r.values()  ]  ).T
...     plt.plot(plt_r,alpha=0.3)
>>> plt.show()
```



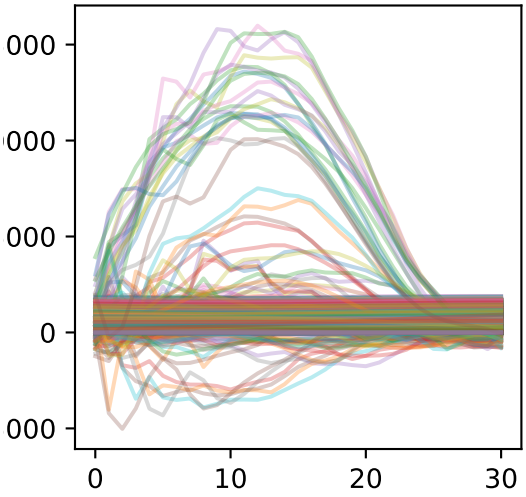
Match feature length and center time series

The first function we use for pre-processing the data is *match_len*. This will take the largest array of all reference datasets and rescale the other datasets to this longer size

```
>>> ref_data = proc.Features.match_len(raw_r)
```

We can now plot the length-matched datasets to see how the reference segments do show similarity over the same scaled timespan

```
>>> plt_r = np.asarray( [ x for y in ref_data for x in y.values()  ]  ).T
>>> plt.figure(figsize=(3,3))
>>> plt.plot(plt_r,alpha=0.3)
>>> plt.show()
```



This next process is not always necessary, but is convenient to know. It simply shifts the mean of each array in the references to center along the 0 axis.

```
>>> ref_data = proc.Features.center(ref_data)
```

Make sure all reference data has valid keys In this step, we use the *shared* function by passing all the datasets we will will use to ensure no feature keys are missing. This requires a dictionary to map each feature correctly

```
>>> ref_data, tar_data = proc.Features.shared(ref_data, raw_t)
```

Generate weights from the segmented data One of the most important steps for many use cases is the automatic weighting of multiple features. The *gen_weights* function takes a list of reference datasets and finds the features that match best among all of the references. It then normalizes all the results from 0 to 1. This is important to note, as it means some features will not be used. If all of your features are necessary or should be used, this function should be skipped.

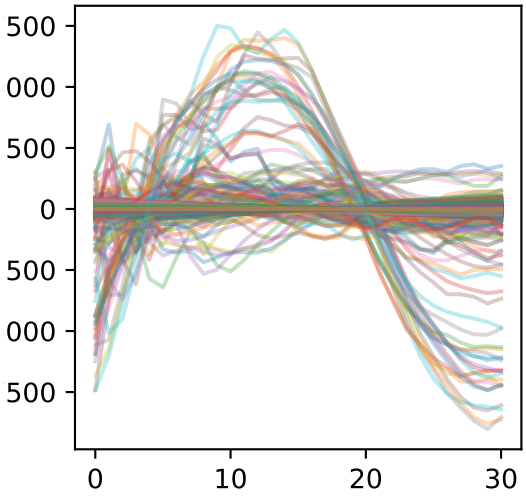
```
>>> weights = proc.Features.gen_weights(ref_data)
```

Finally, the last step of pre-processing is to get which params to use and limit to specific ones. In this example, we take only the top 20 features, limit to the normalized score of 0.8, and pass an empty list, which means we use all available features. If there was a list of features that you wanted to include, passing this key list of strings would be possible.

```
>>> weight_dict = proc.Features.meaningful(weights, limit=0.8, top=20, include_keys=[])
```

Now we can take a look at the reference data that we will use for segmentation. This data has also been centered (a few steps before)

```
>>> plt_r = np.asarray( [ x for y in ref_data for x in y.values()  ]  ).T
>>> plt.figure(figsize=(3,3))
>>> plt.plot(plt_r,alpha=0.3)
>>> plt.show()
```



########################## SEG1D Analysis ########################

At this point the data is properly processed and features weighted. From here on, the normal segmentation process (as seen in the other examples) is applied.

