wdf Documentation

Release 2

Elena Cuoco

Aug 02, 2023

Contents

The library covers the application of the Wavelet Detection Filter (WDF) on the time-series data. The core part of the wdf is based on the WDF implementation p4TSA developed by Elena Cuoco(see p4TSA github repository https://github.com/elenacuoco/p4TSA/). p4TSA can be wrapped in Python and within this library it is denoted by *pytsa*.

CHAPTER 1

Table of content

1.1 Introduction

Wavelet detection filter (WDF) is a python library which wraps some of the routines in C++ of p4TSA and its python wrapper pytsa. In WDF library you can find the WDF itself, that is a pipeline able to detect transient signal, using a wavelet decomposition of the data, followed by a denoising procedure, and later on by the estimation of the energy content of the signal. In the same library you can find the whitening, and double whitneing (equivalent to dividing by the PSD the data) based on the AutoRegressive fit, which is implemented in time domain. You can find also other functionalities linked to parametrice modeling (ARMA, AR, MA) or downsampling filter.

1.1.1 Requirements

- p4TSA
- pytsa
- numpy
- scipy (upgraded version)
- docker with p4TSA and WDF environment

1.1.2 Installation

To install the wdf library, one has to run *setup.py* script from the main directory of the library.

```
python setup.py install
```

1.1.3 Howto

For a quick start, you can have a look at the Tutorial section

1.1.4 Contacts

If you find any issues, please contact:

- Elena Cuoco: elena.cuoco@ego-gw.it
- Filip Morawski: fmorawski@camk.edu.pl
- Alberto Iess: alberto.iess@sns.it

1.2 Tutorials

The following tutorials presents few examples of the WDF and other tool usage.

1.2.1 Whitening

Whitening procedure with AutoRegressive (AR) model

author: Elena Cuoco

We can whitening the data in time domain, using the Autoregressive parameters we estimated on a given chunck of data in frame format.

Double whitening refers to the procedure applied in the time domain of data whitening, using the inverse of PSD. However, the method used in pytsa is based on the parametric estimation (AR) of the PSD and the Lattice Filter implementation in the time domain.

[1]:
```
import time
import os
import json
from pytsa.tsa import SeqView_double_t as SV
from wdf.config.Parameters import Parameters
from wdf.processes.Whitening import Whitening
from wdf.processes.DWhitening import  DWhitening
from pytsa.tsa import FrameIChannel
import logging, sys

logger = logging.getLogger()
logger.setLevel(logging.INFO)
logging.debug("info")

new_json_config_file = True    # set to True if you want to create new Configuration
if new_json_config_file==True:
    configuration = {
      "file": "./data/test.gwf",
      "channel": "H1:GWOSC-4KHZ_R1_STRAIN",
      "len":1.0,
      "gps":1167559200,
      "outdir": "./",
      "dir":"./",
      "ARorder": 1000,
      "learn": 200,
      "preWhite":4
    }
```

(continues on next page)

(continued from previous page)

```
    filejson = os.path.join(os.getcwd(),"parameters.json")
    file_json = open(filejson, "w+")
    json.dump(configuration, file_json)
    file_json.close()
logging.info("read parameters from JSON file")

par = Parameters()
filejson = "parameters.json"
try:
    par.load(filejson)
except IOError:
    logging.error("Cannot find resource file " + filejson)
    quit()

strInfo = FrameIChannel(par.file, par.channel, 1.0, par.gps)
Info = SV()
strInfo.GetData(Info)
par.sampling = int(1.0 / Info.GetSampling())
logging.info("channel= %s at sampling frequency= %s" %(par.channel, par.sampling))

whiten=Whitening(par.ARorder)
par.ARfile = "./ARcoeff-AR%s-fs%s-%s.txt" % (
                par.ARorder, par.sampling, par.channel)
par.LVfile ="./LVcoeff-AR%s-fs%s-%s.txt" % (
                par.ARorder, par.sampling, par.channel)

if os.path.isfile(par.ARfile) and os.path.isfile(par.LVfile):
    logging.info('Load AR parameters')
    whiten.ParametersLoad(par.ARfile, par.LVfile)

else:
    logging.info('Start AR parameter estimation')
    ######## read data for AR estimation###############
    strLearn = FrameIChannel(par.file, par.channel, par.learn, par.gps)
    Learn = SV()
    strLearn.GetData(Learn)
    whiten.ParametersEstimate(Learn)
    whiten.ParametersSave(par.ARfile, par.LVfile)
```

```
INFO:root:read parameters from JSON file
INFO:root:channel= H1:GWOSC-4KHZ_R1_STRAIN at sampling frequency= 4096
INFO:root:Load AR parameters
```

[2]:
```
# sigma for the noise
par.sigma = whiten.GetSigma()
logging.info('Estimated sigma= %s' % par.sigma)
```

```
INFO:root:Estimated sigma= 5.09281e-22
```

We use some chunck of data to pre-heating the whitening procedure and avoiding the filter tail.

[3]:
```
#Initialize the loop for the whitening and double whitening
data = SV()
dataw = SV()
dataww =SV()

streaming = FrameIChannel(par.file, par.channel, par.len, par.gps)
```

(continues on next page)

(continued from previous page)

```
streaming.GetData(data)
N=data.GetSize()

Dwhiten=DWhitening(whiten.LV,N,0)
if os.path.isfile(par.LVfile):
    logging.info('Load LV parameters')
    Dwhiten.ParametersLoad(par.LVfile)
###---whitening preheating---###
for i in range(par.preWhite):
    streaming.GetData(data)
    whiten.Process(data, dataw)
    Dwhiten.Process(data, dataww)
```

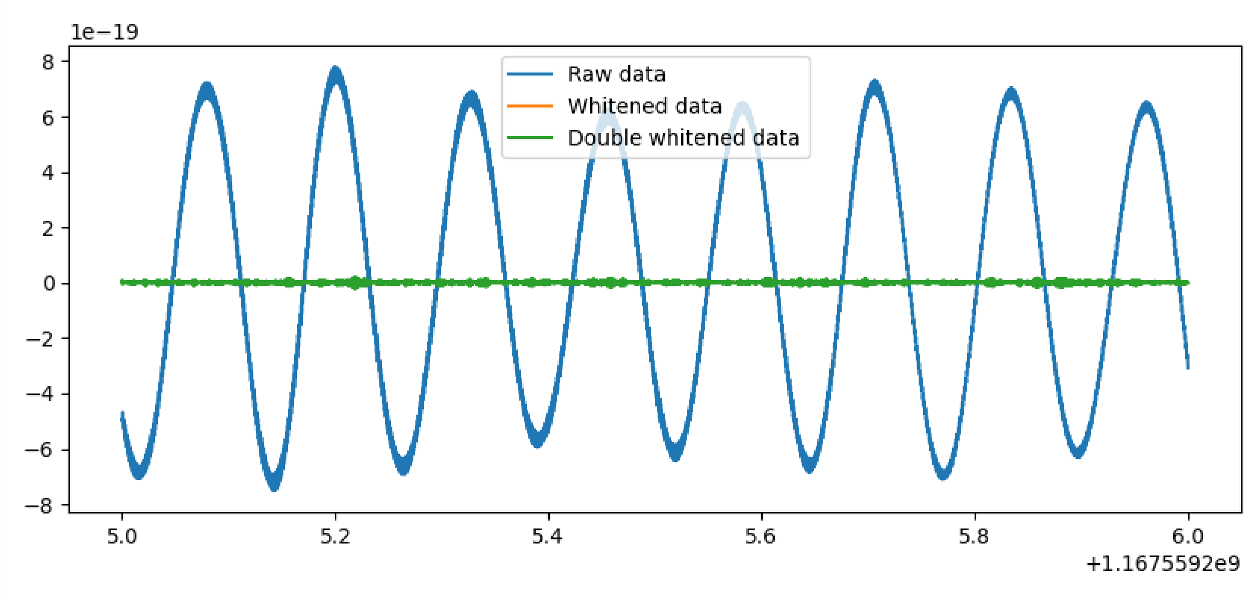
```
INFO:root:Load LV parameters
```

[4]:
```
# data to be plotted
streaming.GetData(data)
whiten.Process(data, dataw)
Dwhiten.Process(data, dataww)
```

Plot: raw, whitened and double-whitened data

Time-domain

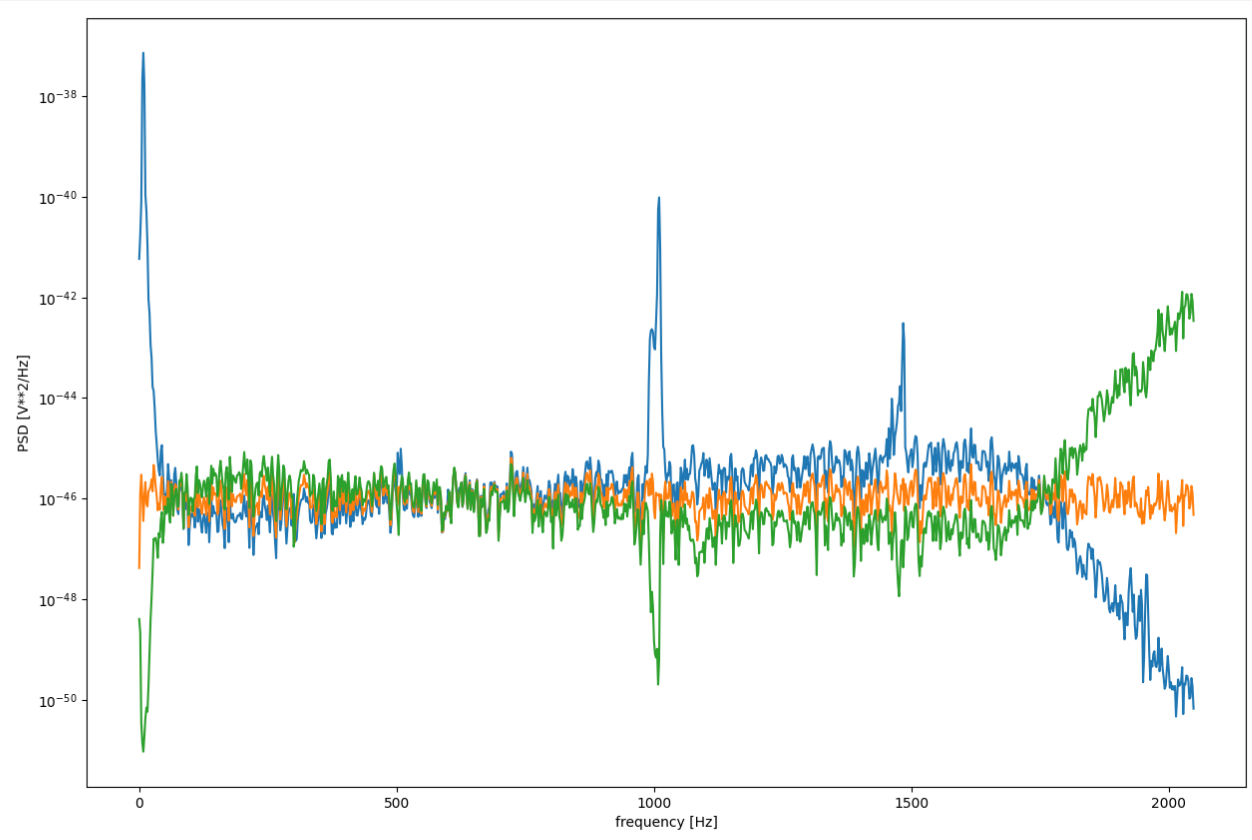
[5]:
```
import numpy as np

import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline

plt.rcParams['figure.figsize'] = (15.0, 10.0)
mpl_logger = logging.getLogger("matplotlib")
mpl_logger.setLevel(logging.WARNING)

x=np.zeros(data.GetSize())
y=np.zeros(data.GetSize())
yw=np.zeros(data.GetSize())
yww=np.zeros(data.GetSize())

for i in range(data.GetSize()):
    x[i]=data.GetX(i)
    y[i]=data.GetY(0,i)
    yw[i]=dataw.GetY(0,i)
    yww[i]=dataww.GetY(0,i)

plt.figure(figsize=(10,4))
plt.plot(x, y,  label='Raw data')
plt.plot(x, yw, label='Whitened data')
plt.plot(x, yww, label='Double whitened data')

plt.legend()
plt.show()
```



Frequency domain (PSD)

[6]:

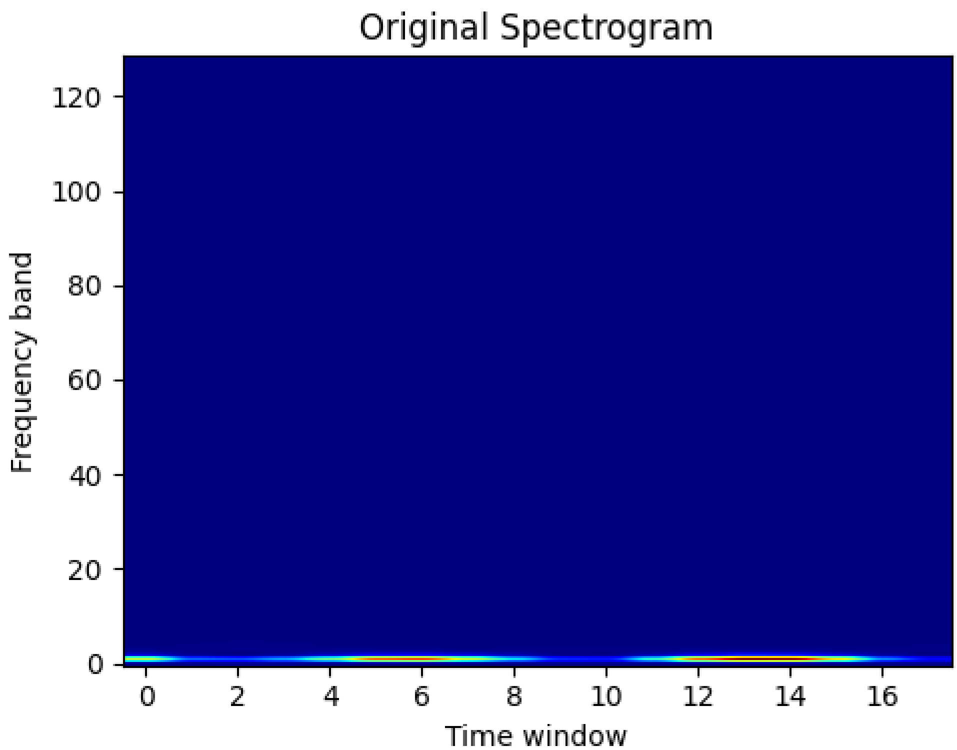
```
from scipy import signal
f, Pxx_den = signal.welch(y, par.sampling, nperseg=2048)
f, Pxx_denW = signal.welch(yw, par.sampling, nperseg=2048)
f, Pxx_denWW = signal.welch(yww, par.sampling, nperseg=2048)
fig, ax = plt.subplots()
ax.semilogy(f, Pxx_den)
ax.semilogy(f, Pxx_denW)
ax.semilogy(f, Pxx_denWW)

plt.xlabel('frequency [Hz]')
plt.ylabel('PSD [V**2/Hz]')
plt.show()
```



