Tuto_2.2_Matched_Filtering_In_action

November 19, 2020

1 Gravitational Wave Open Data Workshop #3

1.1 Tutorial 2.2 PyCBC Tutorial, Matched Filtering in Action

We will be using the PyCBC library, which is used to study gravitational-wave data, find astrophysical sources due to compact binary mergers, and study their parameters. These are some of the same tools that the LIGO and Virgo collaborations use to find gravitational waves in LIGO/Virgo data

In this tutorial we will walk through how find a specific signal in LIGO data. We present matched filtering in PyCBC, which is optimal in the case of Gaussian noise and a known signal model. In reality our noise is not entirely Guassian, and in practice we use a variety of techniques to separate signals from noise in addition to the use of the matched filter.

Additional examples and module level documentation are here

1.2 Installation (execute only if running on a cloud platform!)

[1]:
```
# -- Use the following for Google Colab
! pip install -q 'lalsuite==6.66' 'PyCBC==1.15.3'
```

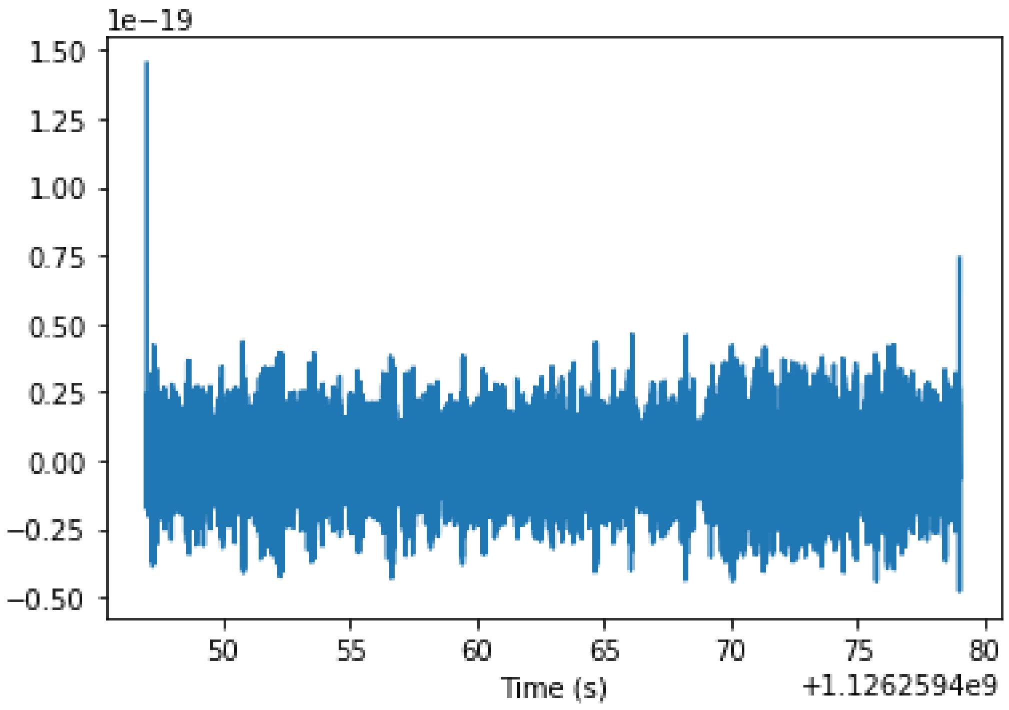
Important: With Google Colab, you may need to restart the runtime after running the cell above.

1.2.1 Looking for a specific signal in the data

If you know what signal you are looking for in the data, then matched filtering is known to be the optimal method in Gaussian noise to extract the siganl. Even when the parameters of the signal are unkown, one can test for each set of parameters one is interesting in finding.

preconditioning the data The purpose of this is to reduce the dynamic range of the data and supress low freqeuncy behavior which can introduce numerical artefacts. We may also wish to reduce the sample rate of the data if high frequency content is not important. PyCBC contains an interface to the GWOSC catalog, so you can easily access the data and parameters of the published gravitational-wave signals

```
[2]: %matplotlib inline

import pylab
from pycbc.catalog import Merger
from pycbc.filter import resample_to_delta_t, highpass

# As an example we use the GW150914 data
merger = Merger("GW150914")

# Get the data from the Hanford detector
strain = merger.strain('H1')

# Remove the low frequency content and downsample the data to 2048Hz
strain = highpass(strain, 15.0)
strain = resample_to_delta_t(strain, 1.0/2048)

pylab.plot(strain.sample_times, strain)
pylab.xlabel('Time (s)')
pylab.show()
```

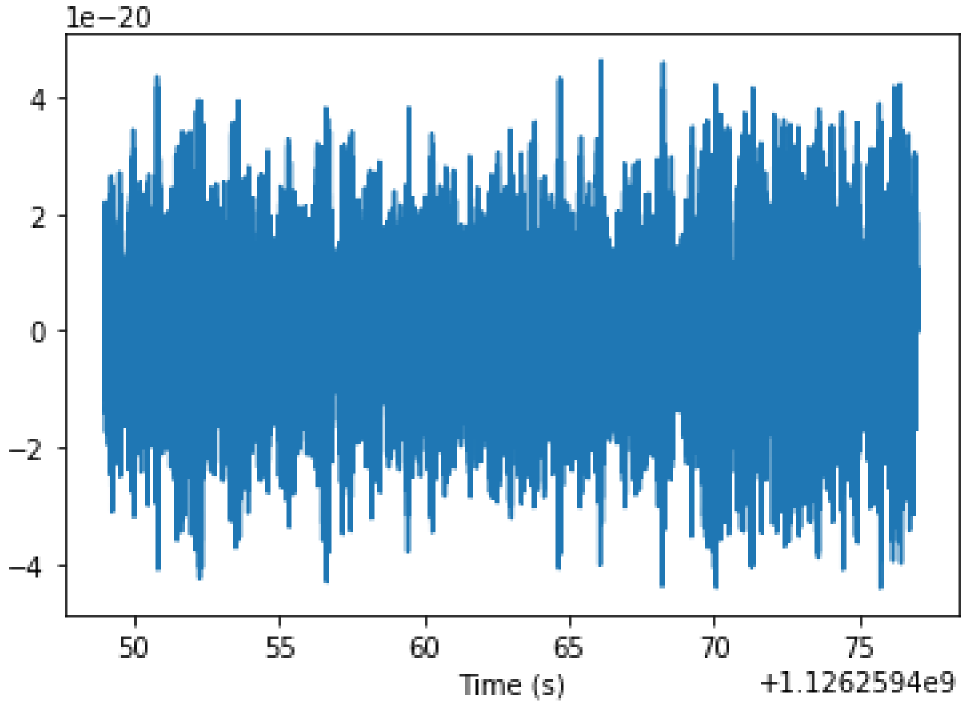


Note: To read data from a local file instead of from the GWOSC server, we can use the pycbc.frame.read_frame(file, channel_name) method.

filter wraparound Note the spike in the data at the boundaries. This is caused by the highpass and resampling stages filtering the data. When the filter is applied to the boundaries, it wraps around to the beginning of the data. Since the data itself has a discontinuity (i.e. it is not cyclic) the filter itself will ring off for a time up to the length of the filter.

Even if a visible transient is not seen, we want to avoid filters that act on times which are not causally connect. To avoid this we trim the ends of the data sufficiently to ensure that they do not wraparound the input. We will enforce this requirement in all steps of our filtering.

[3]:
```
# Remove 2 seconds of data from both the beginning and end
conditioned = strain.crop(2, 2)

pylab.plot(conditioned.sample_times, conditioned)
pylab.xlabel('Time (s)')
pylab.show()
```



calculate the power spectral density Optimal matched filtering requires weighting the frequency components of the potential signal and data by the noise amplitude. We can view this as filtering the data with the time series equivelant of 1 / PSD. To ensure that we can control the effective length of the filter, we window the time domain equivalent of the PSD to a specific length. This has the effect of losing some information about line behavior in the detector, however, since our signals span a large frequency range, and lines are narrow, this is a negligible effect.

Important note: Computing a PSD from data that might contain signals, non-Gaussianities and non-stationarities is not trivial. In this example we use Welch's method to obtain a PSD estimate.

PyCBC's PSD module contains tools for measuring PSDs, or directly using pre-generated PSDs.

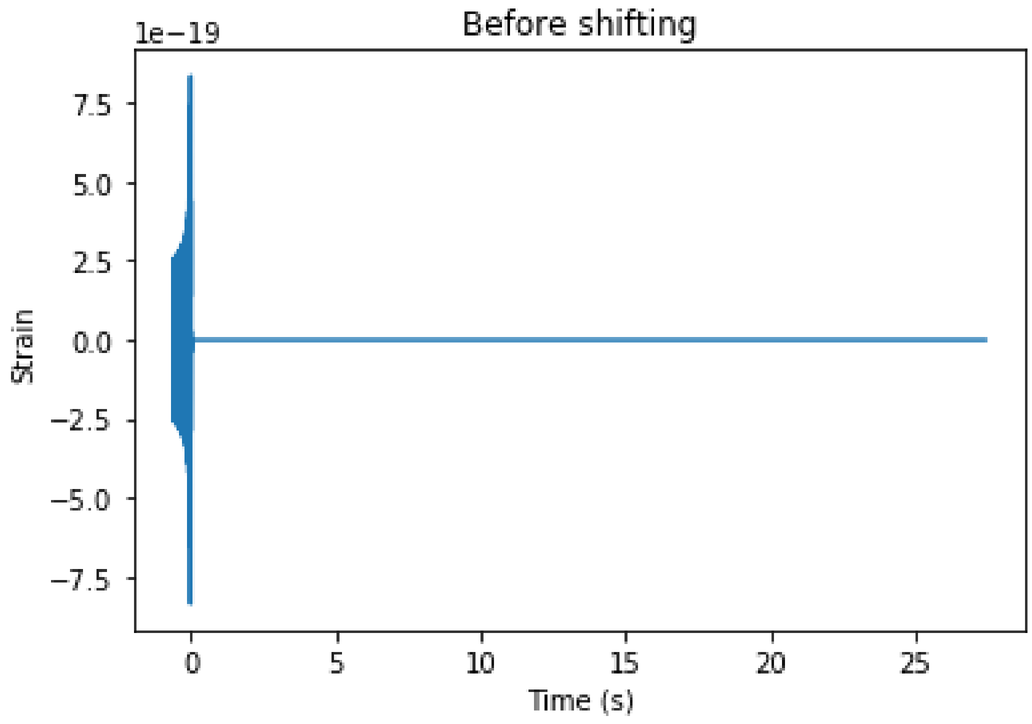
[4]:
```
from pycbc.psd import interpolate, inverse_spectrum_truncation
# Estimate the power spectral density

# We use 4 second samples of our time series in Welch method.
psd = conditioned.psd(4)

# Now that we have the psd we need to interpolate it to match our data
# and then limit the filter length of 1 / PSD. After this, we can
# directly use this PSD to filter the data in a controlled manner
psd = interpolate(psd, conditioned.delta_f)

# 1/PSD will now act as a filter with an effective length of 4 seconds
# Since the data has been highpassed above 15 Hz, and will have low values
# below this we need to inform the function to not include frequencies
# below this frequency.
psd = inverse_spectrum_truncation(psd, 4 * conditioned.sample_rate,
                                  low_frequency_cutoff=15)
```