```
>>> s = seg1d.Segmenter()
>>> s.minW, s.maxW, s.step = 70, 150, 1
```

```
>>> for r in ref_data:
...     s.add_reference(r)
```

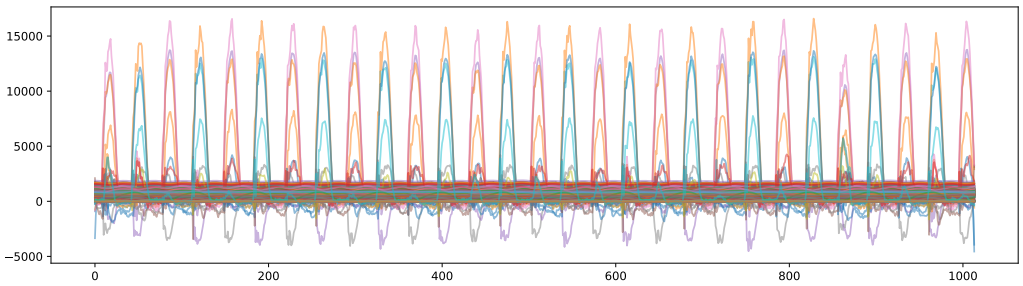
As this sample dataset has multiple targets, we will just use the first one. However, you could iterate over all the target trials by wrapping the remaining code under this example loop: for idx, target in enumerate(tar_data):

Run the analysis on the first target data.

```
>>> target = tar_data[0]
```

We can visualize this target data before segmentation

```
>>> plt_t = np.asarray( [ x for x in target.values() ] )
>>> plt.figure(figsize=(15,4))
>>> plt.plot(plt_t.T,alpha=0.5)
>>> plt.show()
```

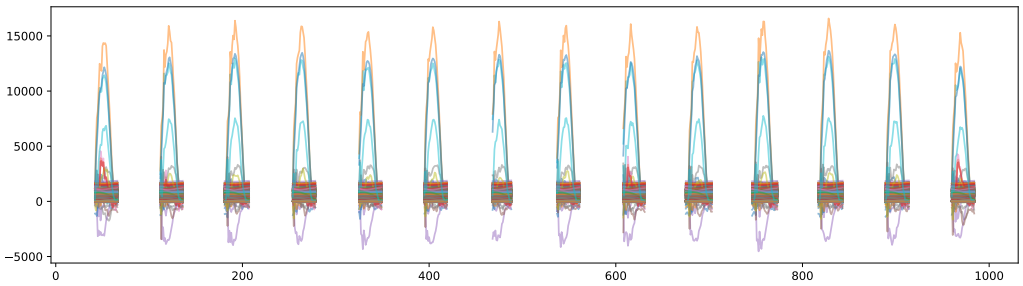



```
>>> s.set_target(target)
```

```
>>> s.w = weight_dict
>>> segments = s.segment()
>>> print(np.around(sorted(segments, key=lambda x: x[0])[:3], decimals=5))
[[ 42.       67.        0.99856]
 [112.      137.        0.99864]
 [181.      208.        0.99709]]
```

Now that we have segmented the data, the masked array can be plotted to show the results.