Time-Frequency domain

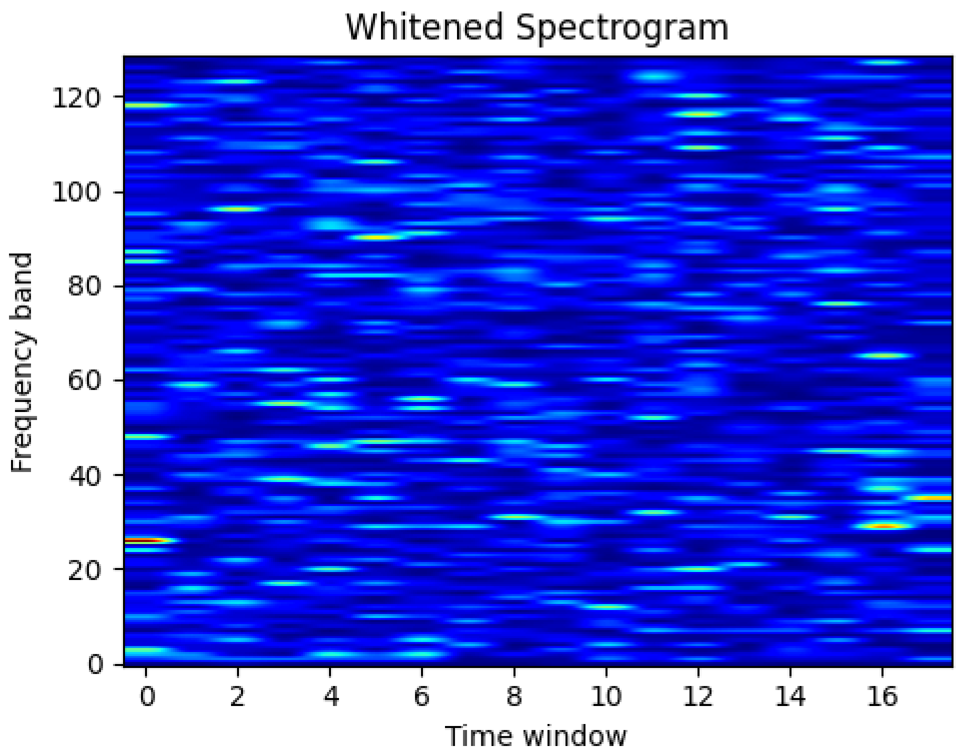
[7]:
```
plt.figure(figsize=(10,4)),
freqs, times, spectrogram = signal.spectrogram(y)

plt.figure(figsize=(5, 4))
plt.imshow(spectrogram, aspect='auto', cmap='jet', origin='lower')
plt.title('Original Spectrogram')
plt.ylabel('Frequency band')
plt.xlabel('Time window')
plt.tight_layout()
```

```
<Figure size 1000x400 with 0 Axes>
```



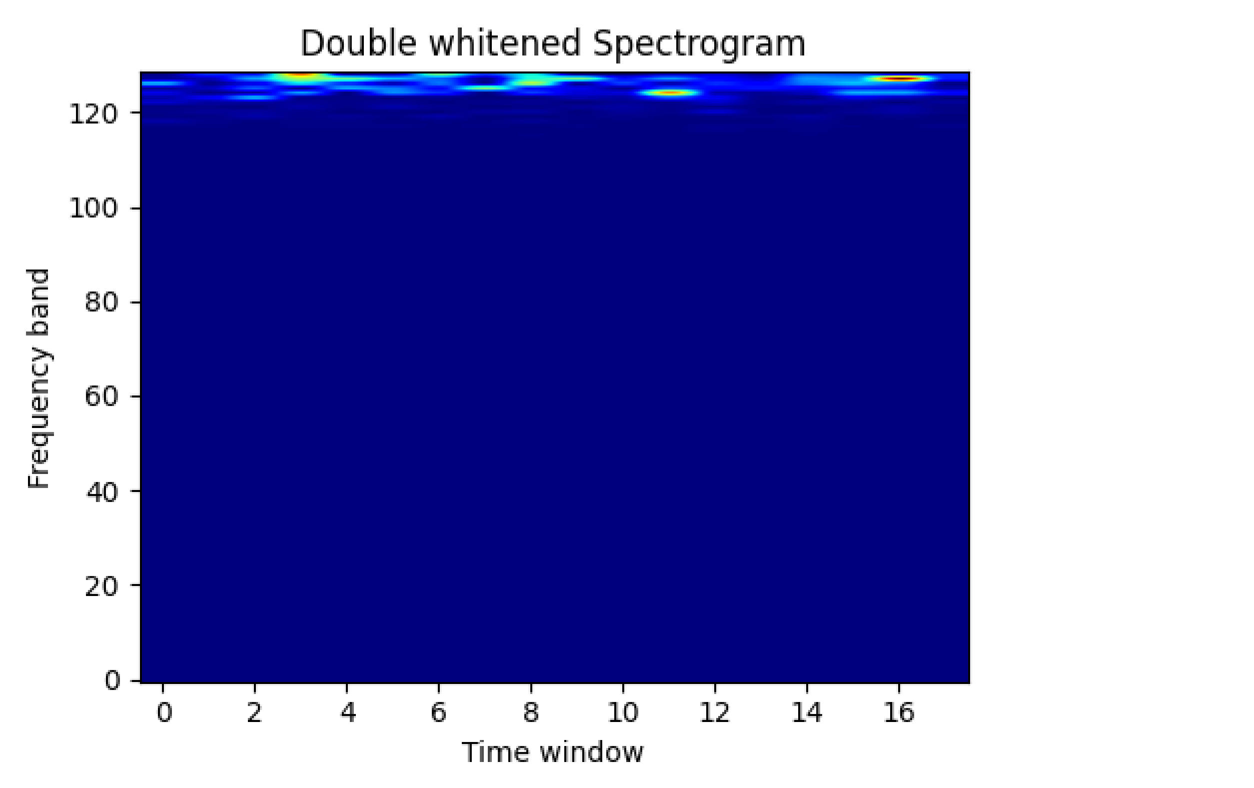
[8]:
```
plt.figure(figsize=(10,4)),
freqs, times, spectrogram = signal.spectrogram(yw)

plt.figure(figsize=(5, 4))
plt.imshow(spectrogram, aspect='auto', cmap='jet', origin='lower')
plt.title('Whitened Spectrogram')
plt.ylabel('Frequency band')
plt.xlabel('Time window')
plt.tight_layout()
```

```
<Figure size 1000x400 with 0 Axes>
```



[9]:
```
plt.figure(figsize=(10,4)),
freqs, times, spectrogram = signal.spectrogram(yww)

plt.figure(figsize=(5, 4))
plt.imshow(spectrogram, aspect='auto', cmap='jet', origin='lower')
plt.title('Double whitened Spectrogram')
plt.ylabel('Frequency band')
plt.xlabel('Time window')
plt.tight_layout()
```

```
<Figure size 1000x400 with 0 Axes>
```



1.2.2 InverseWhitening

Inverse Whitening procedure with AutoRegressive (AR) model

author: Elena Cuoco

- We can 'color' the data which have been whitened, using the P AR parameters and an ARMA(P,1) filter

[1]:
```
import time
import os
import pytsa
from pytsa.tsa import *
from pytsa.tsa import SeqView_double_t as SV
from wdf.config.Parameters import *
from wdf.processes.Whitening import  *
from wdf.processes.DWhitening import  *
import logging, sys

logger = logging.getLogger()
logger.setLevel(logging.INFO)
logging.debug("info")

new_json_config_file = True    # set to True if you want to create new Configuration
if new_json_config_file==True:
    configuration = {
      "file": "./data/test.gwf",
      "channel": "H1:GWOSC-4KHZ_R1_STRAIN",
      "len":1.0,
```

(continues on next page)

(continued from previous page)

```
      "gps":1167559100,
      "outdir": "./",
      "dir":"./",
      "ARorder": 2000,
      "learn": 200,
      "preWhite":4
    }

    filejson = os.path.join(os.getcwd(),"InputParameters.json")
    file_json = open(filejson, "w+")
    json.dump(configuration, file_json)
    file_json.close()
logging.info("read parameters from JSON file")

par = Parameters()
filejson = "InputParameters.json"
try:
    par.load(filejson)
except IOError:
    logging.error("Cannot find resource file " + filejson)
    quit()

strInfo = FrameIChannel(par.file, par.channel, 1.0, par.gps)
Info = SV()
strInfo.GetData(Info)
par.sampling = int(1.0 / Info.GetSampling())
logging.info("channel= %s at sampling frequency= %s" %(par.channel, par.sampling))

whiten=Whitening(par.ARorder)
par.ARfile = "./ARcoeff-AR%s-fs%s-%s.txt" % (
                par.ARorder, par.sampling, par.channel)
par.LVfile ="./LVcoeff-AR%s-fs%s-%s.txt" % (
                par.ARorder, par.sampling, par.channel)

if os.path.isfile(par.ARfile) and os.path.isfile(par.LVfile):
    logging.info('Load AR parameters')
    whiten.ParametersLoad(par.ARfile, par.LVfile)
else:
    logging.info('Start AR parameter estimation')
    ######## read data for AR estimation###############
    strLearn = FrameIChannel(par.file, par.channel, par.learn, par.gps)
    Learn = SV()
    strLearn.GetData(Learn)
    whiten.ParametersEstimate(Learn)
    whiten.ParametersSave(par.ARfile, par.LVfile)
```

```
INFO:root:read parameters from JSON file
INFO:root:channel= H1:GWOSC-4KHZ_R1_STRAIN at sampling frequency= 4096
INFO:root:Start AR parameter estimation
```

[2]:

```
data = SV()
dataw = SV()
streaming = FrameIChannel(par.file, par.channel, par.len, par.gps)

streaming.GetData(data)
N=data.GetSize()
```

(continues on next page)

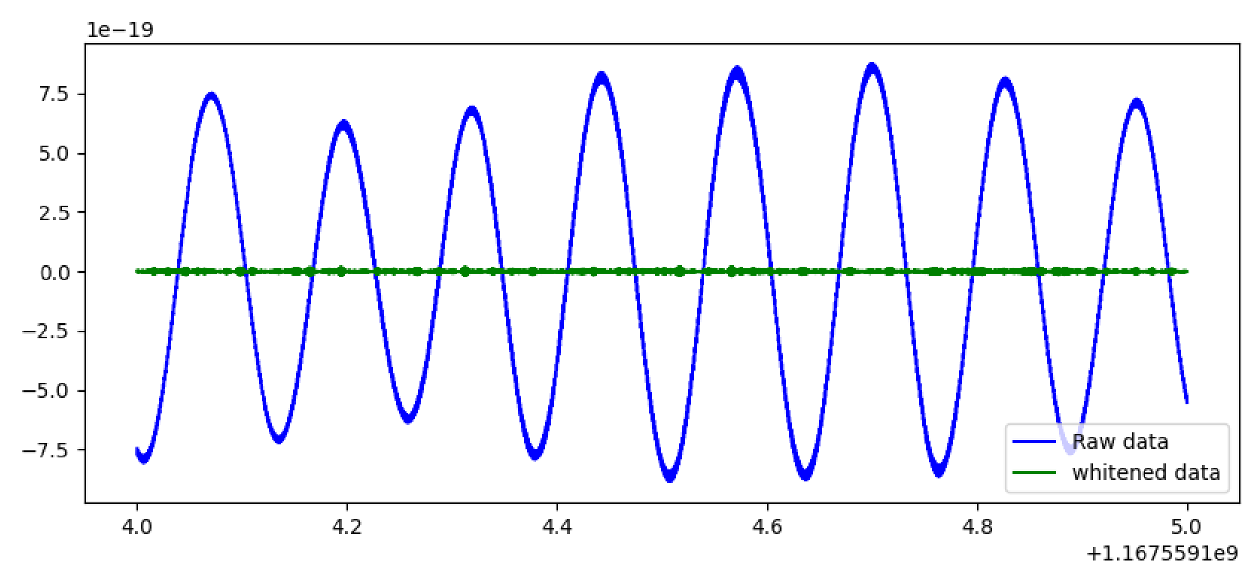
(continued from previous page)

```
Dwhiten=DWhitening(whiten.LV,N,0)

for i in range(par.preWhite):
    streaming.GetData(data)
    Dwhiten.Process(data, dataw)
```

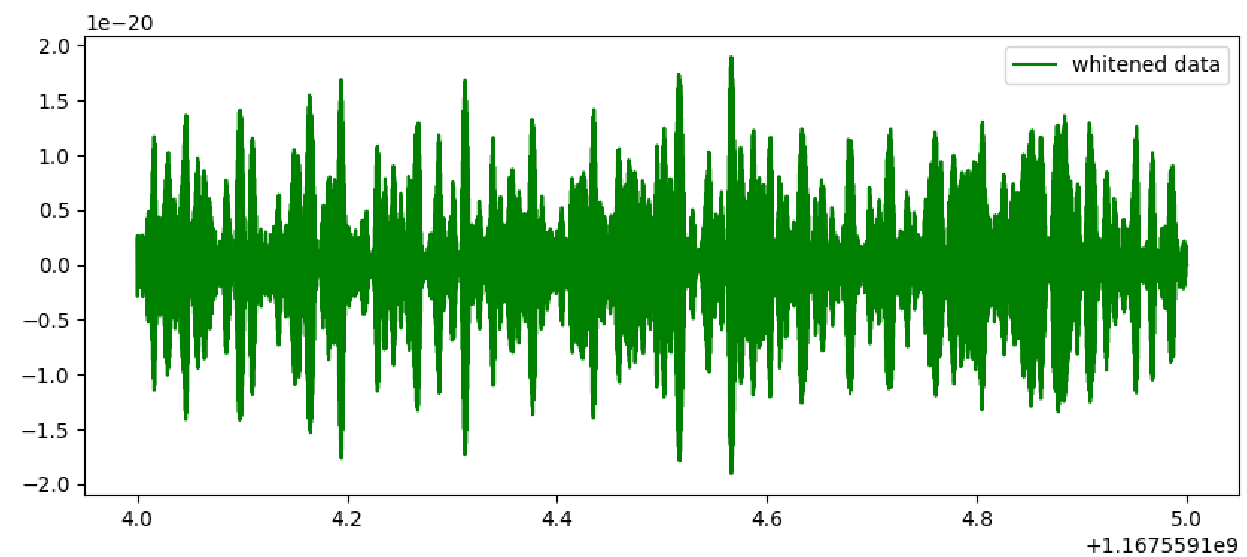
(D)Whiten the data

How whiten your data are depends on a series of factors: the stationarity of the noise, the number of AR parameters you used, the lenght of the sequence of data you used to estimate the parameters