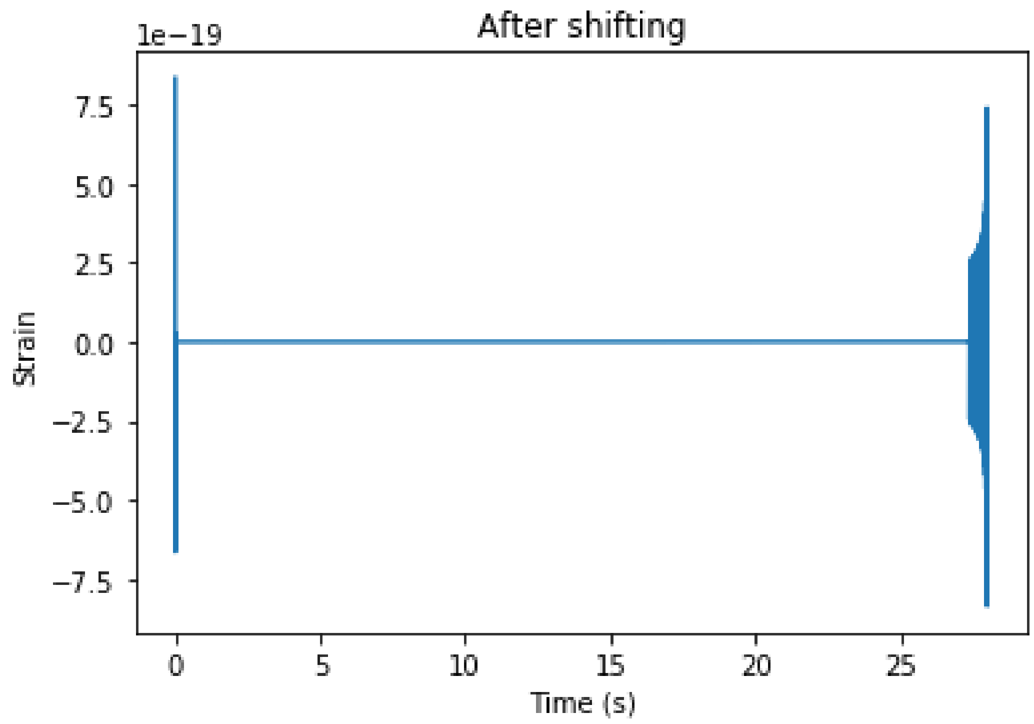
make your signal model Conceptually, matched filtering involves laying the potential signal over your data and integrating (after weighting frequencies correctly). If there is a signal in the data that aligns with your 'template', you will get a large value when integrated over.

[5]:
```
from pycbc.waveform import get_td_waveform
# In this case we "know" what the signal parameters are. In a search
# we would grid over the parameters and calculate the SNR time series
# for each one

# We'll assume equal masses, and non-rotating black holes which is within the␣
 ↪posterior probability
# of GW150914.
m = 36 # Solar masses
hp, hc = get_td_waveform(approximant="SEOBNRv4_opt",
                     mass1=m,
                     mass2=m,
                     delta_t=conditioned.delta_t,
                     f_lower=20)

# We will resize the vector to match our data
hp.resize(len(conditioned))

# The waveform begins at the start of the vector, so if we want the
# SNR time series to correspond to the approximate merger location
# we need to shift the data so that the merger is approximately at the
# first bin of the data.
```

```python
# The cyclic_time_shift method shifts the timeseries by a given amount of time.
# It treats the data as if it were on a ring so points shifted off the end
# of the series reappear at the start. Note that time stamps are *not* in
# general affected (as the start time of the full array is shifted),
# but the index of each point in the vector is.
#
# By convention waveforms returned from `get_td_waveform` have their
# merger stamped with time zero, so we can use the start time to
# shift the merger into position
pylab.figure()
pylab.title('Before shifting')
pylab.plot(hp.sample_times, hp)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')

template = hp.cyclic_time_shift(hp.start_time)

pylab.figure()
pylab.title('After shifting')
pylab.plot(template.sample_times, template)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')
```

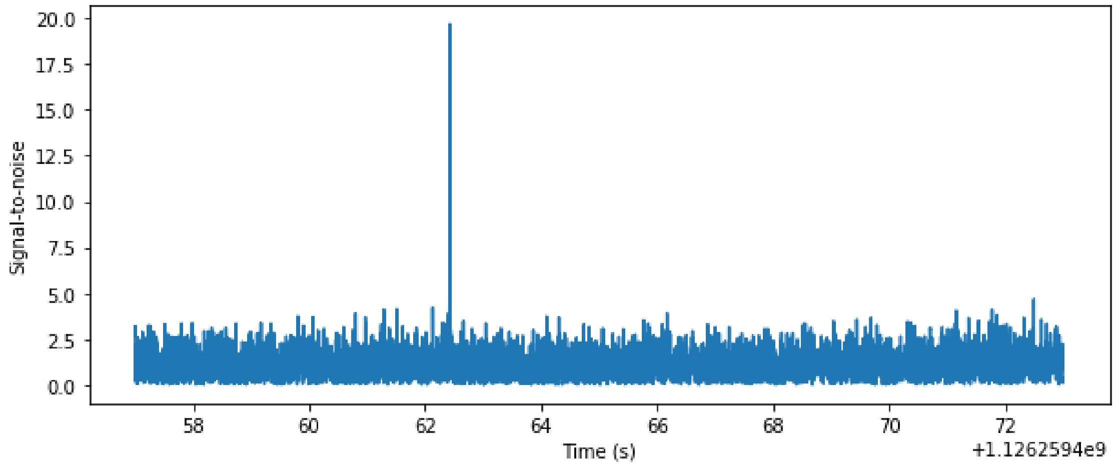
```
[5]: Text(0, 0.5, 'Strain')
```





calculating the signal-to-noise time series In this section we will now calculate the signal-to-noise time series for our template. We'll take care to handle issues of filter corruption / wraparound by truncating the output time series. We need to account for both the length of the template and 1 / PSD.

[6]:
```
from pycbc.filter import matched_filter
import numpy

snr = matched_filter(template, conditioned,
                     psd=psd, low_frequency_cutoff=20)

# Remove time corrupted by the template filter and the psd filter
# We remove 4 seonds at the beginning and end for the PSD filtering
# And we remove 4 additional seconds at the beginning to account for
# the template length (this is somewhat generous for
# so short a template). A longer signal such as from a BNS, would
# require much more padding at the beginning of the vector.
snr = snr.crop(4 + 4, 4)

# Why are we taking an abs() here?
# The `matched_filter` function actually returns a 'complex' SNR.
# What that means is that the real portion correponds to the SNR
# associated with directly filtering the template with the data.
# The imaginary portion corresponds to filtering with a template that
# is 90 degrees out of phase. Since the phase of a signal may be
# anything, we choose to maximize over the phase of the signal.
pylab.figure(figsize=[10, 4])
pylab.plot(snr.sample_times, abs(snr))
pylab.ylabel('Signal-to-noise')
pylab.xlabel('Time (s)')
pylab.show()

peak = abs(snr).numpy().argmax()
snrp = snr[peak]
time = snr.sample_times[peak]

print("We found a signal at {}s with SNR {}".format(time,
                                                    abs(snrp)))
```



```
We found a signal at 1126259462.4248047s with SNR 19.677089013145913
```

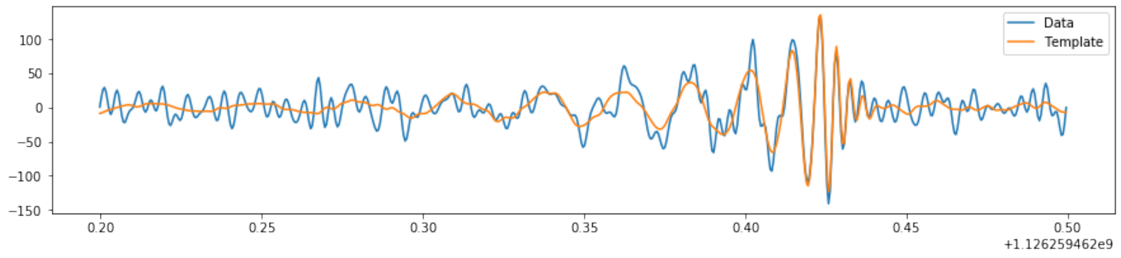
1.2.2 Aligning and Subtracting the Proposed Signal

In the previous section we found a peak in the signal-to-noise for a proposed binary black hole merger. We can use this SNR peak to align our proposal to the data, and to also subtract our proposal from the data.

[7]:
```
from pycbc.filter import sigma
# The time, amplitude, and phase of the SNR peak tell us how to align
# our proposed signal with the data.
# Shift the template to the peak time
dt = time - conditioned.start_time
aligned = template.cyclic_time_shift(dt)

# scale the template so that it would have SNR 1 in this data
aligned /= sigma(aligned, psd=psd, low_frequency_cutoff=20.0)

# Scale the template amplitude and phase to the peak value
aligned = (aligned.to_frequencyseries() * snrp).to_timeseries()
aligned.start_time = conditioned.start_time
```

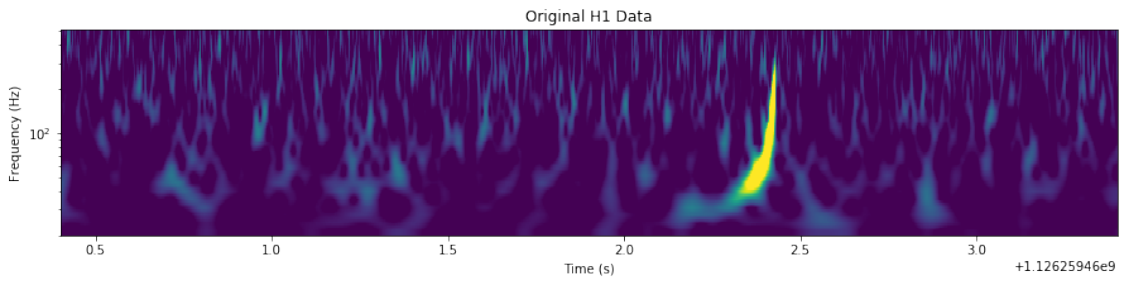
Visualize the overlap between the signal and data To compare the data an signal on equal footing, and to concentrate on the frequency range that is important. We will whiten both the template and the data, and then bandpass both the data and template between 30-300 Hz. In this way, any signal that is in the data is transformed in the same way that the template is.