```
>>> plt_t = s.t_masked #get a NaN masked array of the target data
>>> # plot masked target
>>> plt.figure(figsize=(15,4))
>>> plt.plot(plt_t.T,alpha=0.5)
>>> plt.show()
```



See Also

seg1d.Segmenter

1.4 Code Reference

1.4.1 Simple Interface

This method is shown in the getting started section of the documentation. It provides an easy-to-use interface for segmenting data, although it has limited functionality in fine-tuning parameters of the underlying algorithms.

Simple Interface

seg1d.**segment_data**(*r*, *t*, *w*, *minS*, *maxS*, *step*)

Segmentation manager for interfacing with Segmenter class

Find segments of a reference dataset in a target dataset using a rolling correlation of *n* number of reference examples with a peak detection applied to the average of *m* reference features with weights applied to each feature.

Parameters

r [List[Dict[key,numpy.array]]] reference data of form `[ {(feature Key): [data array] }, {(feature Key): [data array] } ]`

t [Dict[key,numpy.array]] target data of form `{ (feature Key): [data array] }`

w [Dict[key,float] or None] Weights of form `{ (feature key):float,(feature key):float }`

minS [int] Minimum scale to apply for reference data

maxS [int] Maximum scale to apply for reference data

step [int] Size of step to use in rolling correlation

Returns

***3 x n* array** segments of form `[start of segment,end of segment,correlation score]`

Examples

```
>>> import numpy as np
```

First we import sample data from the examples folder that has multiple features derived from motion capture data

```
>>> import seg1d
>>> r,t,w = seg1d.sampleData()
```

Then we define some segmentation parameters such as the scaling percentage of the reference data and index stepping to use in rolling correlation

```
>>> minW = 70 # percent to scale down reference data
>>> maxW = 150 # percent to scale up reference data
>>> step = 1 #step to use for correlating reference to target data
```

Finally we call the segmentation algorithm

```
>>> np.around(seg1d.segment_data(r,t,w,minW,maxW,step),7)
array([[207.       , 240.       ,   0.9124224],
       [342.       , 381.       ,   0.8801901],
       [ 72.       , 112.       ,   0.8776795]])
```

1.4.2 Class

The Class reference should be used for any advanced usage or when needing convenience functions for adding data and tuning parameters of the segmentation process.

Segmenter Class

Segmenter	Segmentation class that exposes all algorithm parameters and attributes for advanced access and tuning of segmentation.

seg1d.Segmenter

class seg1d.**Segmenter**

Segmentation class that exposes all algorithm parameters and attributes for advanced access and tuning of segmentation.

Additional convenience methods for adding reference and target data as numpy arrays are provided.

Results of each step of the algorithm process can be accessed through the class Attributes after running the segmentation. These can likewise be passed to the algorithms methods described in the documentation.

Examples

Simple usage of the class by directly assigning attributes using sample data included with this package.