[3]:
```
import numpy as np
import matplotlib

import matplotlib.pyplot as plt

%matplotlib inline

plt.rcParams['figure.figsize'] = (15.0, 10.0)
mpl_logger = logging.getLogger("matplotlib")
mpl_logger.setLevel(logging.WARNING)


x=np.zeros(data.GetSize())
y=np.zeros(data.GetSize())
yw=np.zeros(dataw.GetSize())

for i in range(data.GetSize()):
    x[i]=data.GetX(i)
    y[i]=data.GetY(0,i)
    yw[i]=dataw.GetY(0,i)
plt.figure(figsize=(10,4))


plt.plot(x, y, 'b',label='Raw data')
plt.plot(x, yw, 'g',label='whitened data')
plt.legend()
plt.show()
```



[4]:
```
plt.figure(figsize=(10,4))


plt.plot(x, yw, 'g', label='whitened data')

plt.legend()
plt.show()
```



Recoloring data using an ARMA (P,Q) filter

P= the numer of AR parameters, Q=1

In order to take into account the transient response of the filter, we need to do a 'preaheating for the filter' and so go first in a loop to get good result

[5]:
```
from wdf.processes.Coloring import *
from wdf.structures.array2SeqView import *


datac = SV()

dataw=SV()

Colored=Coloring(par.ARorder)
Colored.ParametersLoad(par.ARfile)
for j in range(5):
    streaming.GetData(data)
    whiten.Process(data, dataw)
    Colored.Process(dataw, datac)
```

[6]:
```
%matplotlib notebook
```

[7]:
```
x=np.zeros(data.GetSize())
y=np.zeros(data.GetSize())
yc=np.zeros(datac.GetSize())


for i in range(data.GetSize()):
    x[i]=data.GetX(i)
    y[i]=data.GetY(0,i)
    yc[i]=datac.GetY(0,i)

plt.figure(figsize=(10,4))

plt.plot(x, y,  label='Raw data')
plt.plot(x, yc, label='Recolored-data')

plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

TO BE DOCUMENTED

1.2.3 Multi-WDF

Example of WDf usage with multi data segments and multi-processing

author: Elena Cuoco

Cuoco et al., Wavelet-Based Classification of Transient Signals for Gravitational Wave Detectors DO - 10.23919/EUSIPCO.2018.8553393

Please note that many packages, as graphic one or logging are not part of WDF docker, but you can install them locally or use your preferred ones

[1]:
```
# import libraries
import time
import os
from pytsa.tsa import *
from pytsa.tsa import SeqView_double_t as SV
from wdf.config.Parameters import *
from wdf.processes.wdfUnitDSWorker import *
from wdf.processes.wdfUnitWorker import *
import logging
import coloredlogs
#select level of logging
coloredlogs.install(isatty=True)

logging.basicConfig(level=logging.DEBUG)
```

[2]:
```
new_json_config_file = True    # set to True if you want to create new Configuration

if new_json_config_file==True:
    configuration = {
  "window":1024,
  "overlap":768,
  "threshold": 0.2,
  "file": "./data/test.gwf",
  "channel": "H1:GWOSC-4KHZ_R1_STRAIN",
  "run":"offLine",
  "len":10.0,
  "gps":1167559608,
  "segments":[[1167559008,1167559408],[1167559408,1167560008],[1167560008,
↪1167560308]],
  #"segments":[[1167559608,1167560008] ],
  "outdir": "local_dir/",
  "dir":"local_dir/",
  "ID":"WDF_test",
  "ARorder": 1000,
  "learn": 200,
  "preWhite":2,
  "ResamplingFactor":2,
  "LowFrequencyCut":12,
  "FilterOrder":6,
  "nproc":4
}

    filejson = os.path.join(os.getcwd(),"inputWDF.json")
    file_json = open(filejson, "w+")
    json.dump(configuration, file_json)
    file_json.close()

logging.info("read parameters from JSON file")
par = Parameters()

filejson = "inputWDF.json"
try:
    par.load(filejson)
except IOError:
    logging.error("Cannot find resource file " + filejson)
```
(continues on next page)

(continued from previous page)

```
    quit()

par.print()
```

```
2023-03-29 14:18:51 hal2022 root[4076457] INFO read parameters from JSON file
```

```
{   'ARorder': 1000,
    'FilterOrder': 6,
    'ID': 'WDF_test',
    'LowFrequencyCut': 12,
    'ResamplingFactor': 2,
    'channel': 'H1:GWOSC-4KHZ_R1_STRAIN',
    'dir': 'local_dir/',
    'file': './data/test.gwf',
    'gps': 1167559608,
    'learn': 200,
    'len': 10.0,
    'nproc': 4,
    'outdir': 'local_dir/',
    'overlap': 768,
    'preWhite': 2,
    'run': 'offLine',
    'segments': [   [1167559008, 1167559408],
                    [1167559408, 1167560008],
                    [1167560008, 1167560308]],
    'threshold': 0.2,
    'window': 1024}
```

It is important that you define correctly the parameters in the configuration files. WDf is a pipeline which performs a series of steps before producing triggers for transient signals in your data. Here it is the list of parameters you can fix in your configuration file.

- ARorder = order of AutoRegressive model for whitening
- ID = identification numer for your run
- ResamplingFactor = the ratio between the original sampling frequency and the downsampled one
- channel = the name of the channel in you .gwf o .ffl file you want to analyze
- dir = where to find the parameters
- file = file to be anlyzed
- gps = the starting time for for analysis (overwritten by values in segments)
- learn = the length in seconds of the data you will use to estimate AR parameters
- len = the time window in second of data loaded in you loop
- nproc = the number of processors you will use
- outdir = where you want to save the results
- overlap = overlapping number between 2 consecutives windows for WDF analysis
- prewhite = the number of iter to pre-heat the whitening procedure (leave as it is)
- run = additional tag for your data run
- segments = 1 or more segments defined as [start time, end time] where you will run WDF, usually 1 segment/processor
- threshold = the minimum value for WDF snr to identify a trigger

- window = the analyzing window in point for WDF. It should be a power of 2

If you set the par.dir or par.outdir as relative path, we need to give the absolute path.

[3]:
```
import os
par.dir=os.getcwd()+'/'+par.dir
par.outdir=os.getcwd()+'/'+par.outdir
```

Load information for sampling frequency

[4]:
```
strInfo = FrameIChannel(par.file, par.channel, 1.0, par.gps)
Info = SV()
strInfo.GetData(Info)
par.sampling = int(1.0 / Info.GetSampling())
if par.ResamplingFactor!=None:
    par.resampling = int(par.sampling / par.ResamplingFactor)
    logging.info("sampling frequency= %s, resampled frequency= %s" %(par.sampling,␣
↪par.resampling))
del Info, strInfo
```

```
2023-03-29 14:18:52 hal2022 root[4076457] INFO sampling frequency= 4096, resampled␣
↪frequency= 2048
```

Launch WDF runs

The fullPrint option is important to save information about the WDF triggers

- fullPrint = 0 –> you save only the metaparameters for the triggers
- fullPrint = 1 –> you save the metaparameters and the wavelet coefficients for that trigger
- fullPrint = 2 –> you save the metaparameters and the reconstructed waveform for that trigger
- fullPring = 3 –> you save the metaparameters, the wavelet coefficients and the reconstructed waveform for that trigger in the 'window' time (window/sampling frequency)

[5]:
```
import multiprocessing as mp
print("Number of processors: ", mp.cpu_count())
pool = mp.Pool(par.nproc)

wdf=wdfUnitDSWorker(par,fullPrint=2)
pool.map(wdf.segmentProcess, [segment for segment in par.segments])
pool.close()
```

```
Number of processors:  32
```

```
2023-03-29 14:18:52 hal2022 root[4076576] INFO Analyzing segment: 1167560008-
↪1167560308 for channel H1:GWOSC-4KHZ_R1_STRAIN downsampled at 2048Hz
2023-03-29 14:18:52 hal2022 root[4076575] INFO Analyzing segment: 1167559408-
↪1167560008 for channel H1:GWOSC-4KHZ_R1_STRAIN downsampled at 2048Hz
2023-03-29 14:18:52 hal2022 root[4076574] INFO Analyzing segment: 1167559008-
↪1167559408 for channel H1:GWOSC-4KHZ_R1_STRAIN downsampled at 2048Hz
2023-03-29 14:18:52 hal2022 root[4076575] INFO Start AR parameter estimation
2023-03-29 14:18:52 hal2022 root[4076574] INFO Start AR parameter estimation
2023-03-29 14:18:52 hal2022 root[4076576] INFO Start AR parameter estimation
```

(continues on next page)

(continued from previous page)

```
2023-03-29 14:19:27 hal2022 root[4076575] INFO Estimated sigma= 4.0675451444151897e-22
2023-03-29 14:19:27 hal2022 root[4076576] INFO Estimated sigma= 4.0062637269449494e-22
2023-03-29 14:19:27 hal2022 root[4076574] INFO Estimated sigma= 4.0023147527036696e-22
2023-03-29 14:19:32 hal2022 root[4076575] INFO Starting detection loop
2023-03-29 14:19:32 hal2022 root[4076576] INFO Starting detection loop
2023-03-29 14:19:32 hal2022 root[4076574] INFO Starting detection loop
2023-03-29 14:22:25 hal2022 root[4076576] INFO analyzed 300 seconds in 212.
↪9326889514923 seconds
2023-03-29 14:23:22 hal2022 root[4076574] INFO analyzed 400 seconds in 270.
↪102082490921 seconds
2023-03-29 14:25:12 hal2022 root[4076575] INFO analyzed 600 seconds in 379.
↪5118489265442 seconds
```

In the output dir with 'run' tag you will find the estimated AR coefficients, and a .csv files containing the trigger lists

Let's have a look at the results

[6]:
```
import pandas as pd

import glob

dirName = par.outdir # use your path
all_files = glob.glob(os.path.join(dirName, "*","*.csv"))     # advisable to use os.
↪path.join as this makes concatenation OS independent
df_from_each_file = (pd.read_csv(f) for f in all_files)
triggers  = pd.concat(df_from_each_file, ignore_index=True)
```

[7]:
```
triggers.shape
```

[7]:
```
(5810, 1035)
```

[8]:
```
import matplotlib.pylab as plt
pd.set_option('display.max_rows', 999)
pd.set_option('max_colwidth',100)
pd.set_option('display.float_format', lambda x: '%.3f' % x)
import numpy as np

%matplotlib inline
# Alternatives include bmh, fivethirtyeight, ggplot,
# dark_background, seaborn-deep, etc
plt.style.use('ggplot')
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 10
plt.rcParams['axes.labelsize'] = 10
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['legend.fontsize'] = 10
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['figure.figsize'] = (14, 10)

colors = np.array([x for x in 'bgrcmykbgrcmykbgrcmykbgrcmyk'])
colors = np.hstack([colors] * 20)
plt.figure(0)
```

(continues on next page)

(continued from previous page)

```
print (triggers.shape)

triggers.head(10)
```

```
(5810, 1035)
```

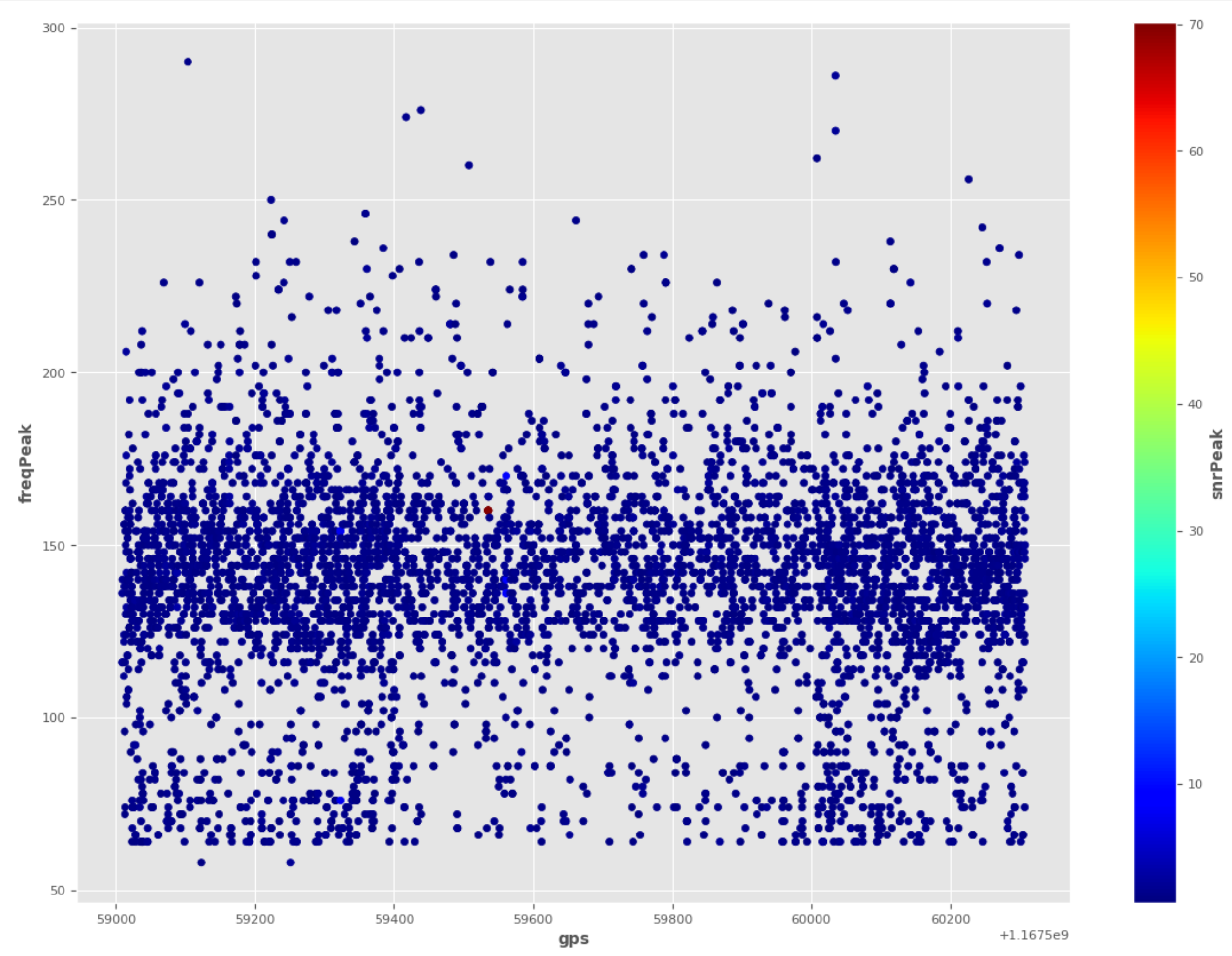
[8]:

```
               gps          gpsPeak  duration  EnWDF  snrMean  snrPeak  freqMin  \
0 1167559009.000 1167559009.421     0.453  0.290    0.256    1.786   62.000
1 1167559009.125 1167559009.421     0.500  0.331    0.275    1.809   62.000
2 1167559009.250 1167559009.421     0.473  0.275    0.248    1.734   58.000
3 1167559009.375 1167559009.421     0.477  0.235    0.220    1.628   48.000
4 1167559011.375 1167559011.782     0.500  0.305    0.219    1.788   88.000
5 1167559011.500 1167559011.782     0.404  0.311    0.224    1.782   86.000
6 1167559011.625 1167559011.782     0.500  0.350    0.255    1.769   78.000
7 1167559011.750 1167559011.782     0.473  0.369    0.278    1.794   70.000
8 1167559011.875 1167559012.096     0.461  0.252    0.221    1.293   56.000
9 1167559012.000 1167559012.096     0.458  0.246    0.219    1.269   54.000

   freqMean  freqMax  freqPeak  ... rw1014  rw1015  rw1016  rw1017  rw1018  \
0   148.346  250.000   136.000  ...  0.000  -0.000  -0.000  -0.000  -0.000
1   155.500  274.000   116.000  ... -0.000  -0.000  -0.000   0.000   0.000
2   153.423  266.000   116.000  ... -0.000   0.000   0.000   0.000   0.000
3   154.846  272.000   140.000  ...  0.000   0.000   0.000   0.000   0.000
4   173.769  282.000   112.000  ...  0.000   0.000   0.000   0.000   0.000
5   169.269  280.000   124.000  ... -0.000  -0.000   0.000   0.000   0.000
6   170.000  302.000   122.000  ... -0.000  -0.000  -0.000   0.000   0.000
7   168.385  298.000   138.000  ...  0.000   0.000   0.000   0.000   0.000
8   164.808  318.000   156.000  ...  0.000   0.000   0.000   0.000   0.000
9   174.000  356.000   156.000  ... -0.000  -0.000  -0.000  -0.000  -0.000

   rw1019  rw1020  rw1021  rw1022  rw1023
0  -0.000  -0.000  -0.000  -0.000   0.000
1   0.000   0.000   0.000   0.000   0.000
2   0.000   0.000  -0.000  -0.000   0.000
3  -0.000  -0.000  -0.000  -0.000  -0.000
4  -0.000  -0.000  -0.000  -0.000  -0.000
5   0.000   0.000   0.000   0.000   0.000
6   0.000   0.000   0.000   0.000   0.000
7   0.000   0.000   0.000  -0.000  -0.000
8  -0.000  -0.000  -0.000  -0.000  -0.000
9  -0.000  -0.000   0.000   0.000   0.000

[10 rows x 1035 columns]
```

```
<Figure size 1400x1000 with 0 Axes>
```

[9]:

```
ax2 = triggers.plot.scatter(x='gps',
                      y='freqPeak',
                      c='snrPeak',
                      colormap='jet')
```



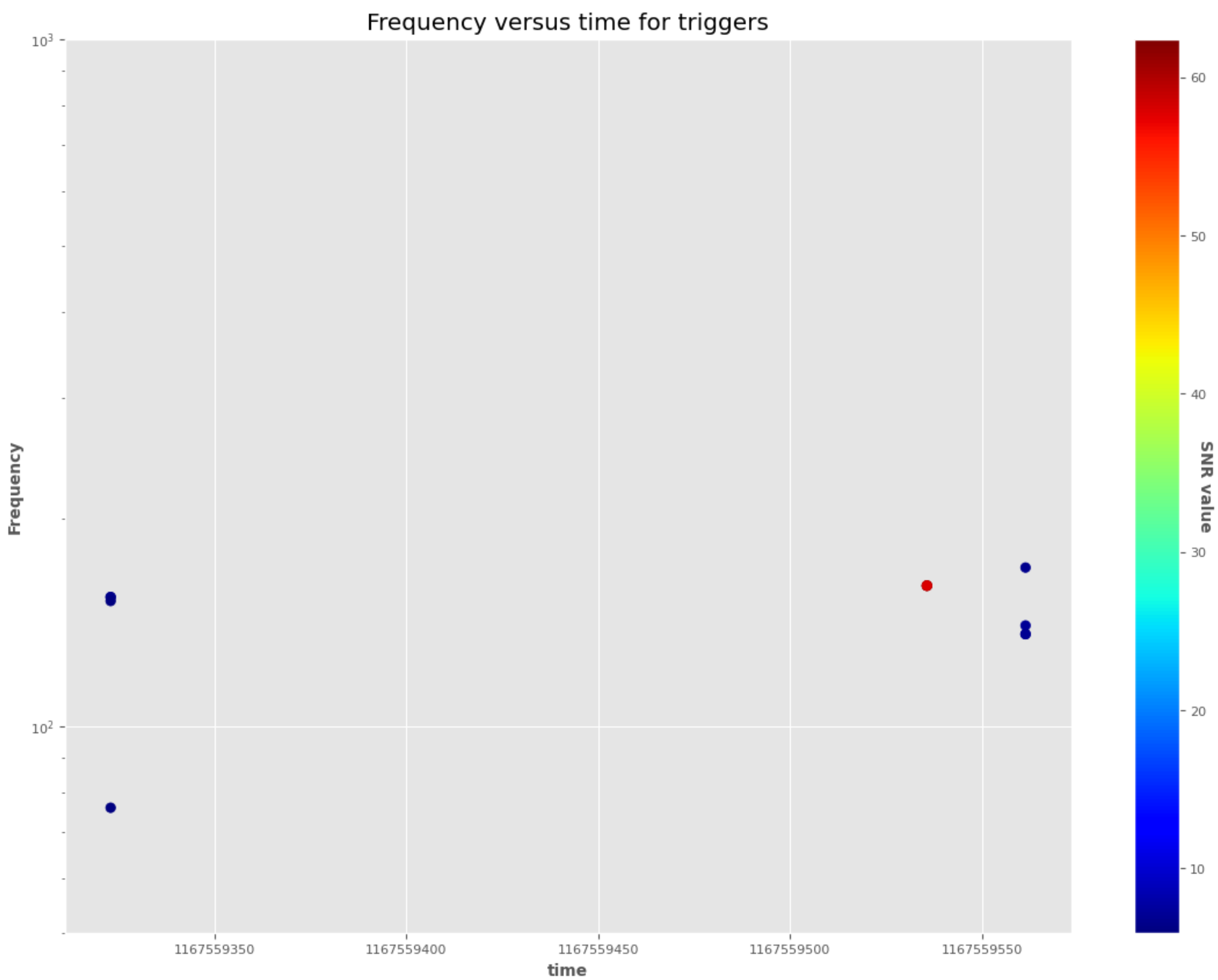
[10]:
```
df=triggers[triggers['snrPeak']>4]
```

[11]:
```
plt.figure(0)
print (df.shape)

from matplotlib.ticker import FormatStrFormatter

sc = plt.scatter(df.gpsPeak,
                 df.freqPeak,c=df.EnWDF, cmap='jet')

# legend
cbar = plt.colorbar(sc)
cbar.ax.set_yticklabels(['5','10', '20', '30', '40', '50', '60','>60'])
cbar.set_label('SNR value', rotation=270)

plt.ylim(50, 1000)
plt.yscale('log')
plt.gca().get_xaxis().get_major_formatter().set_useOffset(False)
plt.gca().get_xaxis().get_major_formatter().set_scientific(False)
plt.xlabel("time")
plt.ylabel("Frequency")
plt.title("Frequency versus time for triggers")
```

```
(12, 1035)
```

```
/tmp/ipykernel_4076457/2239143324.py:11: UserWarning: FixedFormatter should only be␣
↪used together with FixedLocator
  cbar.ax.set_yticklabels(['5','10', '20', '30', '40', '50', '60','>60'])
```

[11]:
```
Text(0.5, 1.0, 'Frequency versus time for triggers')
```



[12]:
```
df=triggers.sort_values('EnWDF', ascending=False)
```

[13]:
```
df.head(10)
```

[13]:
```
                 gps         gpsPeak  duration  EnWDF  snrMean  snrPeak  \
4255 1167559535.375 1167559535.639     0.019  6.242    4.661   70.169
4254 1167559535.250 1167559535.639     0.019  6.219    4.659   70.154
4256 1167559535.500 1167559535.639     0.019  6.170    4.658   70.118
4257 1167559535.625 1167559535.639     0.019  5.798    4.624   69.640
4341 1167559560.875 1167559561.331     0.276  0.706    0.529    8.874
4343 1167559561.125 1167559561.331     0.107  0.687    0.517    8.830
1673 1167559322.500 1167559322.928     0.497  0.685    0.474    7.734
4344 1167559561.250 1167559561.331     0.018  0.679    0.506    8.830
4342 1167559561.000 1167559561.331     0.276  0.675    0.522    8.674
1674 1167559322.625 1167559322.928     0.411  0.648    0.462    7.642
```

(continues on next page)

(continued from previous page)

```
      freqMin  freqMean  freqMax  freqPeak  ... rw1014  rw1015  rw1016  \
4255   76.000   152.577  262.000   160.000  ... -0.000  -0.000  -0.000
4254   74.000   146.962  260.000   160.000  ...  0.000   0.000   0.000
4256   76.000   150.308  262.000   160.000  ... -0.000  -0.000  -0.000
4257   80.000   152.192  264.000   160.000  ...  0.000   0.000   0.000
4341   58.000   156.846  276.000   140.000  ...  0.000   0.000   0.000
4343   58.000   152.923  278.000   136.000  ...  0.000   0.000   0.000
1673   54.000   146.731  240.000   154.000  ...  0.000   0.000   0.000
4344   68.000   149.731  222.000   170.000  ...  0.000   0.000   0.000
4342   58.000   155.308  276.000   136.000  ...  0.000   0.000   0.000
1674   62.000   150.154  240.000   154.000  ...  0.000   0.000   0.000

      rw1017  rw1018  rw1019  rw1020  rw1021  rw1022  rw1023
4255  -0.000  -0.000  -0.000  -0.000  -0.000  -0.000  -0.000
4254   0.000   0.000   0.000   0.000   0.000   0.000   0.000
4256  -0.000  -0.000   0.000   0.000   0.000   0.000   0.000
4257   0.000   0.000   0.000   0.000  -0.000   0.000   0.000
4341   0.000   0.000   0.000   0.000  -0.000  -0.000  -0.000
4343   0.000   0.000   0.000   0.000   0.000   0.000  -0.000
1673   0.000   0.000  -0.000  -0.000  -0.000   0.000   0.000
4344   0.000   0.000   0.000   0.000  -0.000  -0.000  -0.000
4342   0.000   0.000   0.000  -0.000  -0.000  -0.000  -0.000
1674   0.000   0.000   0.000  -0.000  -0.000  -0.000  -0.000

[10 rows x 1035 columns]
```

[14]:

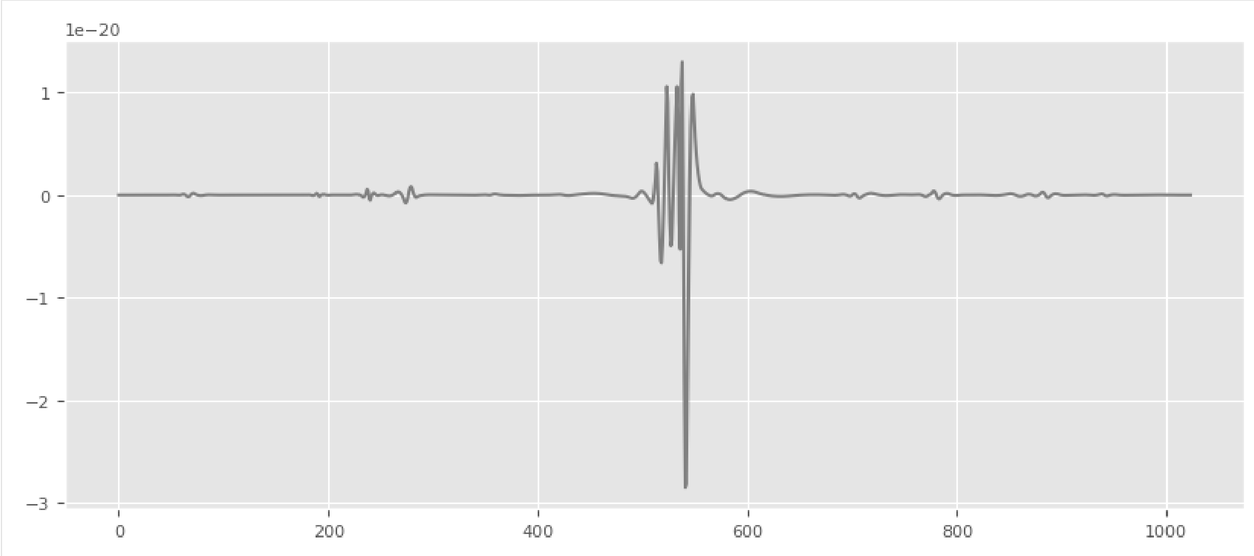
```
wav=np.array(triggers.loc[triggers['EnWDF'].idxmax()][11:].values)
```

[15]:

```
import matplotlib
import matplotlib.pyplot as plt
matplotlib.rcParams['agg.path.chunksize']=10000
%matplotlib inline

plt.figure(figsize=(10,4)),
plt.plot(wav,'gray',label='h'),
```

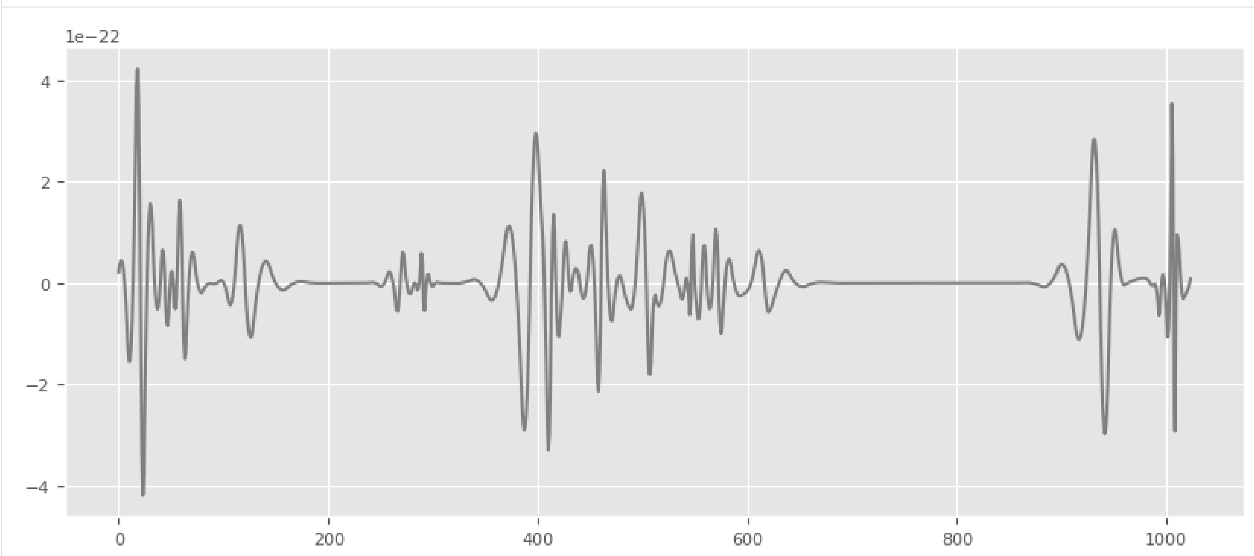
[15]:

```
([<matplotlib.lines.Line2D at 0x7fc742d42310>],)
```



[16]:
```
wav=np.array(triggers.loc[11][11:].values)

plt.figure(figsize=(10,4)),
plt.plot(wav,'gray',label='h'),
```

[16]:
```
([&lt;matplotlib.lines.Line2D at 0x7fc742d950d0>],)
```



[ ]:

1.2.4 SignalFiltering

Example of data filtering, using utility functions in wdf

author: Elena Cuoco

Sometime you may need to high pass, low pass or filter your data, while you are in a loop with SeqView of data. We will show how you can do this with SeqView structures, using utility functions which rely on scipy signal library

[1]:
```
import time
import os
from pytsa.tsa import *
from pytsa.tsa import SeqView_double_t as SV
from wdf.config.Parameters import *
import numpy as np

import logging, sys
logging.disable(sys.maxsize)
logging.basicConfig(level=logging.DEBUG)
new_json_config_file = True    # set to True if you want to create new Configuration

new_json_config_file = True    # set to True if you want to create new Configuration
if new_json_config_file==True:
    configuration = {
      "file": "./data/test.gwf",
      "channel": "H1:GWOSC-4KHZ_R1_STRAIN",
      "len":1.0,
      "gps":1167559100,
      "outdir": "./",
      "dir":"./",
      "ARorder": 2000,
      "learn": 200,
      "preWhite":4
    }

    filejson = os.path.join(os.getcwd(),"InputParameters.json")
    file_json = open(filejson, "w+")
    json.dump(configuration, file_json)
    file_json.close()
logging.info("read parameters from JSON file")

par = Parameters()
filejson = "InputParameters.json"
try:
    par.load(filejson)
except IOError:
    logging.error("Cannot find resource file " + filejson)
    quit()

strInfo = FrameIChannel(par.file, par.channel, 1.0, par.gps)
Info = SV()
strInfo.GetData(Info)
par.sampling = int(1.0 / Info.GetSampling())
logging.info("channel= %s at sampling frequency= %s" %(par.channel, par.sampling))
```

Butterworth Filter

Often it is useful to cut off low frequency in your data, because the noise is too high and you are not looking for signal in that region of frequency. We can do using a Butterworth filter from scipy library. It is advised that you make a design study offline to decide better the order of your filter.

[2]:
```
from wdf.utility.Filters import *
from wdf.utility.HighPassFilter import *
frequency=20
```

(continues on next page)

(continued from previous page)

```
sampling=par.sampling
order=5
filtertype='high'
bf=Butterworth(frequency, sampling, order, filtertype)
hp=HighPassFilter(frequency, sampling,order)
```

Pre-heating of the filter

We use some chunck of data to pre-heating the filtering procedure and avoiding the filter tail.

[3]:

```
data = SV()
dataf = SV()
datafl = SV()
streaming = FrameIChannel(par.file, par.channel, par.len, par.gps)

###---filter preheating---###
for i in range(1):
    streaming.GetData(data)
    dataf=bf.ProcessSeq(data)
    datafl=hp.Process(data)
```

Filtering

[4]:

```
#filtering
streaming.GetData(data)
dataf=bf.ProcessSeq(data)
datafl=hp.Process(data)
```

Plot: raw and filtered data

Time-domain

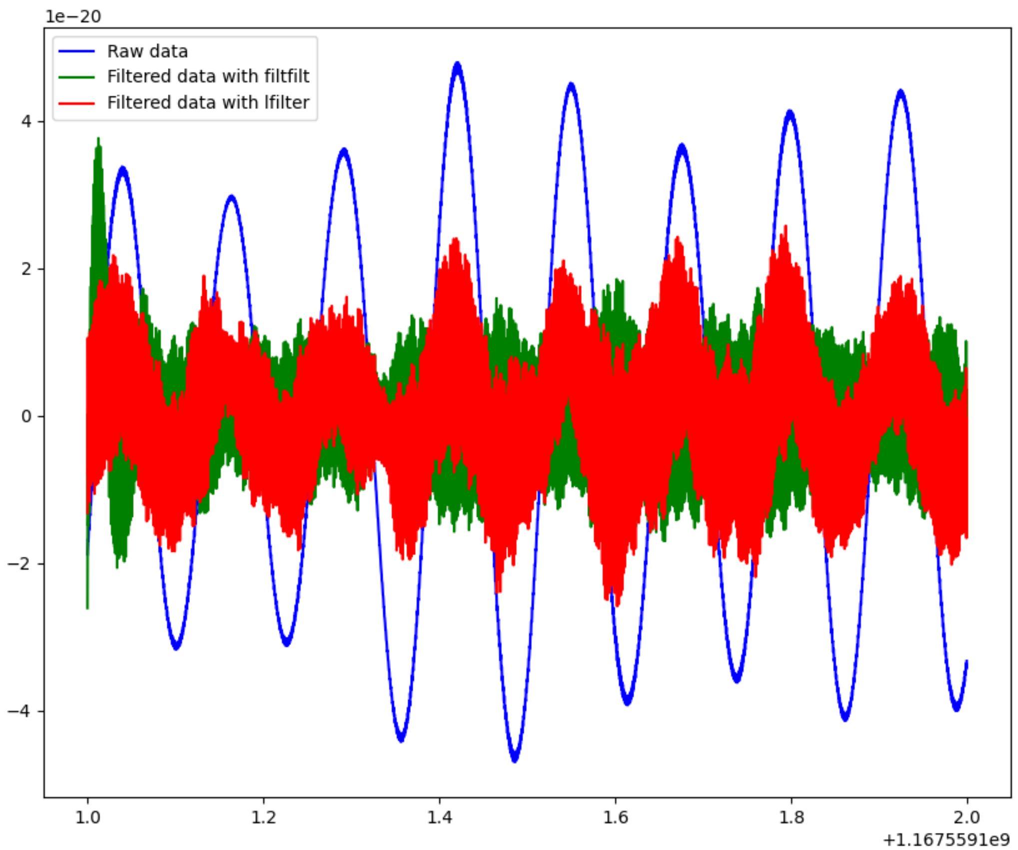
[5]:

```
import matplotlib.pylab as plt
import numpy as np
import logging
%matplotlib widget
mpl_logger = logging.getLogger("matplotlib")
mpl_logger.setLevel(logging.WARNING)
x=np.zeros(data.GetSize())
y=np.zeros(data.GetSize())
yf=np.zeros(dataf.GetSize())
yfl=np.zeros(dataf.GetSize())


for i in range(data.GetSize()):
    x[i]=data.GetX(i)
    y[i]=data.GetY(0,i)
    yf[i]=dataf.GetY(0,i)
    yfl[i]=datafl.GetY(0,i)
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(10,8))

plt.plot(x, y/20,'b', label='Raw data')
plt.plot(x, yf, 'g', label='Filtered data with filtfilt')
plt.plot(x, yfl, 'r',label='Filtered data with lfilter')

plt.legend()
plt.show()
```



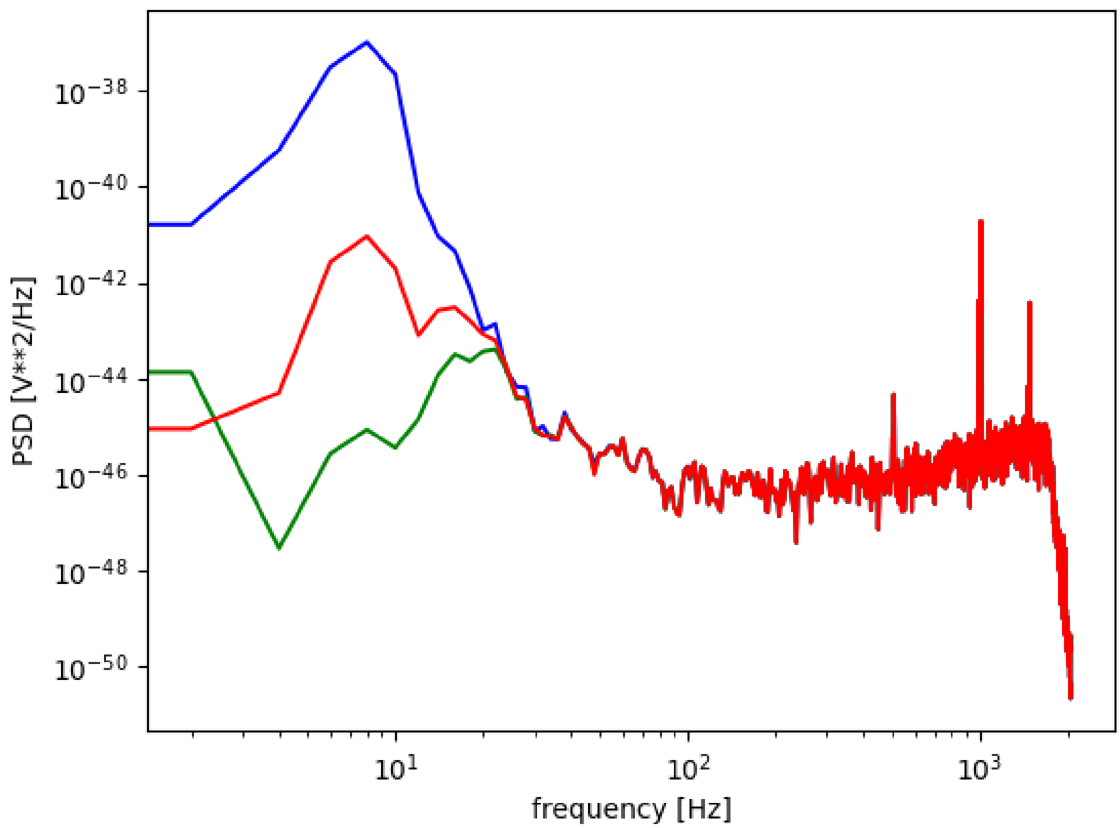
Frequency domain (PSD)

[6]:
```
from scipy import signal
f, Pxx_den = signal.welch(y, par.sampling, nperseg=2048)
f, Pxx_denW = signal.welch(yf, par.sampling, nperseg=2048)
f, Pxx_denHP= signal.welch(yfl, par.sampling, nperseg=2048)
fig, ax = plt.subplots()
ax.loglog(f, Pxx_den,'b')
ax.loglog(f, Pxx_denW,'g')
ax.loglog(f, Pxx_denHP,'r')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('frequency [Hz]')
plt.ylabel('PSD [V**2/Hz]')
plt.show()
```



1.2.5 WaveformRecon

Waveform denoised and reconstructed

author: Elena Cuoco

We want to show you how a gravitational wave signal becomes more apparent after whitening and double whitening of the data. The data are not downsampled

Double whitening refers to the procedure applied in the time domain of data whitening, using the inverse of PSD. However, the method used in pytsa is based on the parametric estimation (AR) of the PSD and the Lattice Filter implementation in the time domain.

[1]:
```
import time
import os
from pytsa.tsa import *
from pytsa.tsa import SeqView_double_t as SV
from wdf.config.Parameters import *
```

(continues on next page)

(continued from previous page)

```
from wdf.processes.Whitening import  *
from wdf.processes.DWhitening import  *

import logging, sys
logger = logging.getLogger()
logger.setLevel(logging.INFO)
logging.debug("info")

new_json_config_file = True    # set to True if you want to create new Configuration
if new_json_config_file==True:
    configuration = {
      "file": "./data/test.gwf",
      "channel": "H1:GWOSC-4KHZ_R1_STRAIN",
      "len":1.0,
      "gps":1167559536,
      "outdir": "./",
      "dir":"./",
      "ARorder": 3000,
      "learn": 300,
      "preWhite":4
    }

    filejson = os.path.join(os.getcwd(),"WavRec.json")
    file_json = open(filejson, "w+")
    json.dump(configuration, file_json)
    file_json.close()
logging.info("read parameters from JSON file")

par = Parameters()
filejson = "WavRec.json"
try:
    par.load(filejson)
except IOError:
    logging.error("Cannot find resource file " + filejson)
    quit()

strInfo = FrameIChannel(par.file, par.channel, 1.0, par.gps)
Info = SV()
strInfo.GetData(Info)
par.sampling = int(1.0 / Info.GetSampling())
logging.info("channel= %s at sampling frequency= %s" %(par.channel, par.sampling))

whiten=Whitening(par.ARorder)
par.ARfile = "./ARcoeff-AR%s-fs%s-%s.txt" % (
                par.ARorder, par.sampling, par.channel)
par.LVfile ="./LVcoeff-AR%s-fs%s-%s.txt" % (
                par.ARorder, par.sampling, par.channel)

if os.path.isfile(par.ARfile) and os.path.isfile(par.LVfile):
    logging.info('Load AR parameters')
    whiten.ParametersLoad(par.ARfile, par.LVfile)
else:
    logging.info('Start AR parameter estimation')
    ######## read data for AR estimation###############
    strLearn = FrameIChannel(par.file, par.channel, par.learn, par.gps)
    Learn = SV()
    strLearn.GetData(Learn)
```

(continues on next page)

(continued from previous page)

```
    whiten.ParametersEstimate(Learn)
    whiten.ParametersSave(par.ARfile, par.LVfile)
```

```
INFO:root:read parameters from JSON file
INFO:root:channel= H1:GWOSC-4KHZ_R1_STRAIN at sampling frequency= 4096
INFO:root:Start AR parameter estimation
```

[2]:
```
# sigma for the noise
par.sigma = whiten.GetSigma()
print('Estimated sigma= %s' % par.sigma)
```