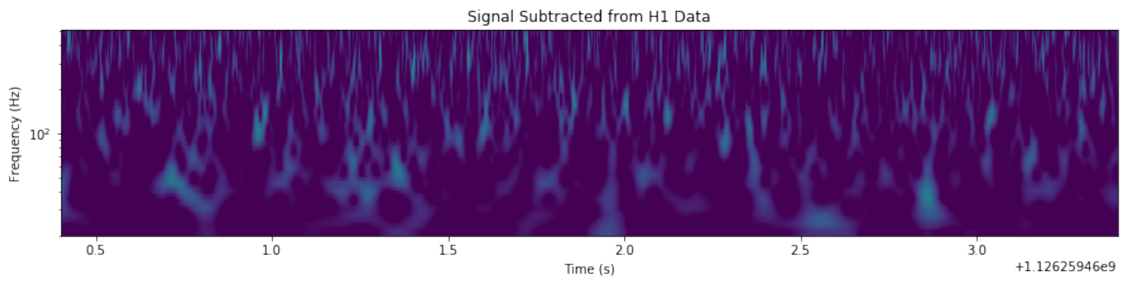
[8]:
```
# We do it this way so that we can whiten both the template and the data
white_data = (conditioned.to_frequencyseries() / psd**0.5).to_timeseries()
white_template = (aligned.to_frequencyseries() / psd**0.5).to_timeseries()

white_data = white_data.highpass_fir(30., 512).lowpass_fir(300, 512)
white_template = white_template.highpass_fir(30, 512).lowpass_fir(300, 512)

# Select the time around the merger
white_data = white_data.time_slice(merger.time-.2, merger.time+.1)
white_template = white_template.time_slice(merger.time-.2, merger.time+.1)

pylab.figure(figsize=[15, 3])
pylab.plot(white_data.sample_times, white_data, label="Data")
pylab.plot(white_template.sample_times, white_template, label="Template")
pylab.legend()
pylab.show()
```



Subtracting the signal from the data Now that we've aligned the template we can simply subtract it. Let's see below how that looks in the time-frequency plots!

```
[9]: subtracted = conditioned - aligned

     # Plot the original data and the subtracted signal data

     for data, title in [(conditioned, 'Original H1 Data'),
                         (subtracted, 'Signal Subtracted from H1 Data')]:

         t, f, p = data.whiten(4, 4).qtransform(.001,
                                                 logfsteps=100,
                                                 qrange=(8, 8),
                                                 frange=(20, 512))
         pylab.figure(figsize=[15, 3])
         pylab.title(title)
         pylab.pcolormesh(t, f, p**0.5, vmin=1, vmax=6)
         pylab.yscale('log')
         pylab.xlabel('Time (s)')
         pylab.ylabel('Frequency (Hz)')
         pylab.xlim(merger.time - 2, merger.time + 1)
         pylab.show()
```





1.3 Challenge!

Use the methods demonstrated above to see if you can calculate the SNR time series in the following data sets. What is the SNR of each signal? Which template matched best to which data?

Information that may be useful:

- Signals are all placed between 100 and 120 seconds into the frame file.
- You may assume mass1 = mass1 (equal mass) and that each component mass is one of 15, 30, or 45.
- Each file starts at gps time 0, and ends at gps time 128
- The channel name in each file is "H1:TEST-STRAIN"

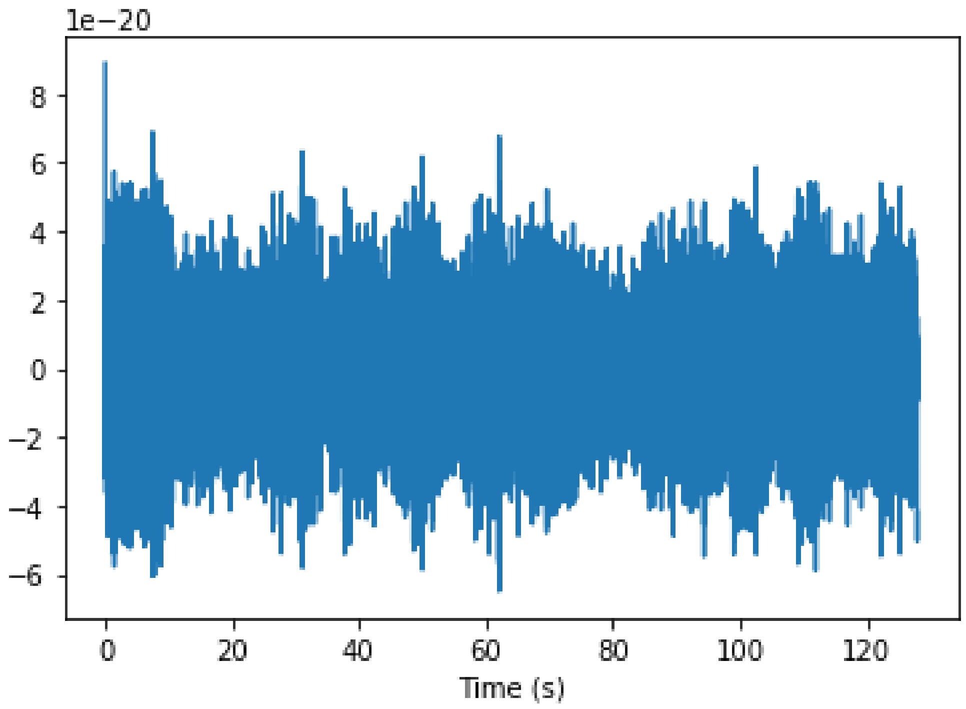
[10]:
```
# Download the challenge set files
from pycbc.frame import read_frame
import urllib

def get_file(fname):
    url = "https://github.com/gw-odw/odw-2020/raw/master/Data/{}"
    url = url.format(fname)
    urllib.request.urlretrieve(url, fname)
    print('Getting : {}'.format(url))

files = ['PyCBC_T2_0.gwf', 'PyCBC_T2_1.gwf', 'PyCBC_T2_2.gwf']

for fname in files:
    get_file(fname)


# An example of how to read the data from these files:
file_name = "PyCBC_T2_0.gwf"

# LOSC bulk data typically uses the same convention for internal channels names
# Strain is typically IFO:LOSC-STRAIN, where IFO can be H1/L1/V1.
channel_name = "H1:TEST-STRAIN"

start = 0
end = start + 128

ts = read_frame(file_name, channel_name, start, end)
```

```
Getting : https://github.com/gw-odw/odw-2020/raw/master/Data/PyCBC_T2_0.gwf
Getting : https://github.com/gw-odw/odw-2020/raw/master/Data/PyCBC_T2_1.gwf
Getting : https://github.com/gw-odw/odw-2020/raw/master/Data/PyCBC_T2_2.gwf
```

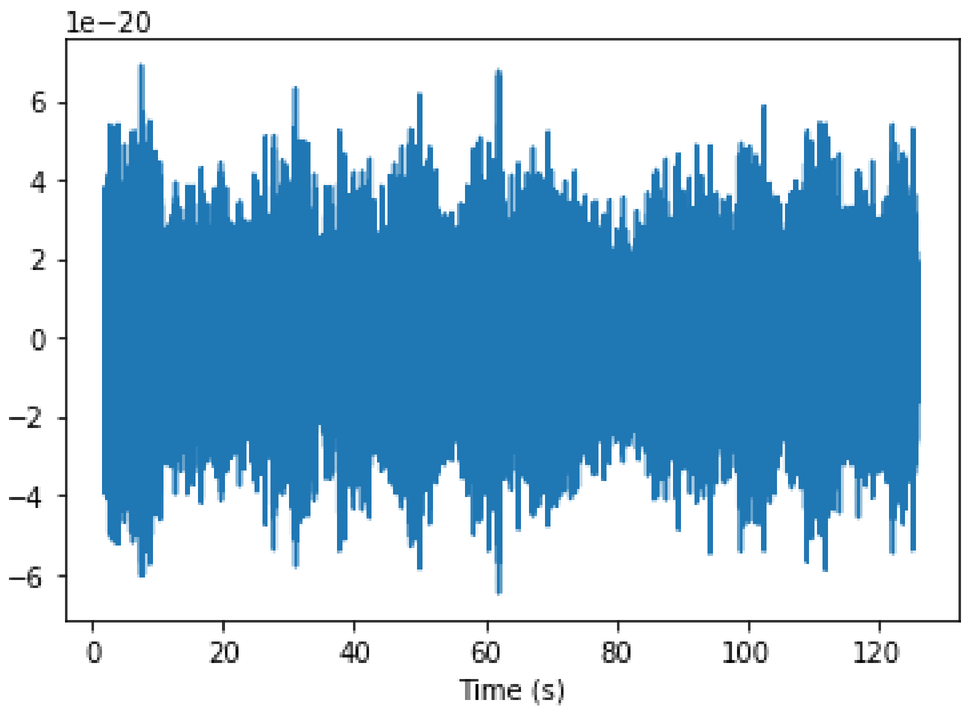
1.3.1 Analysis of PyCBC_T2_0.gwf

[11]:
```
%matplotlib inline

import pylab
from pycbc.catalog import Merger
from pycbc.filter import resample_to_delta_t, highpass
```

```
# As an example we use the GW150914 data
file_name1 = "PyCBC_T2_0.gwf"

channel_name1 = "H1:TEST-STRAIN"

start = 0
end = start + 128

ts1 = read_frame(file_name1, channel_name1, start, end)
# Remove the low frequency content and downsample the data to 2048Hz
strain1 = highpass(ts1, 15.0)
strain1 = resample_to_delta_t(strain1, 1.0/2048)

pylab.plot(strain1.sample_times, strain1)
pylab.xlabel('Time (s)')
pylab.show()
```



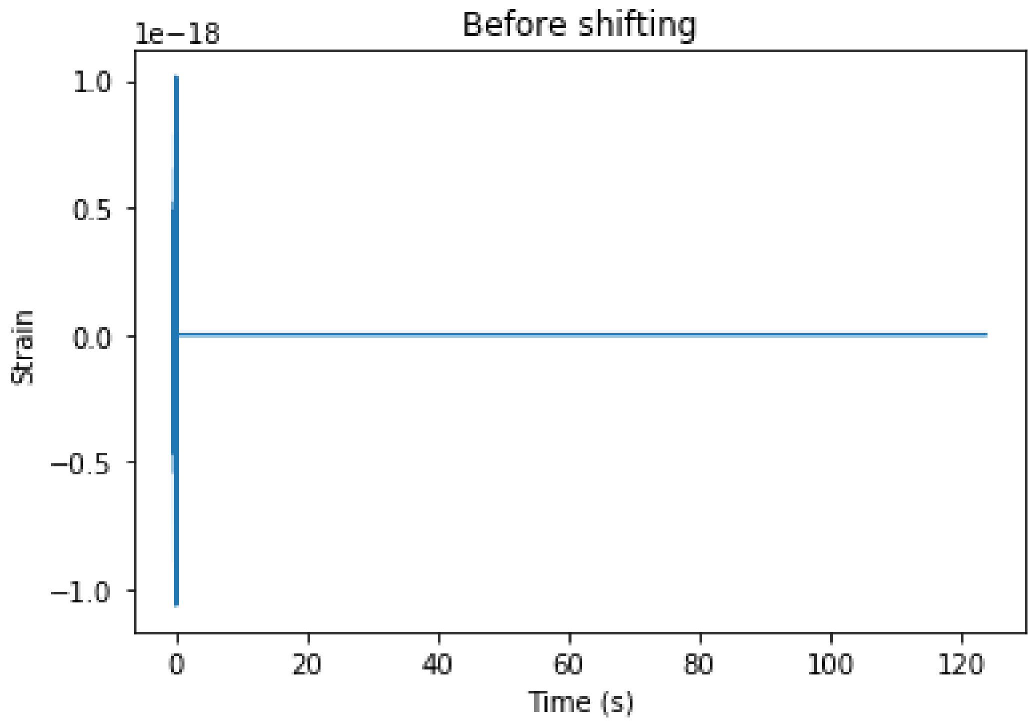
[12]:
```
# Remove 2 seconds of data from both the beginning and end
conditioned1 = strain1.crop(2, 2)

pylab.plot(conditioned1.sample_times, conditioned1)
pylab.xlabel('Time (s)')
pylab.show()
```