```
>>> import seg1d
>>> import numpy as np
>>>
>>> #Make an instance of the segmenter
>>> s = seg1d.Segmenter()
>>>
>>> #retrieve the sample reference, target, and weight data
>>> s.r,s.t,s.w = seg1d.sampleData()
>>>
>>> #set the parameters
>>> s.minW,s.maxW,s.step = 70, 150, 1
>>>
>>> np.around(s.segment(), decimals=7)
array([[207.       , 240.       ,   0.9124224],
       [342.       , 381.       ,   0.8801901],
       [ 72.       , 112.       ,   0.8776795]])
```

__init__()

Initialization of segmentation class and parameters

Attributes

r [array of dicts] The reference dataset

t [dict] The target dataset

w [dict] Weights for correlation

minW [int] minimum percent to scale data

maxW [int] maximum percent to scale data

step [int] step size for rolling correlation

wSizes [list] sizes to use for resampling reference (can be used instead of minW,maxW,step)

cMax [bool] use maximum in rolling correlation (default False)

cMin [float] -1 to 1, min correlation

cAdd [float] 0 to 1 or None, value to add for forcing clusters (Default 0.5)

pD [None] peak distance to use for scipy peak detection (Default None)

nC [int] number of clusters for correlation results

fMode [{'w', 'm', 's'}] keyword to use for aggregating feature correlations (default *w*). Options, w=weighted mean, m=mean, s=sum

fScale [bool] scale the feature correlation by its weight before feature aggregation (Default True)

tSeg [[]] the target data as segmented arrays

Methods

__init__()	Initialization of segmentation class and parameters
add_reference(r[, copy])	Appends a reference containing one or more features to the existing reference dataset.
clear_reference()	Removes any reference data currently assigned
segment()	Method to run the segmentation algorithm on the current Segmenter instance
set_target(t[, copy])	Sets the target data by overiding any existing target.

Attributes

clusters	Segments reduced by clustering algorithm from *algorithm.cluster()*
combined	The averaged correlation of the rolling feature correlation and the weighting table created by *algorithm.combine_corr()*
corrs	Rolling correlation of reference and target features created by *algorithm.rolling_corr()*
groups	Possible segments through parsing overlapping segment locations defined by *algorithm.uniques()*
peaks	Peaks of the correlations created by *algorithm.get_peaks()*
t_masked	The target data as ndarray masked with the non-defined segments as NaNs.
t_segments	Returns an array of segmented target data

—

Segmenter Methods

These methods are used to handle data and run the segmentation process.

set_target(t[, copy])	Sets the target data by overiding any existing target.
add_reference(r[, copy])	Appends a reference containing one or more features to the existing reference dataset.
clear_reference()	Removes any reference data currently assigned
segment()	Method to run the segmentation algorithm on the current Segmenter instance

seg1d.Segmenter.set_target

Segmenter.**set_target**(*t*, *copy=True*)

Sets the target data by overiding any existing target. If the target is not a dict, it will be converted to one.

Parameters

t [dict or ndarray]

Dictionary containing labeled features as keys and values as 1-D arrays (must be same size).
ndarray of dimension 1 will be used as a single feature for the target.
ndarray of n-dimensions will use rows as unique features.

copy [bool, optional] If True, will make a deepcopy of the passed parameter (Default True)

Returns

None

See also:

add_reference Add a reference item

Notes

This is the recommended method for adding a feature. You can also set the target directly through the Attribute *t* by `Segmenter.t = ` however, this method ensures the data labels and length or stored properly. Setting *t* directly must be done with a dictionary.

Examples

Target data can be set to a single numpy array.

```
>>> import numpy as np
>>> import seg1d
>>>
>>> s = seg1d.Segmenter()
>>> t = np.linspace(0,1,4)
>>> s.set_target(t)
>>> s.t
{'0': array([0.        , 0.33333333, 0.66666667, 1.        ])}
```

Alternatively, you can pass a 2-dimensional array representing multiple features.

```
>>> s = seg1d.Segmenter()
>>> t = np.linspace(0,1,6).reshape(2,3)
>>> s.set_target(t)
>>> s.t
{'0': array([0. , 0.2, 0.4]), '1': array([0.6, 0.8, 1. ])}
```

seg1d.Segmenter.add_reference

Segmenter.**add_reference**(*r*, *copy=True*)

Appends a reference containing one or more features to the existing reference dataset. If the reference is not a dict, it will be converted to one. If this should be the only reference set, use `clear_reference()` before calling this method.

Parameters

r [dict or ndarray]

Dictionary containing labeled features as keys and values as 1-D arrays (must be same size).
ndarray of dimension 1 will be used as a single feature for the reference.
ndarray of n-dimensions will use rows as unique features.

copy [bool, optional] If True, will make a deepcopy of the passed parameter (Default True).

See also:

set_target Set the target data

clear_reference Clear the current reference data

Notes

This method allows features that are not in previous references to be added, and vice-versa. It will also allow different sizes of reference data to be added. This is done as you can explicitly declare which features to use when segmenting.

Examples

Add a reference with multiple features

```
>>> import seg1d
>>> import numpy as np
>>>
>>> s = seg1d.Segmenter()
>>> r = np.linspace(0,1,6).reshape(2,3)
>>> s.add_reference( r )
>>> s.r
[{'0': array([0. , 0.2, 0.4]), '1': array([0.6, 0.8, 1. ])}]
```

Alternatively, each row of the array can be added as the same labeled feature for different references by calling this method in a loop. Notice this is now an array of dictionaries containing the same feature label.