```
Estimated sigma= 4.951028717156321e-22
```

We use some chunck of data to pre-heating the whitening procedure and avoiding the filter tail.

[3]:
```
#Try to center 1sec beore and 1 after the event
lenS=2.0
gpsEvent=1167559936.6
gps=gpsEvent-1.0-par.preWhite*lenS
data = SV()
dataw = SV()
dataww = SV()
N=int(par.sampling*lenS)
streaming = FrameIChannel(par.file, par.channel, lenS, gps)
Dwhiten=DWhitening(whiten.LV ,N,0)

###---whitening preheating---###
for i in range(par.preWhite):
    streaming.GetData(data)
    whiten.Process(data, dataw)
    Dwhiten.Process(data, dataww)
```

[4]:
```
print(par.file, par.channel, lenS, gps,dataw.GetStart())
```

```
./data/test.gwf H1:GWOSC-4KHZ_R1_STRAIN 2.0 1167559927.6 1167559933.6
```

[5]:
```
streaming.GetData(data)
whiten.Process(data, dataw)
Dwhiten.Process(data, dataww)
```

[6]:
```
print(dataw.GetStart(),dataw.GetY(0,0))
```

```
1167559935.6 5.647725652827097e-23
```

[7]:
```
print(dataw.GetSize()/par.sampling)
```

```
2.0
```

Plot: raw and whitened data

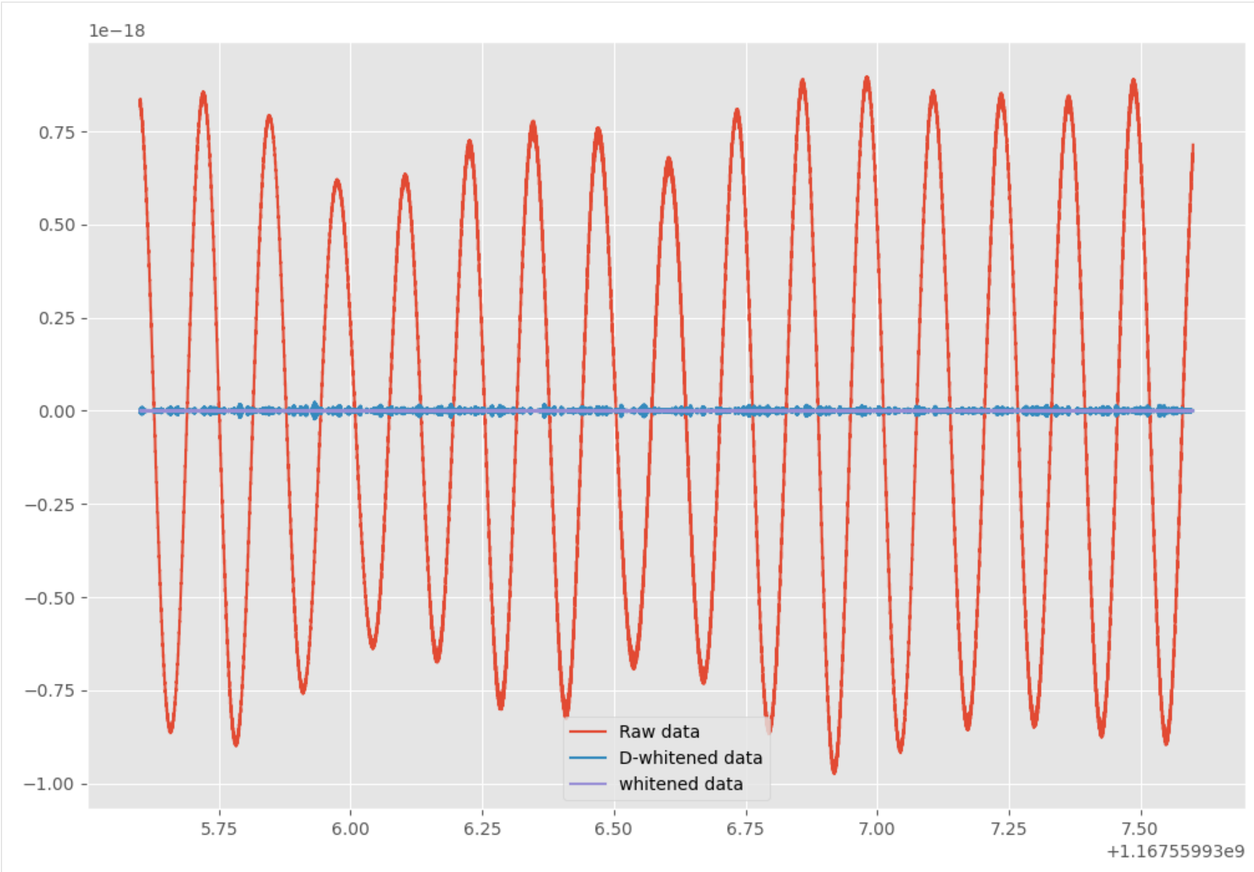
time-domain

[8]:
```
import numpy as np
import logging
```

(continues on next page)

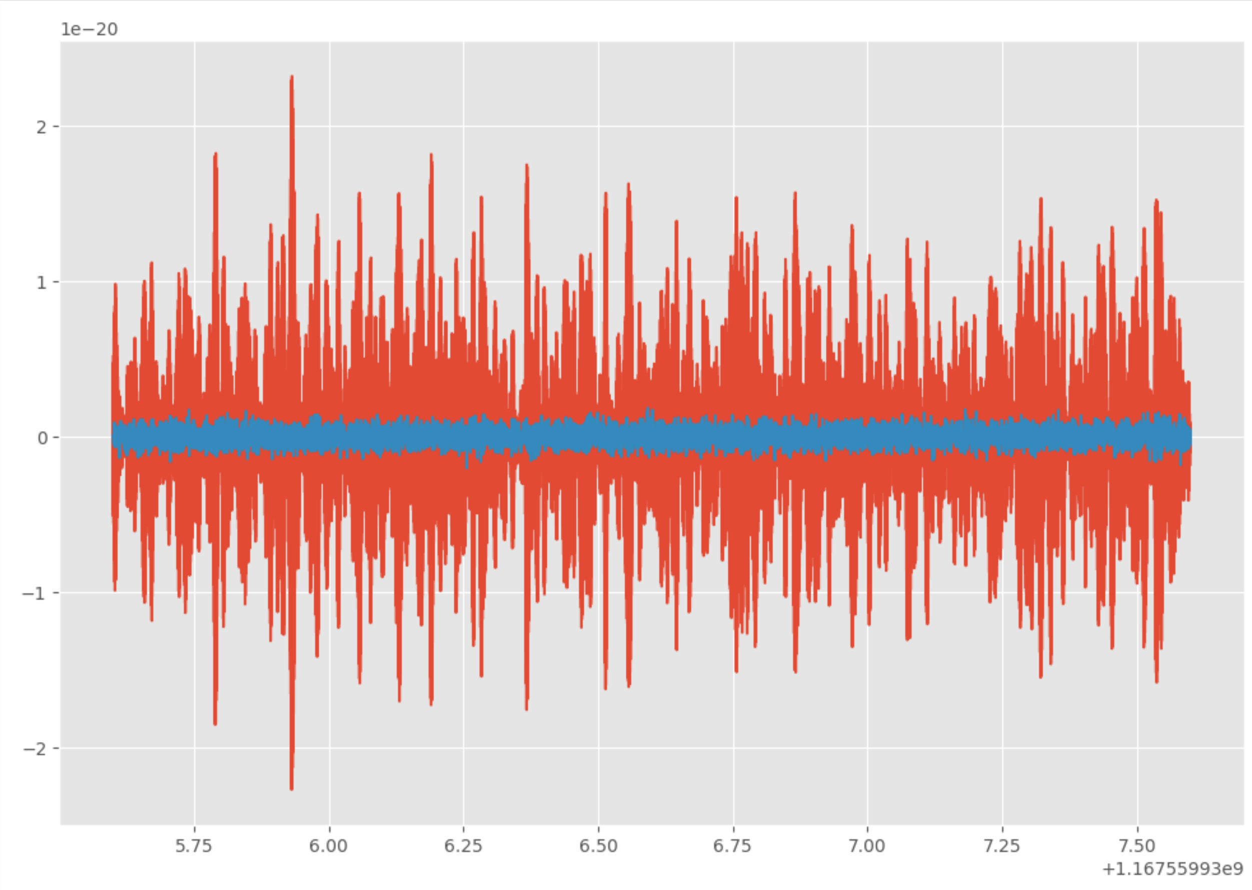
(continued from previous page)

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import pylab
import os
%matplotlib inline
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (12.0, 8.0)
mpl_logger = logging.getLogger("matplotlib")
mpl_logger.setLevel(logging.WARNING)
x=np.zeros(data.GetSize())
y=np.zeros(data.GetSize())
yw=np.zeros(dataw.GetSize())
yww=np.zeros(dataww.GetSize())

for i in range(dataw.GetSize()):
    x[i]=data.GetX(i)
    y[i]=data.GetY(0,i)
    yw[i]=dataw.GetY(0,i)
    yww[i]=dataww.GetY(0,i)

fig, ax = plt.subplots()

ax.plot(x, y,  label='Raw data')
ax.plot(x, yww, label='D-whitened data')
ax.plot(x, yw, label='whitened data')

ax.legend()
plt.show()
```



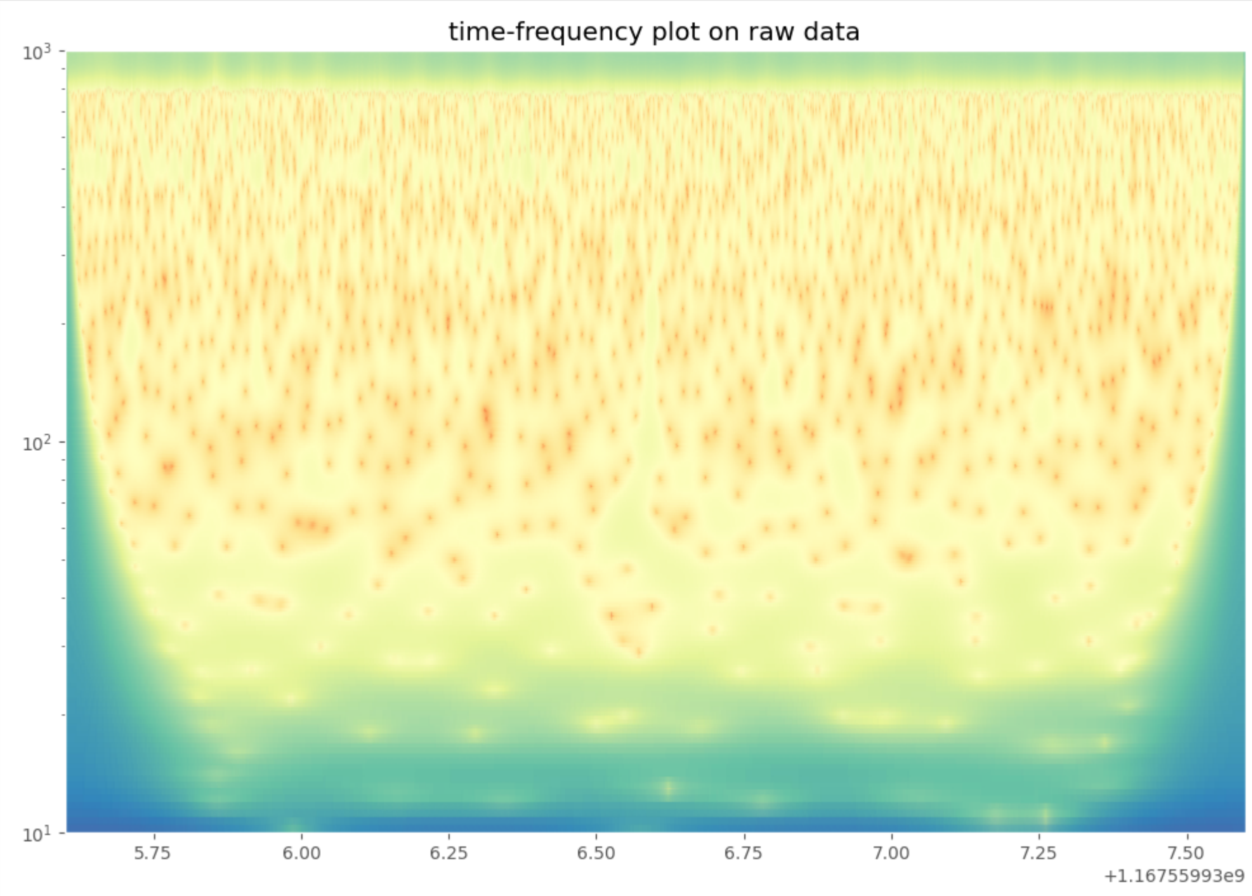
[9]:
```
fig, ax = plt.subplots()
ax.plot(x, yww, label='D-whitened data')
ax.plot(x, yw, label='whitened data')
```