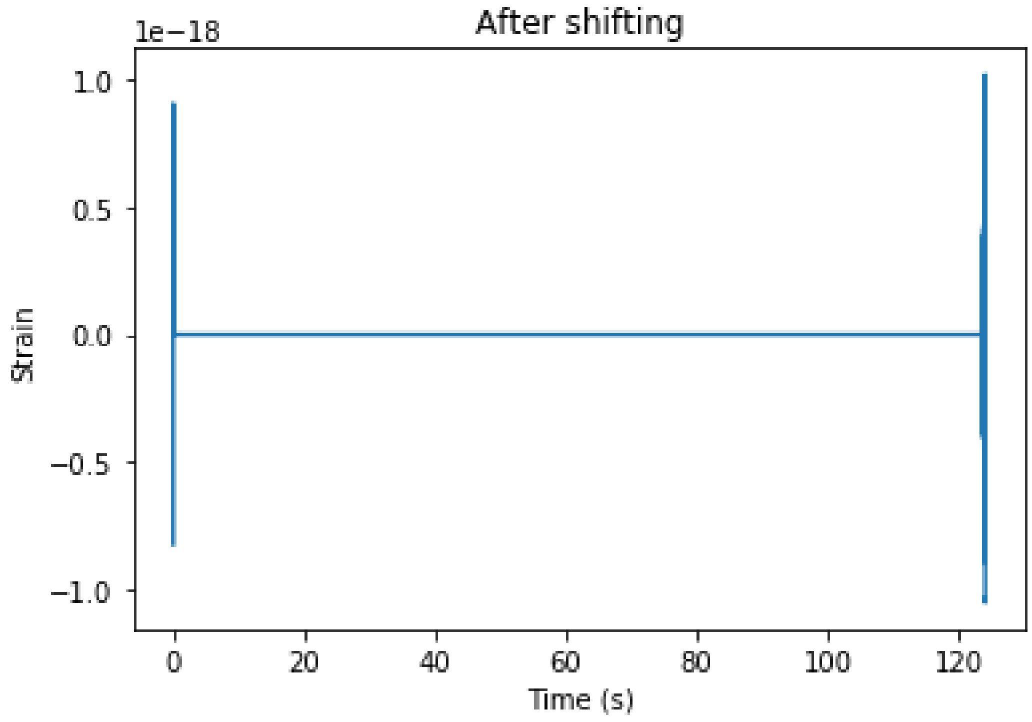


[13]:
```
from pycbc.psd import interpolate, inverse_spectrum_truncation
# Estimate the power spectral density

# We use 4 second samples of our time series in Welch method.
psd1 = conditioned1.psd(4)

# Now that we have the psd we need to interpolate it to match our data
# and then limit the filter length of 1 / PSD. After this, we can
# directly use this PSD to filter the data in a controlled manner
psd1 = interpolate(psd1, conditioned1.delta_f)

# 1/PSD will now act as a filter with an effective length of 4 seconds
# Since the data has been highpassed above 15 Hz, and will have low values
# below this we need to inform the function to not include frequencies
# below this frequency.
psd1 = inverse_spectrum_truncation(psd1, 4 * conditioned1.sample_rate,
                                   low_frequency_cutoff=15)
```

[14]:
```
from pycbc.waveform import get_td_waveform
# In this case we "know" what the signal parameters are. In a search
# we would grid over the parameters and calculate the SNR time series
# for each one
# We'll assume equal masses, and non-rotating black holes which is within the␣
↪posterior probability
# of GW150914.
m = 45 # Solar masses
hp, hc = get_td_waveform(approximant="SEOBNRv4_opt",
                     mass1=m,
                     mass2=m,
                     delta_t=conditioned1.delta_t,
                     f_lower=20)

# We will resize the vector to match our data
hp.resize(len(conditioned1))

# The waveform begins at the start of the vector, so if we want the
# SNR time series to correspond to the approximate merger location
# we need to shift the data so that the merger is approximately at the
# first bin of the data.

# The cyclic_time_shift method shifts the timeseries by a given amount of time.
# It treats the data as if it were on a ring so points shifted off the end
# of the series reappear at the start. Note that time stamps are *not* in
# general affected (as the start time of the full array is shifted),
# but the index of each point in the vector is.
#
# By convention waveforms returned from `get_td_waveform` have their
# merger stamped with time zero, so we can use the start time to
# shift the merger into position
pylab.figure()
pylab.title('Before shifting')
pylab.plot(hp.sample_times, hp)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')

template = hp.cyclic_time_shift(hp.start_time)

pylab.figure()
pylab.title('After shifting')
pylab.plot(template.sample_times, template)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')
```

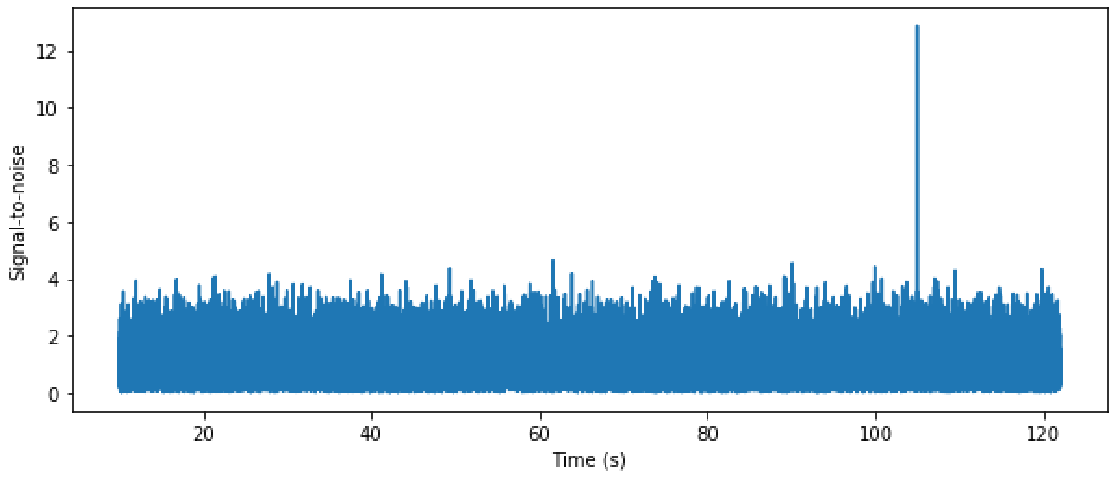
```
[14]: Text(0, 0.5, 'Strain')
```


Before shifting
1e−18
1.0
0.5
0.0
−0.5
−1.0
Strain
0
20
40
60
80
100
120
Time (s)


After shifting
1e−18
1.0
0.5
0.0
−0.5
−1.0
Strain
0
20
40
60
80
100
120
Time (s)

[15]:

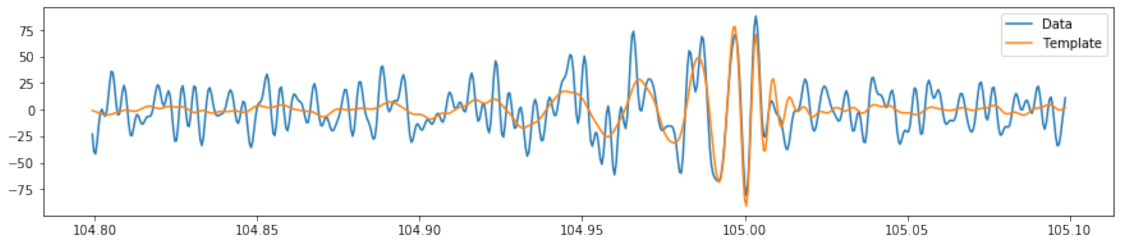
```python
from pycbc.filter import matched_filter
import numpy

snr = matched_filter(template, conditioned1,
                     psd=psd1, low_frequency_cutoff=20)

# Remove time corrupted by the template filter and the psd filter
# We remove 4 seonds at the beginning and end for the PSD filtering
# And we remove 4 additional seconds at the beginning to account for
# the template length (this is somewhat generous for
# so short a template). A longer signal such as from a BNS, would
# require much more padding at the beginning of the vector.
snr = snr.crop(4 + 4, 4)

# Why are we taking an abs() here?
# The `matched_filter` function actually returns a 'complex' SNR.
# What that means is that the real portion correponds to the SNR
# associated with directly filtering the template with the data.
# The imaginary portion corresponds to filtering with a template that
# is 90 degrees out of phase. Since the phase of a signal may be
# anything, we choose to maximize over the phase of the signal.
pylab.figure(figsize=[10, 4])
pylab.plot(snr.sample_times, abs(snr))
pylab.ylabel('Signal-to-noise')
pylab.xlabel('Time (s)')
pylab.show()

peak = abs(snr).numpy().argmax()
snrp = snr[peak]
time = snr.sample_times[peak]

print("We found a signal at {}s with SNR {}".format(time,
                                                    abs(snrp)))
```



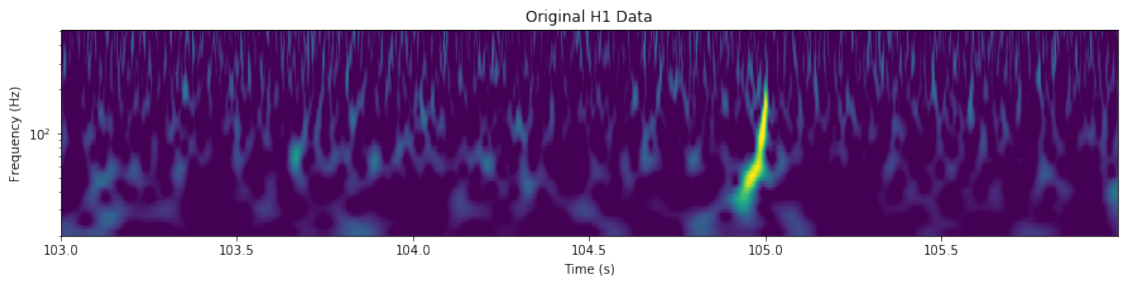
```
We found a signal at 104.99951171875s with SNR 12.872423980605582
```