```
>>> s = seg1d.Segmenter()
>>> r = np.linspace(0,1,6).reshape(2,3)
>>> for _r in r: s.add_reference(_r)
>>> s.r
[{'0': array([0. , 0.2, 0.4])}, {'0': array([0.6, 0.8, 1. ])}]
```

seg1d.Segmenter.clear_reference

Segmenter.**clear_reference**()

Removes any reference data currently assigned

Parameters

None

Returns

None

See also:

add_reference Add a reference item

Notes

This method also clears the *rF*, and *rLen* attributes.

Examples

```
>>> import numpy as np
>>> import seg1d
>>>
>>> s = seg1d.Segmenter()
>>> s.add_reference( np.linspace(0,3,3) )
>>> s.r
[{'0': array([0. , 1.5, 3. ])}]
>>> s.clear_reference()
>>> s.r
[]
```

seg1d.Segmenter.segment

Segmenter.**segment**()

Method to run the segmentation algorithm on the current Segmenter instance

Parameters

None

Returns

***3 x n* array** segments of form `[start of segment,end of segment,correlation score]`

Examples

This example is the same as the main `Segmenter` class as it is the interface method.

```
>>> import numpy as np
>>> import seg1d
>>>
>>> #Make an instance of the segmenter
>>> s = seg1d.Segmenter()
>>>
>>> #retrieve the sample reference, target, and weight data
>>> s.r,s.t,s.w = seg1d.sampleData()
>>>
>>> #set the parameters
>>> s.minW,s.maxW,s.step = 70, 150, 1
>>>
>>> np.around(s.segment(),7)
array([[207.       , 240.       ,   0.9124224],
       [342.       , 381.       ,   0.8801901],
       [ 72.       , 112.       ,   0.8776795]])
```

See Also

seg1d.Segmenter

Features Class

`match_len`(dataset)	interpolate to the maximum sized data in the reference set
`center`(ref_data)	subtract the mean of each feature from itself
`shared`(D1[, D2])	get data with only features shared amongst all sets
`gen_weights`(dataset)	Create weight table from reference data
`meaningful`(weights[, limit, top, include_keys])	get a weight table of the most meaningful features

1.4.3 Methods

The Methods reference explains the individual routines that are used in the segmentation process. Although they are accessible directly, it is likely easier to (and recommended) to use the Class.

Algorithm Methods

General Functions

rolling_corr(x, yData, winSize[, cMax])	Rolling Correlation
combine_corr(x, w[, method, scale])	Combines Weighted Correlation
uniques(sortedPeaks, srcLen)	Unique Segment Identification
get_peaks(x[, minC, dst])	Peak Detection

continues on next page

Table 7 – continued from previous page

cluster(segGroups[, segAdder, nClust])	Clustering
resample(x, s)	Interpolation

seg1d.algorithm.rolling_corr

seg1d.algorithm.**rolling_corr**(*x*, *yData*, *winSize*, *cMax=False*)

Rolling Correlation

Calculates the rolling correlation coefficient over the given window sizes

Parameters

x [1-D array] array of target data

yData [2-D array] array of reference data

winSize [int] scale of the that the reference data should be rescaled to

Returns

ndarray 1-D array of length (size(x) - winSize + 1)

Other Parameters

cMax [bool, optional] Use maximum of correlations (Default False)

Warning:

The reference data (yData) must be smaller than the target data (x) AFTER resampling.
This means if the reference data is length 80, and the target data is length 100, it will work. However, if the winSize is supposed to be length 120, the reference will be scaled and correlation will crash.

See also:

combine_corr (takes the return of this function)

Examples

```
>>> import numpy as np
>>> import seg1d.algorithm as alg
```

```
>>> #make waves
>>> x = np.sin( np.linspace(-np.pi*1, np.pi*1, 20) )
>>> y = np.sin( np.linspace(-np.pi*2, np.pi*2, 80) ).reshape(4,20)
```