[9]:
```
[<matplotlib.lines.Line2D at 0x7ff8dbc57ad0>]
```

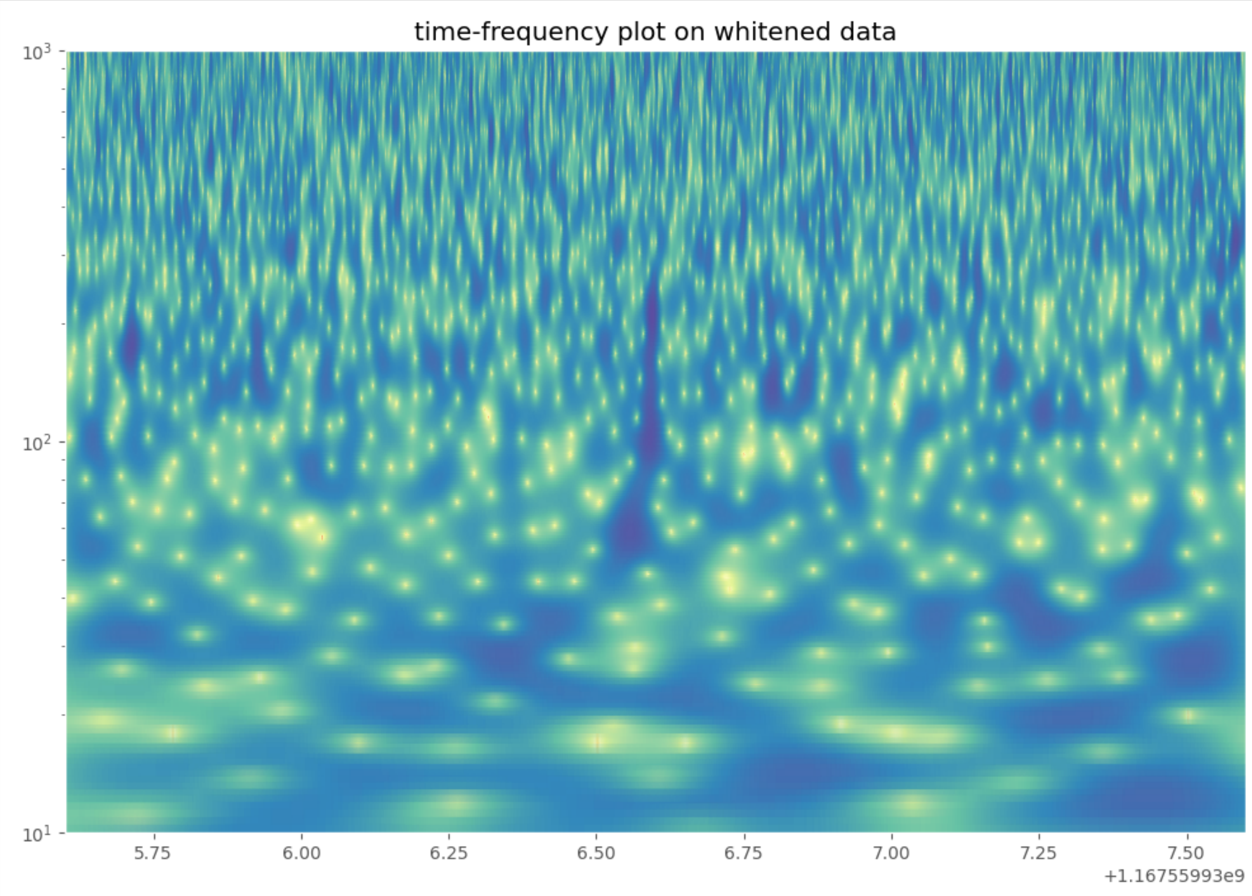


[20]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from scipy import signal
from matplotlib.colors import LogNorm

def prepareImage_gw(x,y,fs,title="title"):
    w = 10.
    freq = np.linspace(1, fs/2, int(fs/2))
    widths = w*fs / (2*freq*np.pi)
    z = np.abs(signal.cwt(y, signal.morlet2, widths, w=w))**2

    plt.pcolormesh(x, freq,z,cmap='Spectral',shading='gouraud',alpha=0.95,
↪norm=LogNorm())
    plt.yscale('log')
    plt.ylim(10, 1000)
    plt.title(str(title))
    plt.show()

    return
```

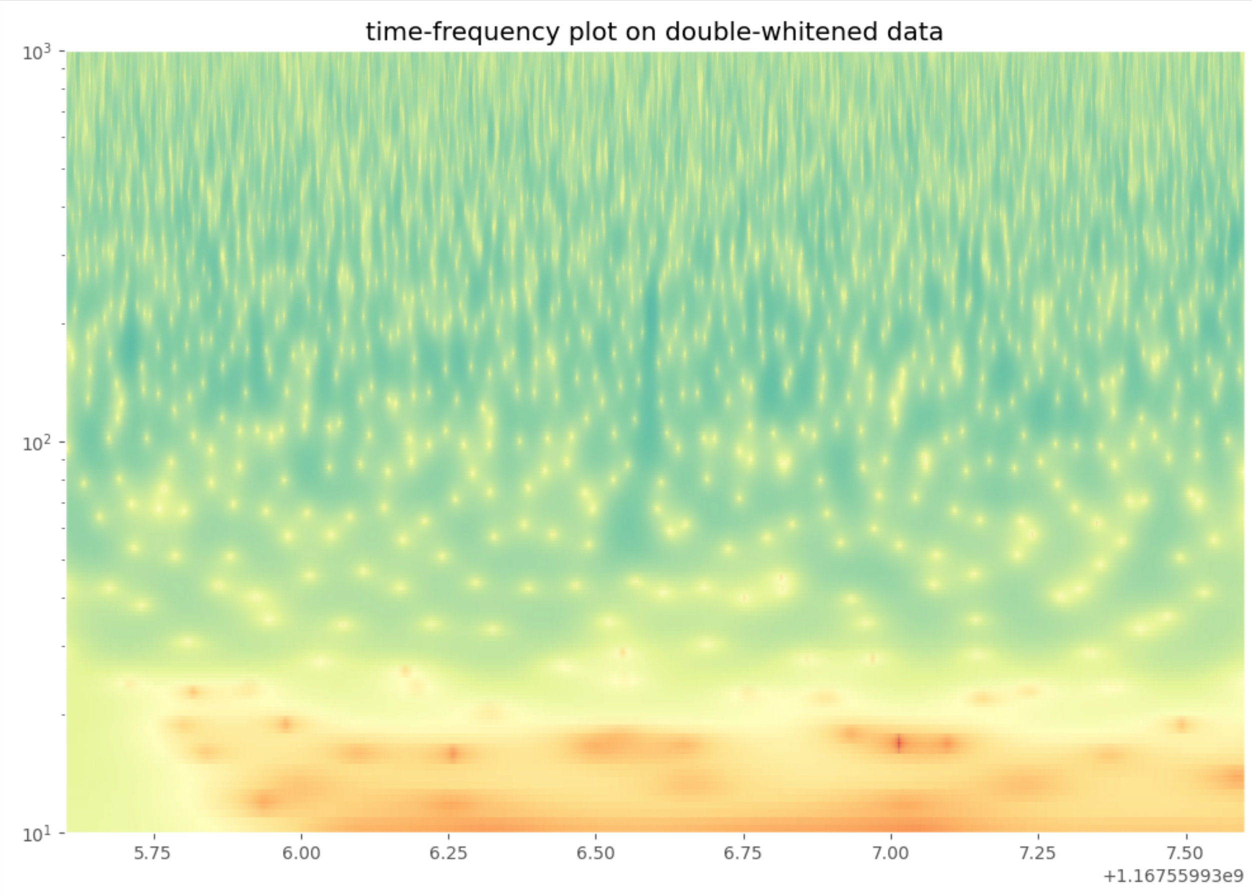
[21]:
```
prepareImage_gw(x,y,par.sampling,"time-frequency plot on raw data")
```



[23]:
```
prepareImage_gw(x,yw,par.sampling,"time-frequency plot on whitened data")
```



[24]:
```
prepareImage_gw(x,yww,par.sampling,"time-frequency plot on double-whitened data")
```



[25]:

```
datasize=data.GetSize()
yr=np.zeros(data.GetSize())
sigma=whiten.GetSigma()

wt = WaveletTransform.BsplineC309
WT = WaveletTransform(datasize, wt)
t =WaveletThreshold.dohonojohnston
wavthres = WaveletThreshold(datasize, 1, sigma);

WT.Forward(dataw);
wavthres(dataw, t);
WT.Inverse(dataw);
for i in range(data.GetSize()):
    x[i]=data.GetX(i)
    yr[i]=dataw.GetY(0,i)
```

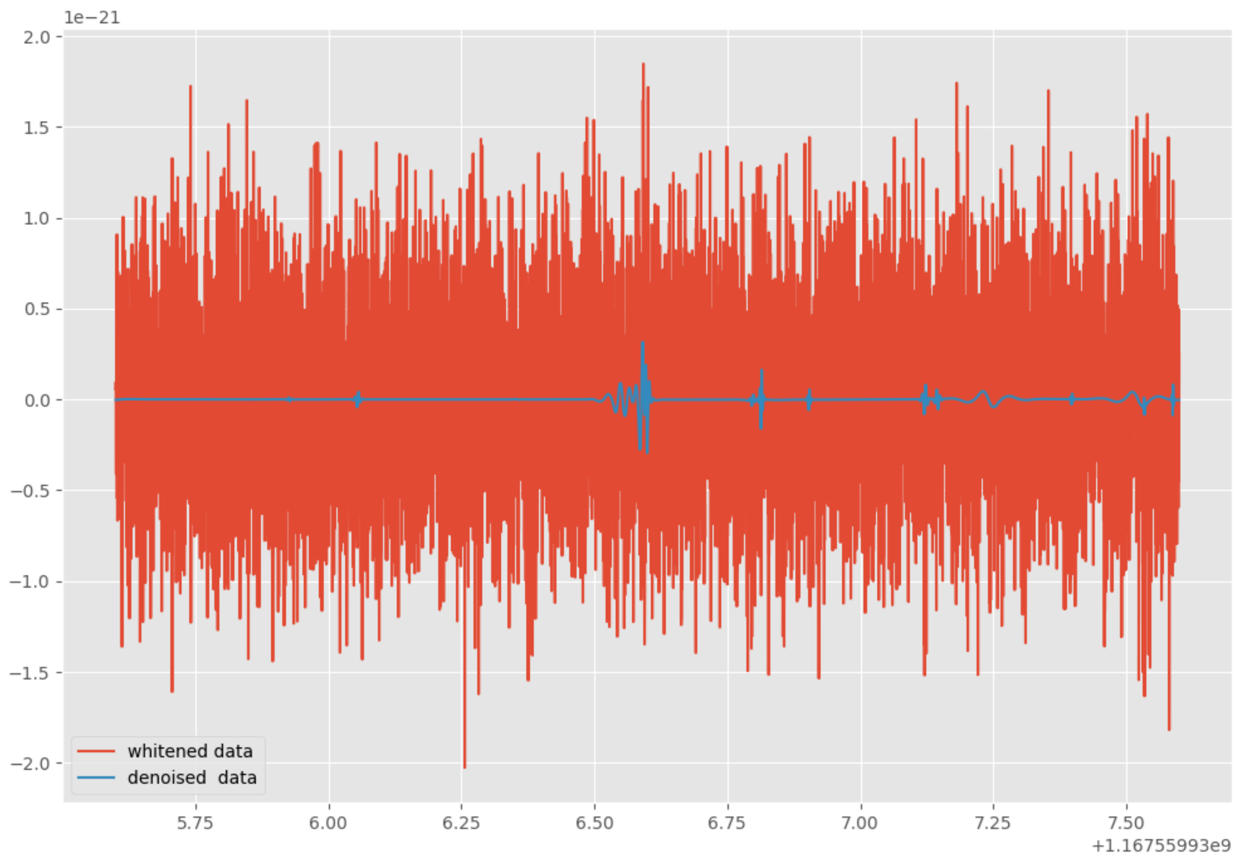
[26]:

```


fig, ax = plt.subplots()

ax.plot(x, yw,  label='whitened data')
ax.plot(x, yr, label='denoised  data')

ax.legend()
```

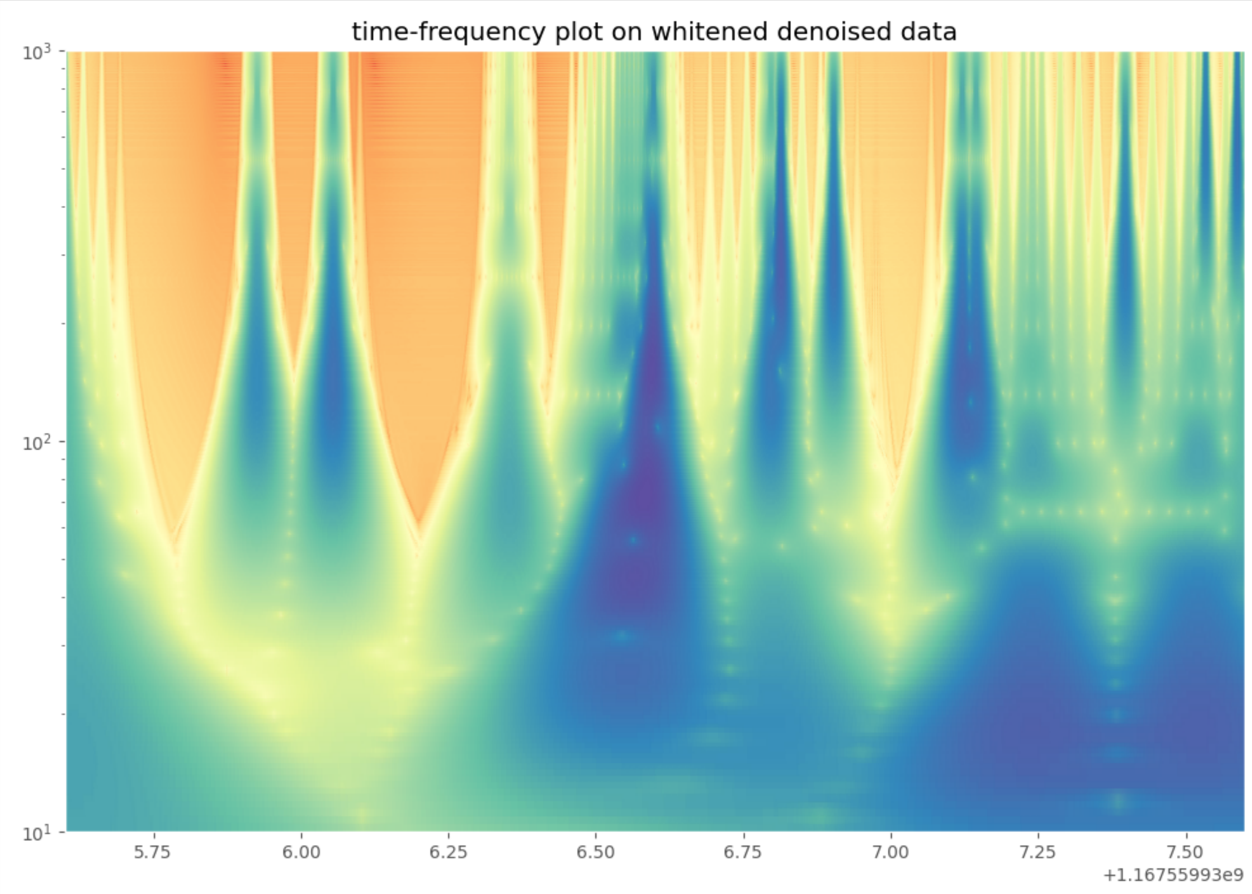
(continues on next page)

(continued from previous page)