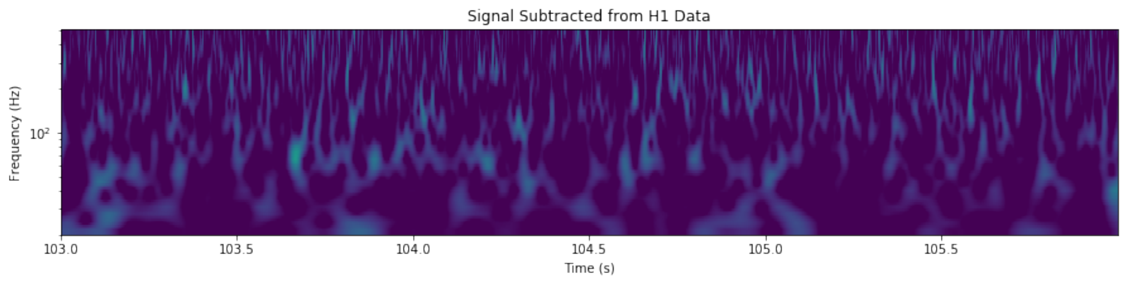
[16]:
```python
from pycbc.filter import sigma
# The time, amplitude, and phase of the SNR peak tell us how to align
# our proposed signal with the data.

# Shift the template to the peak time
dt = time - conditioned1.start_time
aligned = template.cyclic_time_shift(dt)

# scale the template so that it would have SNR 1 in this data
aligned /= sigma(aligned, psd=psd1, low_frequency_cutoff=20.0)

# Scale the template amplitude and phase to the peak value
aligned = (aligned.to_frequencyseries() * snrp).to_timeseries()
aligned.start_time = conditioned1.start_time
```

[17]:
```python
# We do it this way so that we can whiten both the template and the data
white_data = (conditioned1.to_frequencyseries() / psd1**0.5).to_timeseries()
white_template = (aligned.to_frequencyseries() / psd1**0.5).to_timeseries()

white_data = white_data.highpass_fir(30., 512).lowpass_fir(300, 512)
white_template = white_template.highpass_fir(30, 512).lowpass_fir(300, 512)

# Select the time around the merger
white_data = white_data.time_slice(time-.2, time+.1)
white_template = white_template.time_slice(time-.2, time+.1)

pylab.figure(figsize=[15, 3])
pylab.plot(white_data.sample_times, white_data, label="Data")
pylab.plot(white_template.sample_times, white_template, label="Template")
pylab.legend()
pylab.show()
```



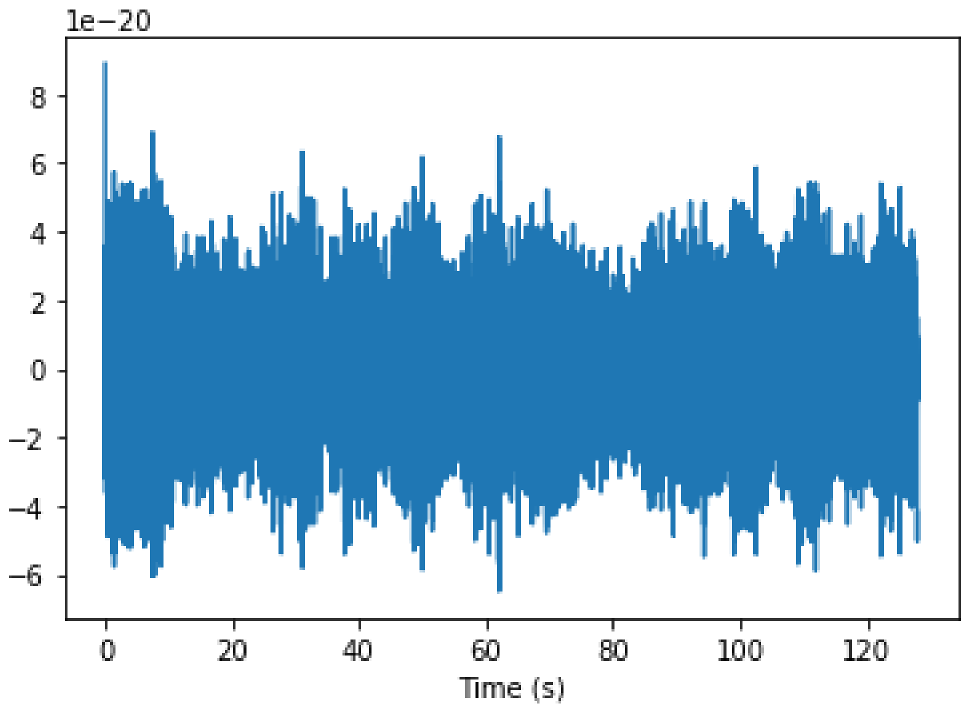
[19]:
```
subtracted = conditioned1 - aligned

# Plot the original data and the subtracted signal data

for data, title in [(conditioned1, 'Original H1 Data'),
                    (subtracted, 'Signal Subtracted from H1 Data')]:

    t, f, p = data.whiten(4, 4).qtransform(.001,
                                                logfsteps=100,
                                                qrange=(8, 8),
                                                frange=(20, 512))
    pylab.figure(figsize=[15, 3])
    pylab.title(title)
    pylab.pcolormesh(t, f, p**0.5, vmin=1, vmax=6)
    pylab.yscale('log')
    pylab.xlabel('Time (s)')
    pylab.ylabel('Frequency (Hz)')
    pylab.xlim(time - 2, time + 1)
    pylab.show()
```



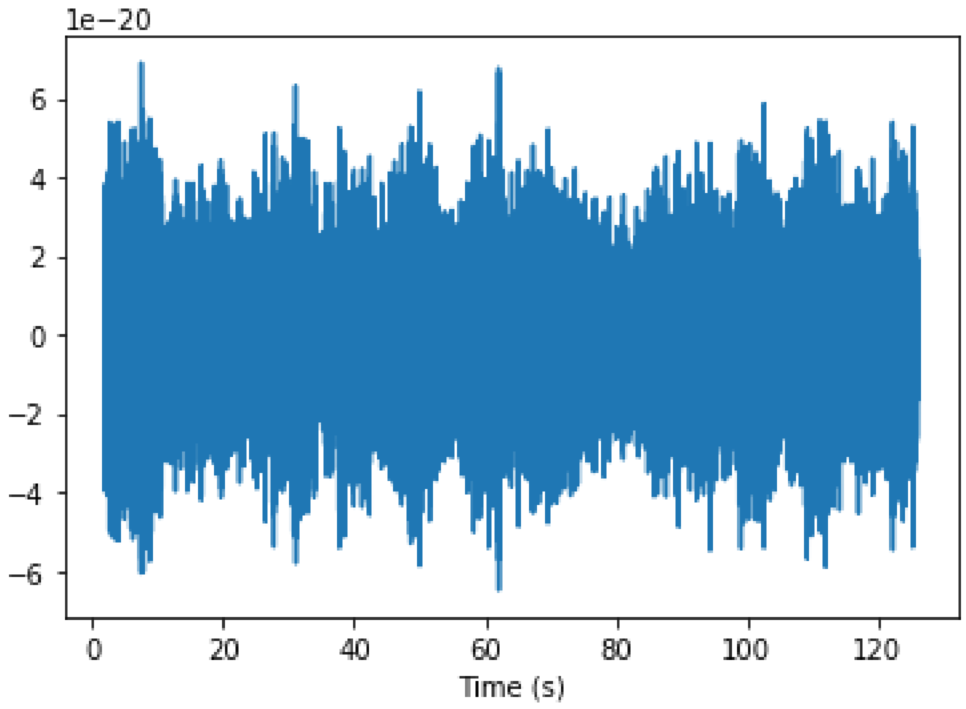


1.3.2 Analysis of PyCBC_T2_1.gwf

[20]:
```
%matplotlib inline

import pylab
from pycbc.catalog import Merger
from pycbc.filter import resample_to_delta_t, highpass

# As an example we use the GW150914 data
file_name1 = "PyCBC_T2_1.gwf"

channel_name1 = "H1:TEST-STRAIN"

start = 0
end = start + 128

ts1 = read_frame(file_name1, channel_name1, start, end)
# Remove the low frequency content and downsample the data to 2048Hz
strain1 = highpass(ts1, 15.0)
strain1 = resample_to_delta_t(strain1, 1.0/2048)

pylab.plot(strain1.sample_times, strain1)
pylab.xlabel('Time (s)')
pylab.show()
```



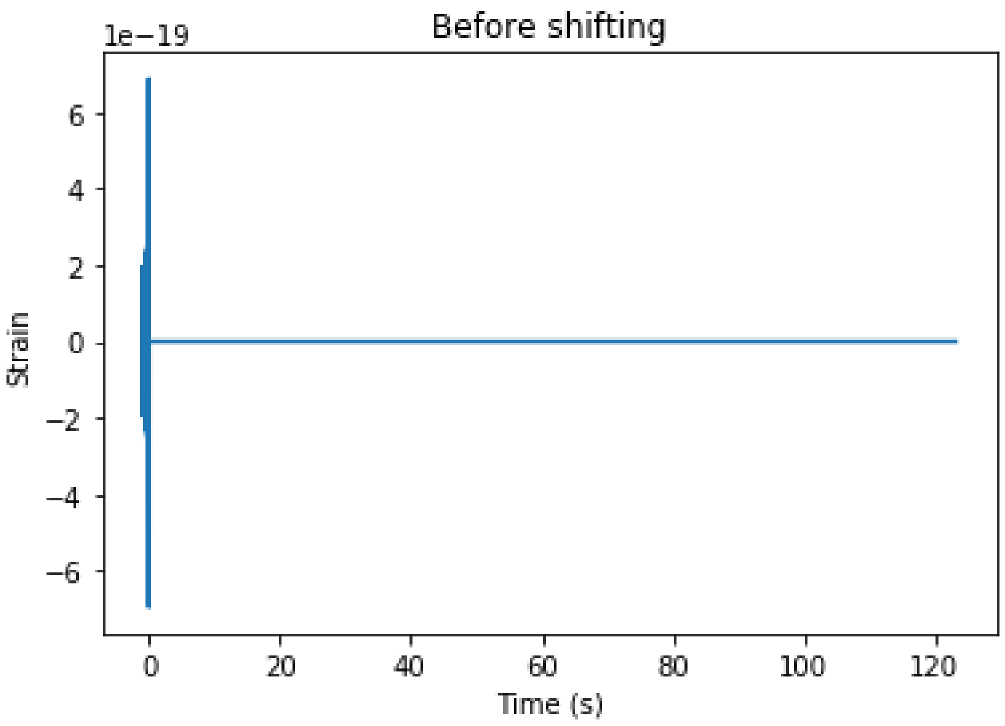
[21]:
```python
# Remove 2 seconds of data from both the beginning and end
conditioned1 = strain1.crop(2, 2)

pylab.plot(conditioned1.sample_times, conditioned1)
pylab.xlabel('Time (s)')
pylab.show()
```