```
>>> #apply rolling correlations with 10 and 15
>>> alg.rolling_corr(x, y, 10 )
array([-0.00766151,  0.02078156,  0.03501678,  0.04019572,  0.04211895,
        0.04262637,  0.04211895,  0.04019572,  0.03501678,  0.02078156,
       -0.00766151])
>>> alg.rolling_corr(x, y, 15 )
array([0.03321832, 0.03972237, 0.04254858, 0.04254858, 0.03972237,
       0.03321832])
```

seg1d.algorithm.combine_corr

seg1d.algorithm.**combine_corr**(*x*, *w*, *method='m'*, *scale=True*)

Combines Weighted Correlation

Takes in the correlated data results and multiply the weighting values to each array of data for that feature. | Combines the results of the weighted features

Parameters

x [Dict[int,Dict[string,numpy.array]]] `{scale:{ feature: array([correlations]) } }`

w [Dict[string,float]] `{ feature: weight }`

method [{'m','w', 's'}] keyword to use for aggregating feature correlations (default *m*). Options, m=mean, w=weighted mean, s=sum

scale [bool, optional] keyword argument for scaling the correlated feature before applying any of the aggregation methods

Returns

Dict[int,numpy.array] `{scale: array([weighted correlations]) }`

See also:

rolling_corr (input for this function)

get_peaks (takes the return of this function)

Examples

```
>>> import random
>>> import numpy as np
>>> import seg1d.algorithm as alg
```

```
>>> #make a convenience function to get a wave for sample data
>>> def s(f1, f2, f3): return np.sin( np.linspace(f1, f2, f3) )
```

```
>>> x = {
...     10: {'a': s(-np.pi*0.8, 0, 10), 'b': s(0, np.pi*0.8, 10)},
...     20: {'a': s(-np.pi*0.7, 0, 10), 'b': s(0, np.pi*0.7, 10)}
... }
```

Assign some weights and find the averaged value

```
>>> w = { 'a': 0.5, 'b': 0.9 }
>>> a = alg.combine_corr(x, w )
>>> for k,v in a.items(): print(k,v)
10 [-0.14694631 -0.07296588  0.00666771  0.0857847   0.15825538  0.21846498
  0.26174865  0.28475292  0.2856955   0.26450336]
20 [-0.20225425 -0.12293111 -0.03630481  0.0524783   0.13814375  0.21560229
  0.2802522   0.32825274  0.35675226  0.36405765]
```

Change the weight values and see the weighted scores change

```
>>> w = { 'a': 0.9, 'b': 0.2 }
>>> a = alg.combine_corr(x, w )
>>> for k,v in a.items(): print(k,v)
10 [-0.26450336 -0.3270411  -0.36424081 -0.37322037 -0.35328408 -0.30597655
 -0.23496298 -0.14574528 -0.04523573  0.05877853]
20 [-0.36405765 -0.39304054 -0.39867347 -0.38062179 -0.33995792 -0.27909765
 -0.20165658 -0.1122354  -0.01614647  0.0809017 ]
```

seg1d.algorithm.uniques

seg1d.algorithm.**uniques**(*sortedPeaks*, *srcLen*)
Unique Segment Identification

Find unique segment(s) in a sequence of correlation values.
Guarantees segments are not overlapping

Parameters

sortedPeaks [n x 3 array] n x 3 array sorted by highest to lowest correlation of form `[ scale (int), correlation(float) , peak index (int) ]`

srcLen [int] length of the target data, used to block out possible segments

Returns

n x 3 array `[ start index, end index, correlation ]`

None if no segments are found

See also:

get_peaks (input for this function)

cluster (takes in the return of this function)

Examples

```
>>> import numpy as np
>>> import seg1d.algorithm as alg
```

```
>>> p = [ [10, 0.90, 7  ],
...      [10, 0.89, 8  ],
...      [20, 0.80, 20 ],
...      [25, 0.70, 40 ],
...      ]
```