```
plt.show()
```

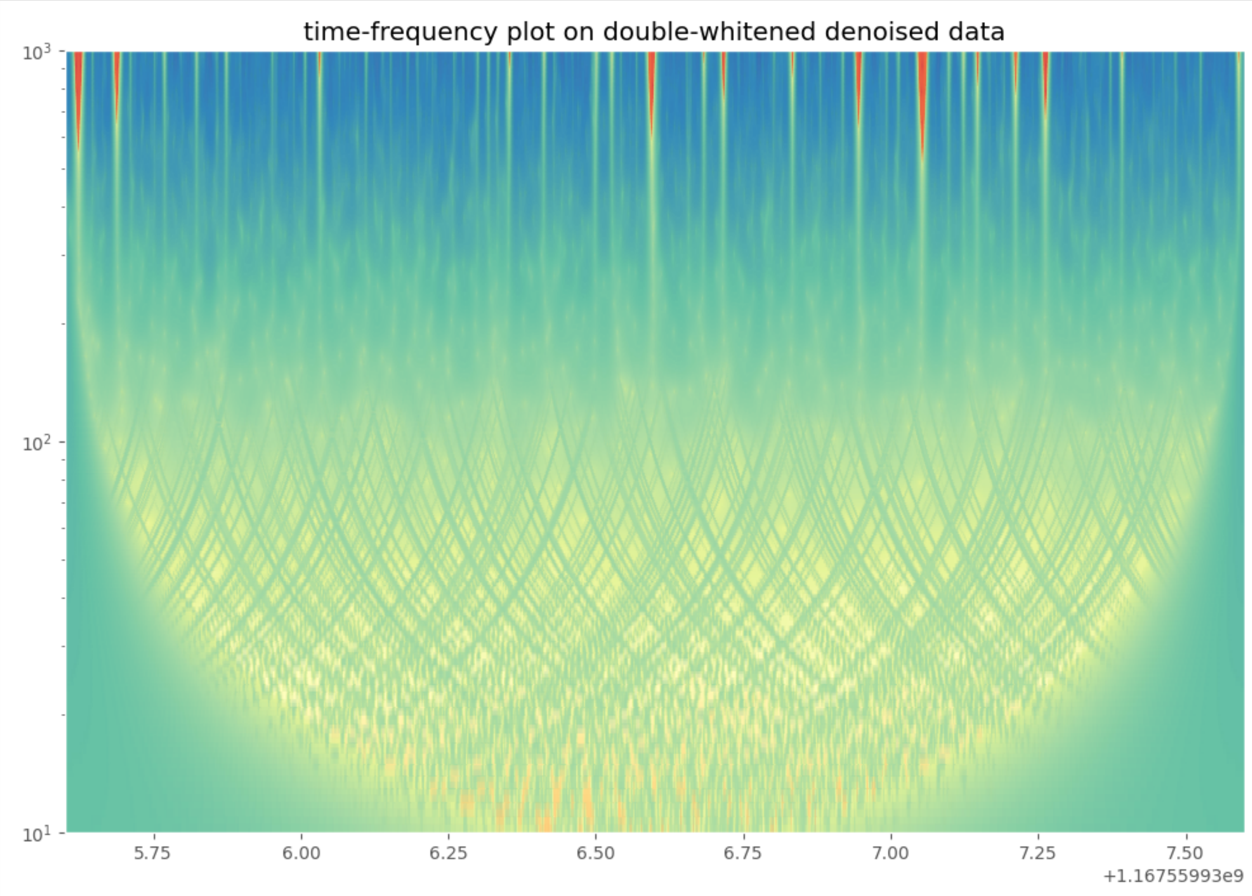


[27]:
```
prepareImage_gw(x,yr,par.sampling,"time-frequency plot on whitened denoised data")
```

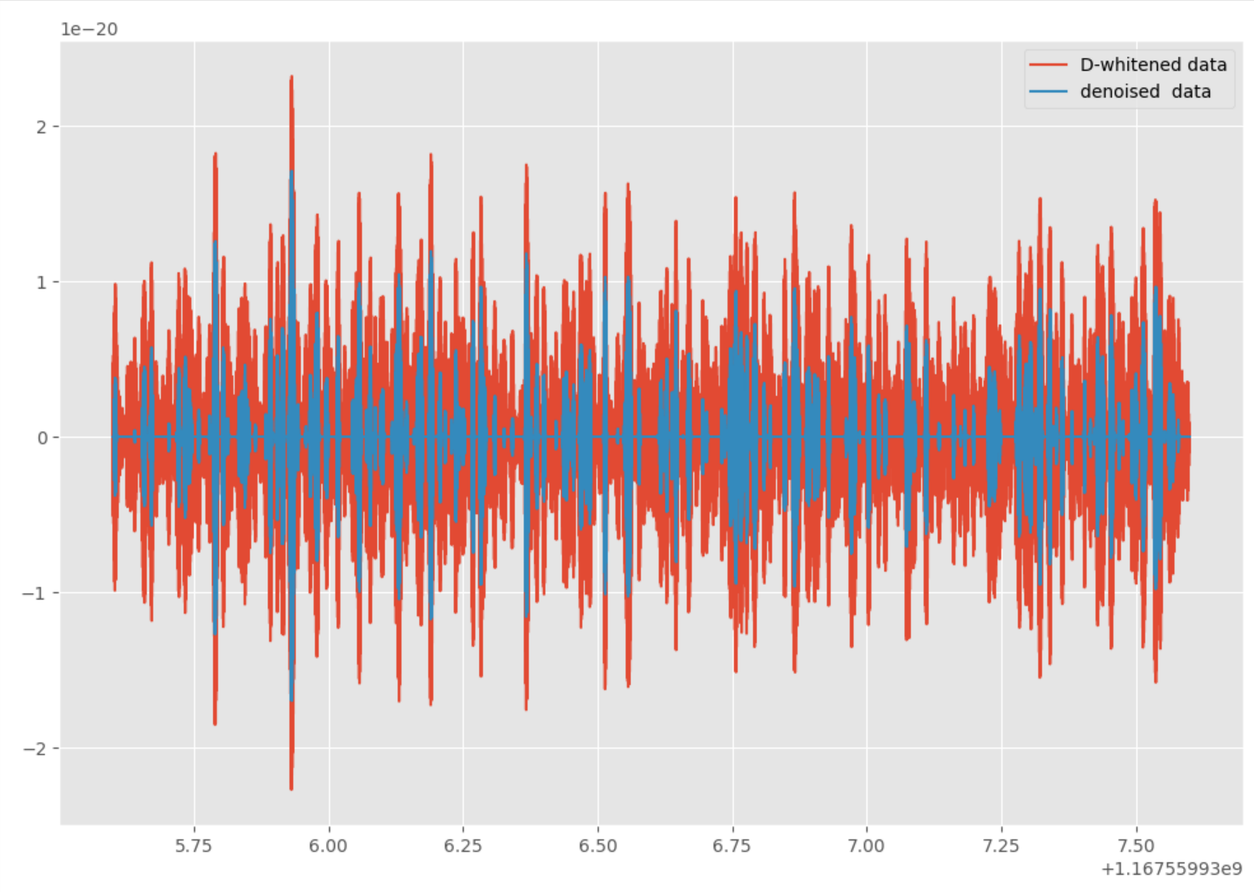


[28]:
```
datasize=data.GetSize()
yr=np.zeros(data.GetSize())
sigma=whiten.GetSigma()

wt = WaveletTransform.BsplineC309
WT = WaveletTransform(datasize, wt)
t =WaveletThreshold.dohonojohnston
wavthres = WaveletThreshold(datasize, 1, sigma);

WT.Forward(dataww);
wavthres(dataww, t);
WT.Inverse(dataww);
for i in range(data.GetSize()):
    x[i]=data.GetX(i)
    yr[i]=dataww.GetY(0,i)
```

[29]:
```
prepareImage_gw(x,yr,par.sampling,"time-frequency plot on double-whitened denoised␣
↪data")
```



[30]:
```
fig, ax = plt.subplots()
ax.plot(x,yww, label='D-whitened data')

ax.plot(x,yr, label='denoised  data')

ax.legend()
plt.show()
```




```
[ ]:
```

1.3 WDF source code

The library's design splits the WDF pipeline into four directories: config, observers, structures, processes

1.3.1 WDF source code

The config functions

parameters

parametersFilePersistence

The observers functions

observer

class observers.observer.**Observer**

The observer class

observable

class observers.observable.**Observable**

This class registers and updates various observers

register(*observer*)

This method registers an observer

Parameters **observer** (*object*) – An observer to be registered

unregister(*observer*)

This methods unregisters an observer

Parameters **observer** (*object*) – An observer to be unregistered

unregister_all()

This method unregisters all observers

update_observers(**args*, ***kwargs*)

This method calls an update function for each observers with various parameters

Parameters

- **args** (*object*) – First parameter of the update function for the given observer
- **kwargs** (*object*) – The following parameters of the update function for the given observer

Returns The object with triggers; type of object depends on the observer

PrintFileObserver

class observers.PrintFileObserver.**PrintTriggers**(*par*)

PrintFilePEObserver

ParameterEstimationObserver

SegmentsObserver

SingleEventPrintFileObserver

class observers.SingleEventPrintFileObserver.**SingleEventPrintTriggers**(*par*, *fullPrint=0*)

The class defining methods to save single event

update(*CEV*)

This methods saves the triggers to the csv file

Parameters

- **eventPE** (*pytsa object*) – Metadata, wavelet coefficients and reconstructed wavelets of the trigger
- **CEV** (*pytsa object*) – pytsa object that contains metadata, wavelet coefficients and reconstructed wavelets of the trigger.

wdfWorkerObserver

The structures functions

array2SeqView

ClusteredEvent

eventPE

class `structures.eventPE.`**eventPE**(*gps*, *gpsPeak*, *duration*, *EnWDF*, *snrMean*, *snrPeak*, *freqMin*, *freqMean*, *freqMax*, *freqPeak*, *wave*, *coeff*, *Icoeff*)

This class stands for the encapsulation of the trigger data into one object

evCopy(*ev*)

This method copies the parameter of the ev, eventPE object

Parameters

- **ev** (`eventPE`) – The eventPE object to copy parameters from
- **gps** (*float*) – GPS time of the trigger denoting the first gps of analyzing window
- **gps** – GPS time of the trigger denoting the moment it appeared at maximum SNR
- **EnWDF** (*float*) – The Signal to Noise Ratio of the trigger statistics of WDF
- **snrMean** (*float*) – The estimated mean Signal to Noise Ratio of the trigger
- **snrPeak** (*float*) – The estimated Signal to Noise Ratio of the trigger at its peak
- **freqMin** (*float*) – The minimum frequency of the trigger
- **freqMax** (*float*) – The maximum frequency of the trigger
- **freqMean** (*float*) – The mean frequency of the trigger
- **freqPeak** (*float*) – The frequency at the peak of the trigger
- **duration** (*float*) – The time duration of the trigger
- **wave** (*str*) – The type of the wavelet
- **coeff** (*list*) – The list containing wavelet coefficients of the trigger
- **Icoeff** (*list*) – The list containing raw wavelet coefficients of the trigger

update(*gps*, *gpsPeak*, *duration*, *EnWDF*, *snrMean*, *snrPeak*, *freqMin*, *freqMean*, *freqMax*, *freqPeak*, *wave*, *coeff*, *Icoeff*)

This method updates the eventPE object with new parameters

Parameters

- **gps** (*float*) – GPS time of the trigger denoting the first gps of analyzing window
- **gps** – GPS time of the trigger denoting the moment it appeared at maximum SNR
- **EnWDF** (*float*) – The Signal to Noise Ratio of the trigger statistics of WDF
- **snrMean** (*float*) – The estimated mean Signal to Noise Ratio of the trigger
- **snrPeak** (*float*) – The estimated Signal to Noise Ratio of the trigger at its peak
- **freqMin** (*float*) – The minimum frequency of the trigger

- **freqMax** (*float*) – The maximum frequency of the trigger
- **freqMean** (*float*) – The mean frequency of the trigger
- **freqPeak** (*float*) – The frequency at the peak of the trigger
- **duration** (*float*) – The time duration of the trigger
- **wave** (*str*) – The type of the wavelet
- **coeff** (*list*) – The list containing wavelet coefficients of the trigger
- **Icoeff** (*list*) – The list containing raw wavelet coefficients of the trigger

segment

The processes functions

AdaptiveWhitening

Whitening

createsegments

createsegmentsMinMax

`class processes.createsegmentsMinMax.`**`createSegmentsMinMax`**`(`*parameters*`)`

DWhitening

Whitening

StateVectorSegments

`class processes.StateVectorSegments.`**`createSegments`**`(`*parameters*`)`

wdf_reconstruct

`class processes.wdf_reconstruct.`**`wdf_reconstruct`**`(`*parameters, wTh=<sphinx.ext.autodoc.importer._MockObject object>*`)`

The main WDF class responsible for the communication with the p4TSA library regarding the application of WDF onto data

FindEvents()

This method calls wdf2reconstruct function from pytsa to search for triggers in the data

Returns trigger

Process()

This method calls wdf2reconstruct function from pytsa to search for triggers in the data If the triggers are found, they are stored in tosend_triggers variable that is later on used for further processing

SetData (*data*)
This methods sets sets the data for the p4TSA wdf2reconstruct class for further search of triggers

Parameters data (*pytsa.SeqViewDouble*) – An pytsa.SeqViewDouble object storing data to be processed

wdf

class processes.wdf.**wdf** (*WdfParams: <sphinx.ext.autodoc.importer._MockObject object at 0x7f9b91530790>, wTh=<sphinx.ext.autodoc.importer._MockObject object>*)
The main WDF class responsible for the communication with the p4TSA library regarding the application of WDF onto data

FindEvents ()
This method calls wdf2classify function from pytsa to search for triggers in the data

Returns trigger

Process ()
This method calls wdf2classify function from pytsa to search for triggers in the data If the triggers are found, they are stored in tosend_triggers variable that is later on used for further processing

SetData (*data*)
This methods sets sets the data for the p4TSA wdf2classify class for further search of triggers

Parameters data (*pytsa.SeqViewDouble*) – An pytsa.SeqViewDouble object storing data to be processed

wdfUnitWorker

wdfUnitDSWorker

Whitening

class processes.Whitening.**Whitening** (*ARorder*)
This class is responsible for the communiction with whitening functions from pytsa

GetLV ()
This method returns LV object

Returns LV object

GetSigma ()
This method returns the sigma parameter of the Whitening process

Returns The sigma parameter of the whitened data

ParametersEstimate (*data*)
This method estimates parameters of data by calling proper methods from pytsa

Parameters data (*pytsa.SeqViewDouble*) – The Sequence View object containing the data to be processed

ParametersLoad (*ARfile, LVfile*)
This method loads the calculated AR and LV parameter from the file

Parameters

- **ARfile** (*basestring*) – file for AutoRegressive parameters
- **LVfile** (*basestring*) – file for Lattice View parameters

Returns Autoregressive and Lattice View

ParametersSave (*ARfile*, *LVfile*)
This method saves the calculated AR and LV parameter to the file

Parameters

- **ARfile** (*basestring*) – file for AutoRegressive parameters
- **LVfile** (*basestring*) – file for Lattice View parameters

Process (*data*, *dataw*)
This method whitens the data by calling proper function from pytsa

Parameters

- **data** – pytsa.SeqViewDouble
- **dataw** – pytsa.SeqViewDouble

DWhitening

Whitening

The utility functions

Filters

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

c

o

p

s

Index

W