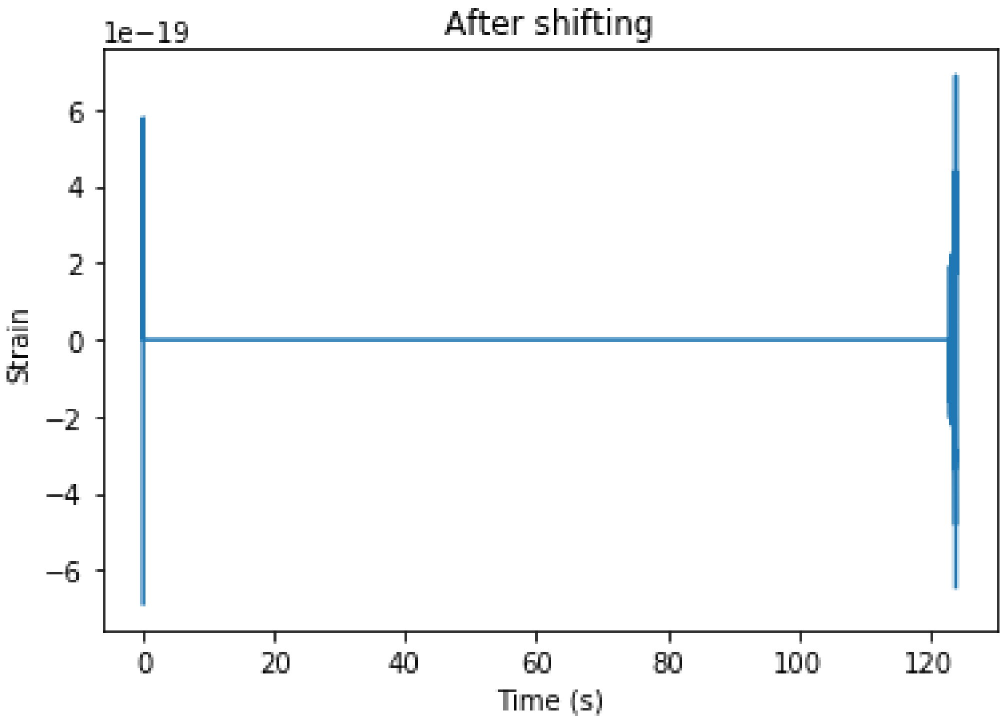


[22]:
```python
from pycbc.psd import interpolate, inverse_spectrum_truncation
# Estimate the power spectral density

# We use 4 second samples of our time series in Welch method.
psd1 = conditioned1.psd(4)

# Now that we have the psd we need to interpolate it to match our data
# and then limit the filter length of 1 / PSD. After this, we can
# directly use this PSD to filter the data in a controlled manner
psd1 = interpolate(psd1, conditioned1.delta_f)

# 1/PSD will now act as a filter with an effective length of 4 seconds
# Since the data has been highpassed above 15 Hz, and will have low values
# below this we need to inform the function to not include frequencies
# below this frequency.
psd1 = inverse_spectrum_truncation(psd1, 4 * conditioned1.sample_rate,
                                  low_frequency_cutoff=15)
```

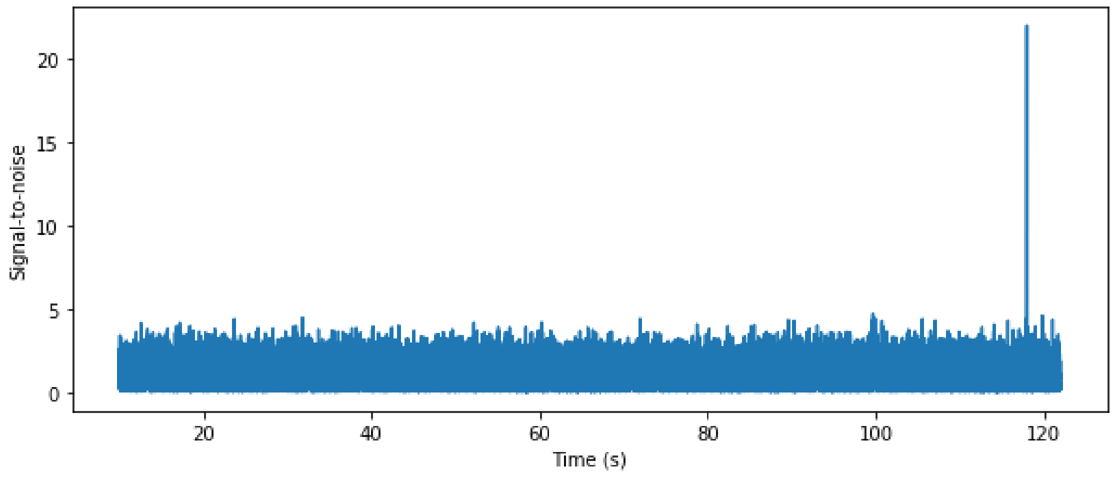
[23]:
```
from pycbc.waveform import get_td_waveform
# In this case we "know" what the signal parameters are. In a search
# we would grid over the parameters and calculate the SNR time series
# for each one

# We'll assume equal masses, and non-rotating black holes which is within the␣
 ↪posterior probability
# of GW150914.
m = 30 # Solar masses
hp, hc = get_td_waveform(approximant="SEOBNRv4_opt",
                     mass1=m,
                     mass2=m,
                     delta_t=conditioned1.delta_t,
                     f_lower=20)

# We will resize the vector to match our data
hp.resize(len(conditioned1))

# The waveform begins at the start of the vector, so if we want the
# SNR time series to correspond to the approximate merger location
# we need to shift the data so that the merger is approximately at the
# first bin of the data.

# The cyclic_time_shift method shifts the timeseries by a given amount of time.
# It treats the data as if it were on a ring so points shifted off the end
# of the series reappear at the start. Note that time stamps are *not* in
# general affected (as the start time of the full array is shifted),
# but the index of each point in the vector is.
#
# By convention waveforms returned from `get_td_waveform` have their
# merger stamped with time zero, so we can use the start time to
# shift the merger into position
pylab.figure()
pylab.title('Before shifting')
pylab.plot(hp.sample_times, hp)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')

template = hp.cyclic_time_shift(hp.start_time)

pylab.figure()
pylab.title('After shifting')
pylab.plot(template.sample_times, template)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')
```

[23]:
```
Text(0, 0.5, 'Strain')
```


Before shifting
1e−19
6
4
2
0
−2
−4
−6
Strain
0
20
40
60
80
100
120
Time (s)


After shifting
1e−19
6
4
2
0
−2
−4
−6
Strain
0
20
40
60
80
100
120
Time (s)

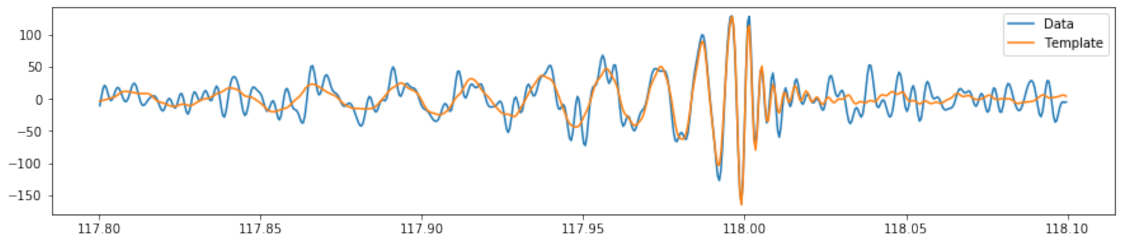
[24]:
```python
from pycbc.filter import matched_filter
import numpy

snr = matched_filter(template, conditioned1,
                     psd=psd1, low_frequency_cutoff=20)

# Remove time corrupted by the template filter and the psd filter
# We remove 4 seonds at the beginning and end for the PSD filtering
# And we remove 4 additional seconds at the beginning to account for
# the template length (this is somewhat generous for
# so short a template). A longer signal such as from a BNS, would
# require much more padding at the beginning of the vector.
snr = snr.crop(4 + 4, 4)

# Why are we taking an abs() here?
# The `matched_filter` function actually returns a 'complex' SNR.
# What that means is that the real portion correponds to the SNR
# associated with directly filtering the template with the data.
# The imaginary portion corresponds to filtering with a template that
# is 90 degrees out of phase. Since the phase of a signal may be
# anything, we choose to maximize over the phase of the signal.
pylab.figure(figsize=[10, 4])
pylab.plot(snr.sample_times, abs(snr))
pylab.ylabel('Signal-to-noise')
pylab.xlabel('Time (s)')
pylab.show()

peak = abs(snr).numpy().argmax()
snrp = snr[peak]
time = snr.sample_times[peak]

print("We found a signal at {}s with SNR {}".format(time,
                                                    abs(snrp)))
```



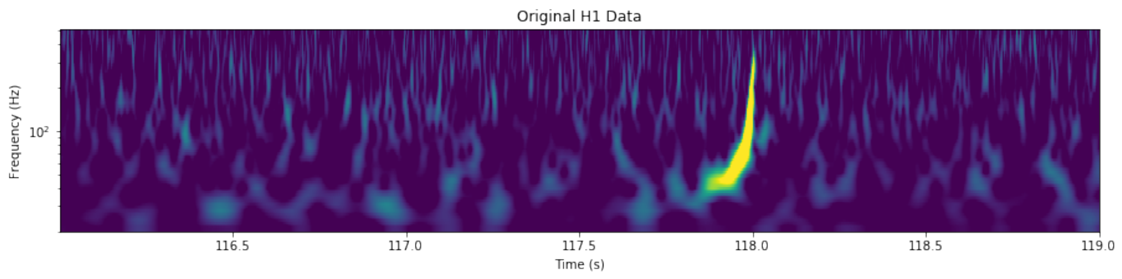
```
We found a signal at 118.00048828125s with SNR 21.98363577473979
```

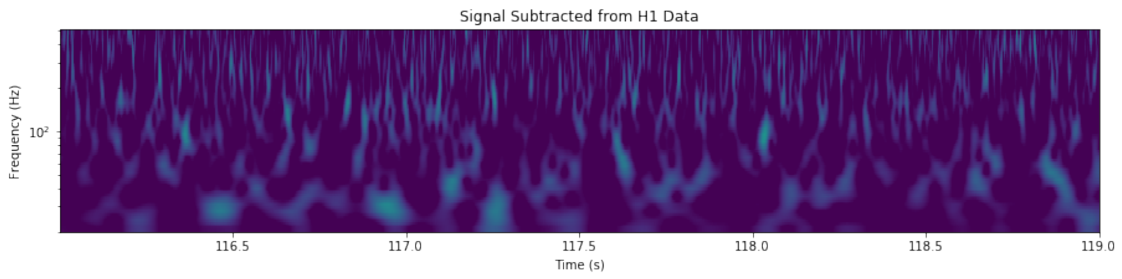
[25]:
```python
from pycbc.filter import sigma
# The time, amplitude, and phase of the SNR peak tell us how to align
# our proposed signal with the data.

# Shift the template to the peak time
dt = time - conditioned1.start_time
aligned = template.cyclic_time_shift(dt)

# scale the template so that it would have SNR 1 in this data
aligned /= sigma(aligned, psd=psd1, low_frequency_cutoff=20.0)

# Scale the template amplitude and phase to the peak value
aligned = (aligned.to_frequencyseries() * snrp).to_timeseries()
aligned.start_time = conditioned1.start_time
```