```
>>> el = 50
```

```
>>> alg.uniques(p,el)
[[7, 17, 0.9], [20, 40, 0.8], [40, 65, 0.7]]
```

seg1d.algorithm.get_peaks

seg1d.algorithm.**get_peaks**(*x*, *minC=0.7*, *dst=None*)

Peak Detection

Find the peaks of a data array with a minimum value of a peak and an optional distance parameter.

Relies on `scipy.signal.find_peaks`

Parameters

x [Dict[int,List[float]]] `{scale: [correlations] }`

Returns

n x 3 array sorted by highest to lowest correlation of form `[ scale, correlation , peak index ]`

Other Parameters

minC [float, optional] -1 to 1

dst [real, optional] int or float

See also:

combine_corr (input for this function)

uniques (takes the return of this function)

Examples

```
>>> import numpy as np
>>> import seg1d.algorithm as alg
```

```
>>> # convenience function for generating wave
>>> def s(f1, f2, f3): return np.sin( np.linspace(f1, f2, f3) )
```

Define some scales that have correlations

```
>>> x = { 10: s(-np.pi*1, np.pi*1, 10), 20: s(-np.pi*2, np.pi*2, 10) }
```

Query the peaks in the data

```
>>> np.around(alg.get_peaks(x), decimals=7)
array([[10.       ,  0.9848078,  7.       ],
       [20.       ,  0.9848078,  1.       ],
       [20.       ,  0.8660254,  6.       ]])
```

Define a minimum for the peak

```
>>> np.around(alg.get_peaks(x,minC = 0.9), decimals=7)
array([[10.       ,  0.9848078,  7.       ],
       [20.       ,  0.9848078,  1.       ]])
```

seg1d.algorithm.cluster

seg1d.algorithm.**cluster**(*segGroups*, *segAdder=0.5*, *nClust=2*)

Clustering

Clusters segments based on correlation values

Parameters

segGroups [n x 3 array] `[ [ start index, end index, correlation ] ]`

segAdder [float, optional] 0.0 to 1.0 or None If not None, the value that is added to the cluster groups to force a correlation cluster of the highest values

Returns

n x 3 array `[start segment, end segment, correlation score of segment]`

Other Parameters

nClust [int, optional] number of clusters to group data in (Default 2)

If `nClust=0`, returns segGroups

Warns

Segment Adder value was included in final cluster. This may mean cluster is poorly defined or Adder is too high. It is removed before being returned. However, it may be a sign of poor clustering settings as the intention of the segment adder is to force clustering of highly similar segments by creating a lower group (therefore, it should not be in the high cluster group).

See also:

uniques (input for this function)

Examples

```
>>> import numpy as np
>>> import seg1d.algorithm as alg
```

```
>>> x = [[7, 17, 0.90], [20, 40, 0.88], [40, 65, 0.8], [50, 65, 0.70]]
>>> alg.cluster(x)
[[7, 17, 0.9], [20, 40, 0.88], [40, 65, 0.8], [50, 65, 0.7]]
>>> alg.cluster(x,segAdder=None)
[[7, 17, 0.9], [20, 40, 0.88], [40, 65, 0.8]]
>>> alg.cluster(x,segAdder=0.85)
[[7, 17, 0.9], [20, 40, 0.88], [40, 65, 0.8]]
```

Note: This should raise the following warning:

UserWarning: Segment Adder value was included in final cluster. This may mean cluster is poorly defined or Adder is too high.