[26]:
```python
# We do it this way so that we can whiten both the template and the data
white_data = (conditioned1.to_frequencyseries() / psd1**0.5).to_timeseries()
white_template = (aligned.to_frequencyseries() / psd1**0.5).to_timeseries()

white_data = white_data.highpass_fir(30., 512).lowpass_fir(300, 512)
white_template = white_template.highpass_fir(30, 512).lowpass_fir(300, 512)

# Select the time around the merger
white_data = white_data.time_slice(time-.2, time+.1)
white_template = white_template.time_slice(time-.2, time+.1)

pylab.figure(figsize=[15, 3])
pylab.plot(white_data.sample_times, white_data, label="Data")
pylab.plot(white_template.sample_times, white_template, label="Template")
pylab.legend()
pylab.show()
```



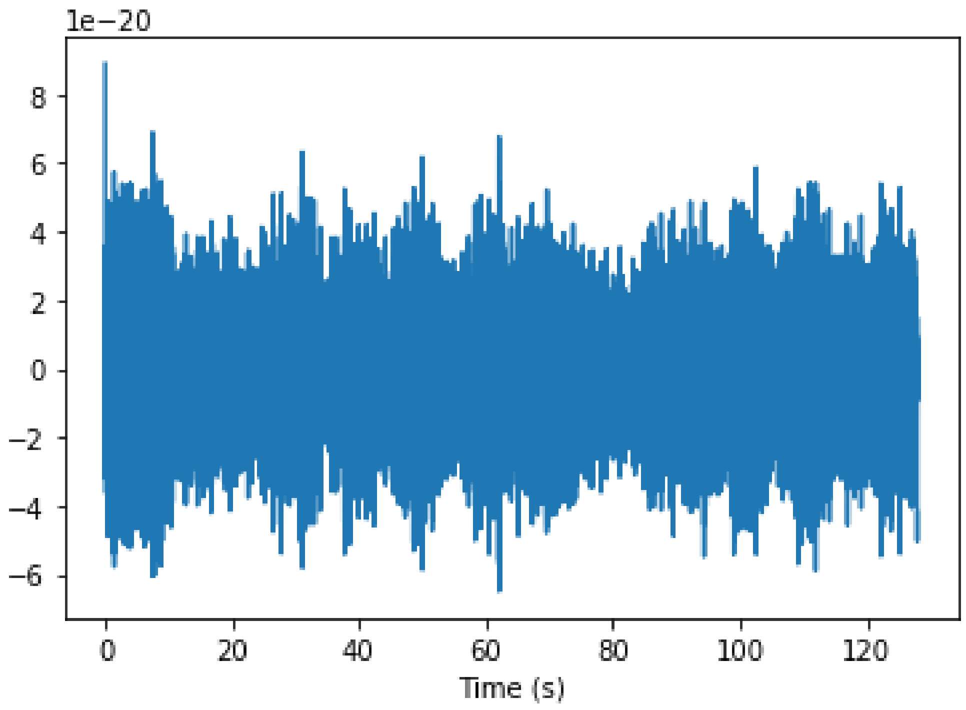
[27]:
```
subtracted = conditioned1 - aligned

# Plot the original data and the subtracted signal data

for data, title in [(conditioned1, 'Original H1 Data'),
                    (subtracted, 'Signal Subtracted from H1 Data')]:

    t, f, p = data.whiten(4, 4).qtransform(.001,
                                            logfsteps=100,
                                            qrange=(8, 8),
                                            frange=(20, 512))
    pylab.figure(figsize=[15, 3])
    pylab.title(title)
    pylab.pcolormesh(t, f, p**0.5, vmin=1, vmax=6)
    pylab.yscale('log')
    pylab.xlabel('Time (s)')
    pylab.ylabel('Frequency (Hz)')
    pylab.xlim(time - 2, time + 1)
    pylab.show()
```



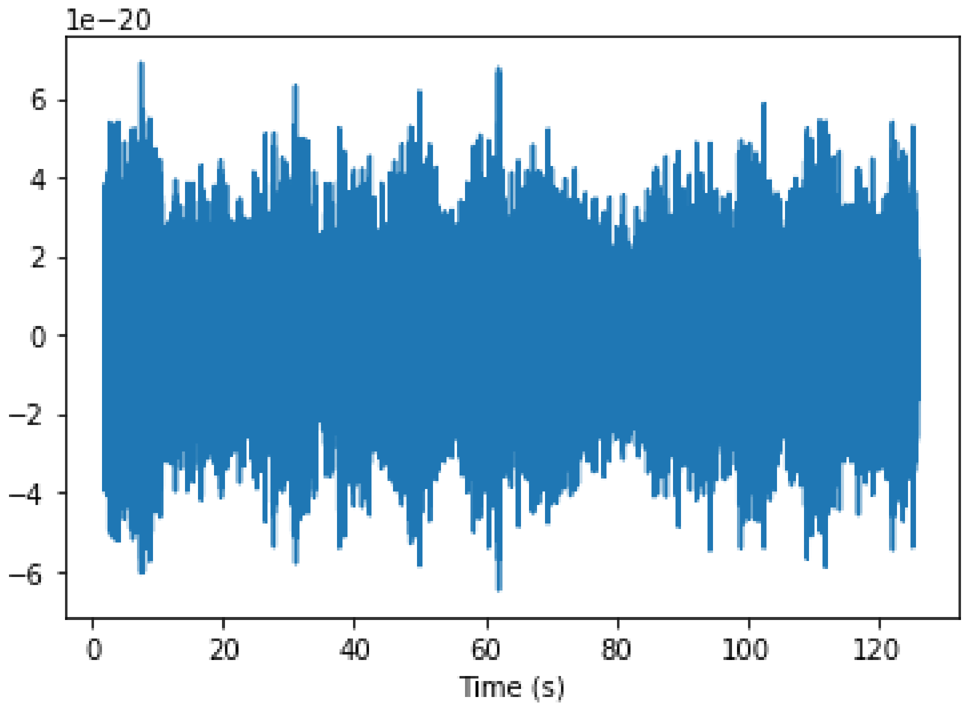


1.3.3 Analysis of PyCBC_T2_2.gwf

[28]:
```
%matplotlib inline

import pylab
from pycbc.catalog import Merger
from pycbc.filter import resample_to_delta_t, highpass

# As an example we use the GW150914 data
file_name1 = "PyCBC_T2_2.gwf"

channel_name1 = "H1:TEST-STRAIN"

start = 0
end = start + 128

ts1 = read_frame(file_name1, channel_name1, start, end)
# Remove the low frequency content and downsample the data to 2048Hz
strain1 = highpass(ts1, 15.0)
strain1 = resample_to_delta_t(strain1, 1.0/2048)

pylab.plot(strain1.sample_times, strain1)
pylab.xlabel('Time (s)')
pylab.show()
```



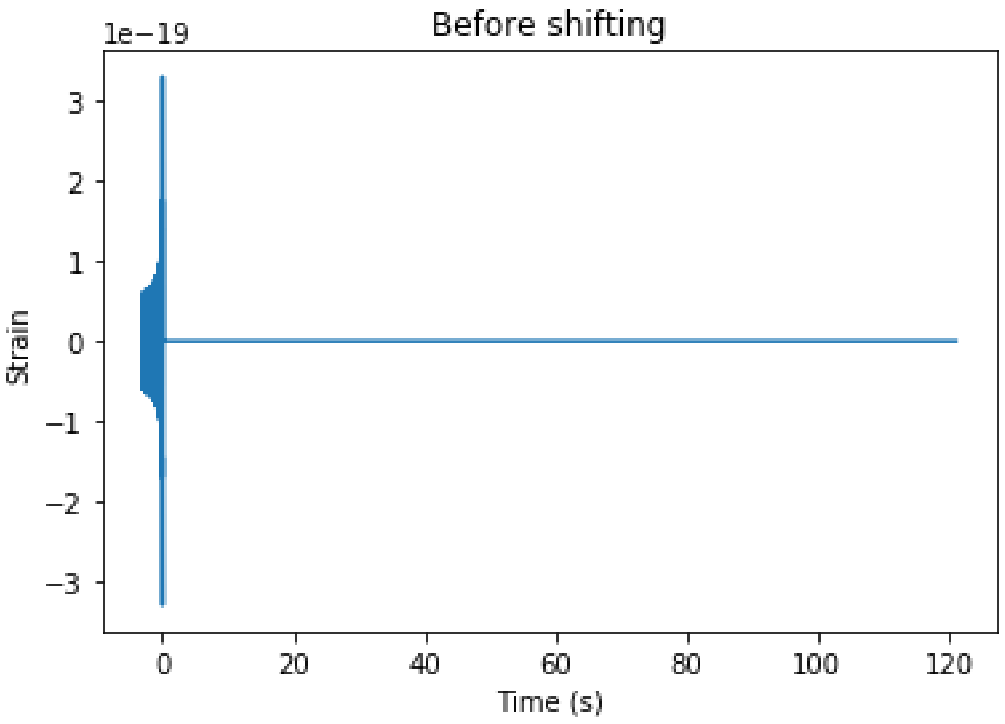
[29]:
```
# Remove 2 seconds of data from both the beginning and end
conditioned1 = strain1.crop(2, 2)

pylab.plot(conditioned1.sample_times, conditioned1)
pylab.xlabel('Time (s)')
pylab.show()
```

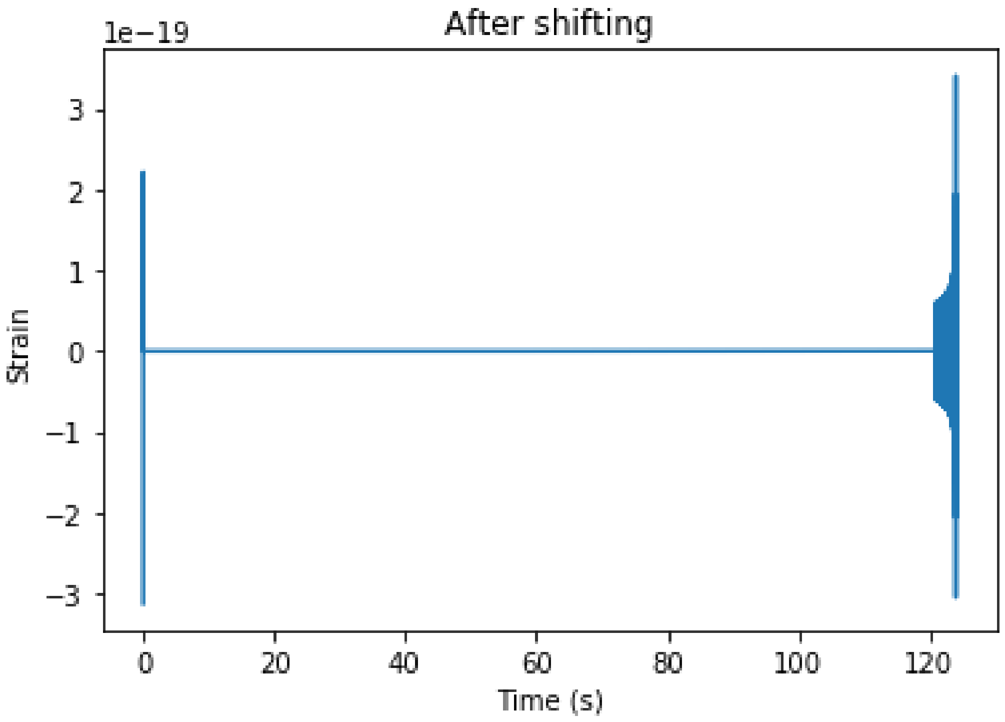


[30]:
```
from pycbc.psd import interpolate, inverse_spectrum_truncation
# Estimate the power spectral density

# We use 4 second samples of our time series in Welch method.
psd1 = conditioned1.psd(4)

# Now that we have the psd we need to interpolate it to match our data
# and then limit the filter length of 1 / PSD. After this, we can
# directly use this PSD to filter the data in a controlled manner
psd1 = interpolate(psd1, conditioned1.delta_f)

# 1/PSD will now act as a filter with an effective length of 4 seconds
# Since the data has been highpassed above 15 Hz, and will have low values
# below this we need to inform the function to not include frequencies
# below this frequency.
psd1 = inverse_spectrum_truncation(psd1, 4 * conditioned1.sample_rate,
                                   low_frequency_cutoff=15)
```