```
>>> alg.cluster(x,nClust=3)
[[7, 17, 0.9], [20, 40, 0.88], [40, 65, 0.8]]
>>> alg.cluster(x,segAdder=None,nClust=3)
[[7, 17, 0.9], [20, 40, 0.88]]
```

seg1d.algorithm.resample

seg1d.algorithm.**resample**(*x*, *s*)

Interpolation

Apply a cubic interpolation on an n x m dataset that is resampled to the number of samples

Parameters

x [n x m array] n-number of datasets with length m

s [int] number of samples to interpolate x

Returns

n x s array interpolated dataset

See also:

cluster (input for this function)

resample (takes in the return of this function)

Examples

```
>>> import numpy as np
>>> import seg1d.algorithm as alg
```

```
>>> x = np.sin( np.linspace(-3, 3, 10) )
>>> alg.resample(x,6)
array([[-0.14112001, -0.97319156, -0.56423116,  0.56423116,  0.97319156,
         0.14112001]])
>>> x = np.array([x,x**2])
>>> alg.resample(x,6)
array([[-0.14112001, -0.97319156, -0.56423116,  0.56423116,  0.97319156,
         0.14112001],
       [ 0.01991486,  0.94687756,  0.31972116,  0.31972116,  0.94687756,
         0.01991486]])
```

Optimized Functions

rcor(x, Y)	Correlation of multiple arrays to a single array using a rolling window correlation.
vcor(x, y)	Rolling correlation between two arrays.

seg1d.optimized_funcs.rcor

seg1d.optimized_funcs.**rcor**(*x*, *Y*)

Correlation of multiple arrays to a single array using a rolling window correlation.

Parameters

x [1d array] target array

Y [ndarray] references resampled to correct size

Returns

n x m array correlations of one ndarray to an m x ndarray

Notes

This will try to use numba for optimization.

Examples

```
>>> import numpy as np
>>> import seg1d.optimized_funcs as optF
```

```
>>> x = np.sin( np.linspace(-3, 3, 25) )
>>> y = np.sin( np.linspace(-3, 3, 60) ).reshape(3,20)
```

```
>>> optF.rcor(x,y)
array([[-0.50743663, -0.66692675, -0.78849873, -0.87803067, -0.93682968,
        -0.96013818],
       [ 0.83362263,  0.91097751,  0.94663428,  0.94663428,  0.91097751,
         0.83362263],
       [-0.96013818, -0.93682968, -0.87803067, -0.78849873, -0.66692675,
        -0.50743663]])
```

seg1d.optimized_funcs.vcor

seg1d.optimized_funcs.**vcor**(*x*, *y*)

Rolling correlation between two arrays. Optimized by numba if available

Parameters

x [1D array] array to use as static data

y [1D array] array to use as rolling data

Returns

1D array correlations at each increment

```
size = (size(x) - size(y)) + 1
```

Notes

Required: `size(x) > size(y)` This will try to use numba for optimization.

Examples

```
>>> import numpy as np
>>> import seg1d.optimized_funcs as optF
```

```
>>> x = np.sin( np.linspace(-3, 3, 25) )
>>> y = np.sin( np.linspace(-3, 3, 20) )
```

```
>>> optF.vcor(x,y)
array([0.83212194, 0.90933756, 0.94493014, 0.94493014, 0.90933756,
       0.83212194])
```

seg1d: Python module for automated 1D subsequence segmentation Copyright (C) 2020 Mathew Schwartz

1.5 Community Guidelines

1.5.1 Issues

Issues and feature requests should be submitted on github .

1.5.2 Contributing

Please follow the "fork-and-pull" Git workflow. However, it is suggested to create an issue first to confirm the contributions align with the future of the module.

1. **Fork** the repo on GitHub
2. **Clone** the project to your own machine
3. **Commit** changes to your own branch
4. **Push** your work back up to your fork
5. Submit a **Pull request** so that we can review your changes

Documentation is on the main branch of the repository (rather than gh-pages) and should be built as-is. A redirect `index.html` file handles moving the github pages to the build directory and a `.nojekyll` file preserves folder types.

1.5.3 Copyright and Licensing

Please refer to the full LICENSE text.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

C

G

M

O

R

S

U

V