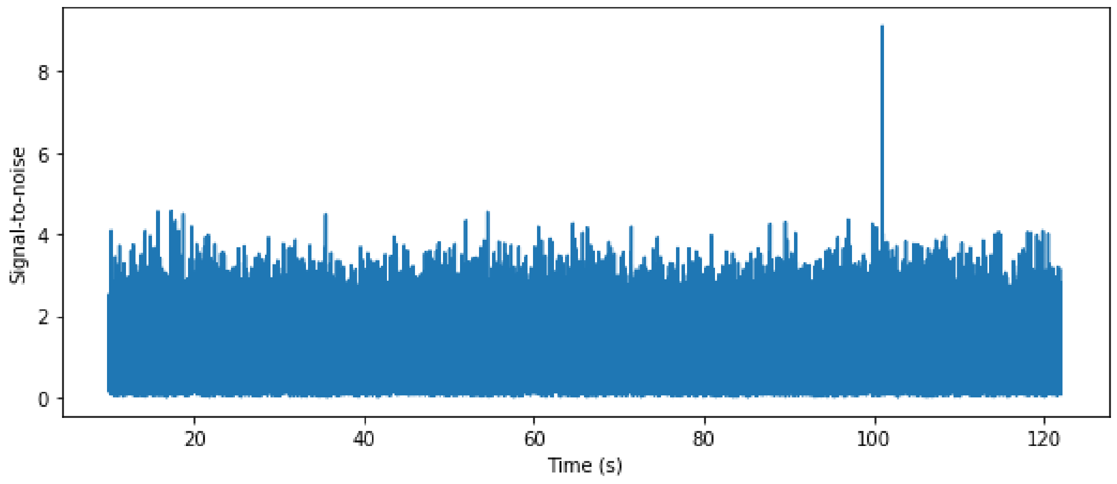
[31]:
```
from pycbc.waveform import get_td_waveform
# In this case we "know" what the signal parameters are. In a search
# we would grid over the parameters and calculate the SNR time series
# for each one

# We'll assume equal masses, and non-rotating black holes which is within the␣
 ↪posterior probability
# of GW150914.
m = 15 # Solar masses
hp, hc = get_td_waveform(approximant="SEOBNRv4_opt",
                     mass1=m,
                     mass2=m,
                     delta_t=conditioned1.delta_t,
                     f_lower=20)

# We will resize the vector to match our data
hp.resize(len(conditioned1))

# The waveform begins at the start of the vector, so if we want the
# SNR time series to correspond to the approximate merger location
# we need to shift the data so that the merger is approximately at the
# first bin of the data.

# The cyclic_time_shift method shifts the timeseries by a given amount of time.
# It treats the data as if it were on a ring so points shifted off the end
# of the series reappear at the start. Note that time stamps are *not* in
# general affected (as the start time of the full array is shifted),
# but the index of each point in the vector is.
#
# By convention waveforms returned from `get_td_waveform` have their
# merger stamped with time zero, so we can use the start time to
# shift the merger into position
pylab.figure()
pylab.title('Before shifting')
pylab.plot(hp.sample_times, hp)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')

template = hp.cyclic_time_shift(hp.start_time)

pylab.figure()
pylab.title('After shifting')
pylab.plot(template.sample_times, template)
pylab.xlabel('Time (s)')
pylab.ylabel('Strain')
```

[31]:
```
Text(0, 0.5, 'Strain')
```


Before shifting
1e−19
Strain
Time (s)
3
2
1
0
−1
−2
−3
0
20
40
60
80
100
120


After shifting
1e−19
Strain
Time (s)
3
2
1
0
−1
−2
−3
0
20
40
60
80
100
120

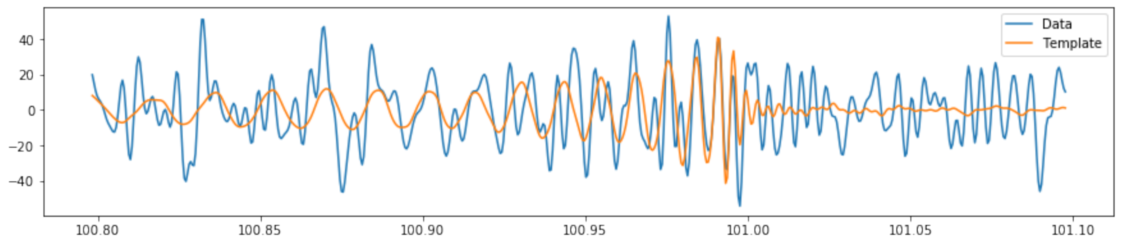
[32]:
```
from pycbc.filter import matched_filter
import numpy

snr = matched_filter(template, conditioned1,
                     psd=psd1, low_frequency_cutoff=20)

# Remove time corrupted by the template filter and the psd filter
# We remove 4 seonds at the beginning and end for the PSD filtering
# And we remove 4 additional seconds at the beginning to account for
# the template length (this is somewhat generous for
# so short a template). A longer signal such as from a BNS, would
# require much more padding at the beginning of the vector.
snr = snr.crop(4 + 4, 4)

# Why are we taking an abs() here?
# The `matched_filter` function actually returns a 'complex' SNR.
# What that means is that the real portion correponds to the SNR
# associated with directly filtering the template with the data.
# The imaginary portion corresponds to filtering with a template that
# is 90 degrees out of phase. Since the phase of a signal may be
# anything, we choose to maximize over the phase of the signal.
pylab.figure(figsize=[10, 4])
pylab.plot(snr.sample_times, abs(snr))
pylab.ylabel('Signal-to-noise')
pylab.xlabel('Time (s)')
pylab.show()

peak = abs(snr).numpy().argmax()
snrp = snr[peak]
time = snr.sample_times[peak]

print("We found a signal at {}s with SNR {}".format(time,
                                                    abs(snrp)))
```



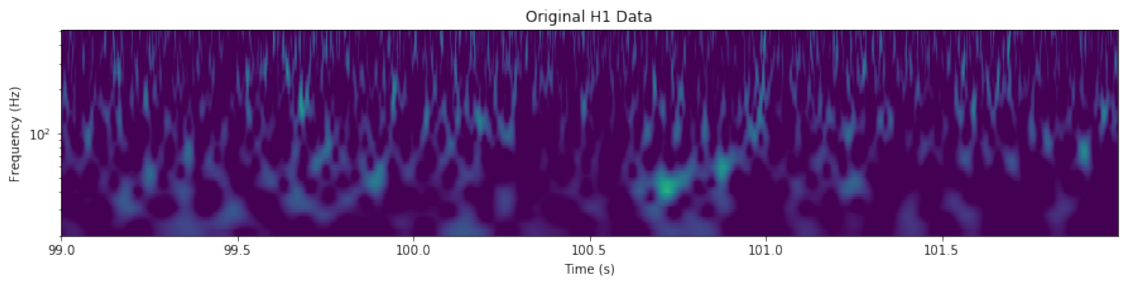
```
We found a signal at 100.99853515625s with SNR 9.108136792827866
```

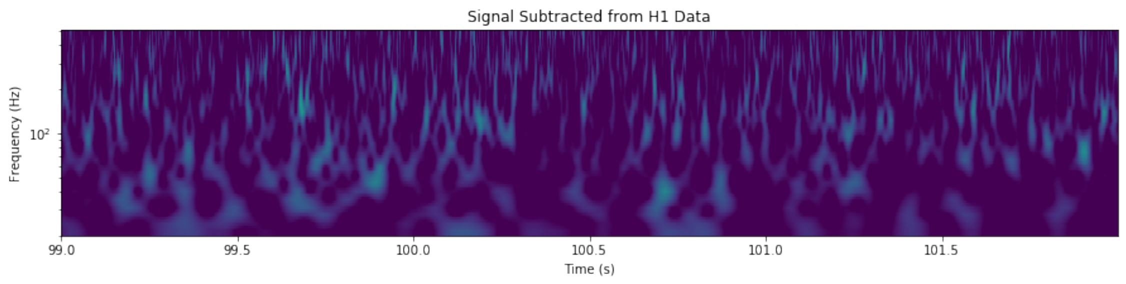
[33]:
```python
from pycbc.filter import sigma
# The time, amplitude, and phase of the SNR peak tell us how to align
# our proposed signal with the data.

# Shift the template to the peak time
dt = time - conditioned1.start_time
aligned = template.cyclic_time_shift(dt)

# scale the template so that it would have SNR 1 in this data
aligned /= sigma(aligned, psd=psd1, low_frequency_cutoff=20.0)

# Scale the template amplitude and phase to the peak value
aligned = (aligned.to_frequencyseries() * snrp).to_timeseries()
aligned.start_time = conditioned1.start_time
```

[34]:
```python
# We do it this way so that we can whiten both the template and the data
white_data = (conditioned1.to_frequencyseries() / psd1**0.5).to_timeseries()
white_template = (aligned.to_frequencyseries() / psd1**0.5).to_timeseries()

white_data = white_data.highpass_fir(30., 512).lowpass_fir(300, 512)
white_template = white_template.highpass_fir(30, 512).lowpass_fir(300, 512)

# Select the time around the merger
white_data = white_data.time_slice(time-.2, time+.1)
white_template = white_template.time_slice(time-.2, time+.1)

pylab.figure(figsize=[15, 3])
pylab.plot(white_data.sample_times, white_data, label="Data")
pylab.plot(white_template.sample_times, white_template, label="Template")
pylab.legend()
pylab.show()
```



[35]:
```
subtracted = conditioned1 - aligned

# Plot the original data and the subtracted signal data

for data, title in [(conditioned1, 'Original H1 Data'),
                    (subtracted, 'Signal Subtracted from H1 Data')]:

    t, f, p = data.whiten(4, 4).qtransform(.001,
                                          logfsteps=100,
                                          qrange=(8, 8),
                                          frange=(20, 512))
    pylab.figure(figsize=[15, 3])
    pylab.title(title)
    pylab.pcolormesh(t, f, p**0.5, vmin=1, vmax=6)
    pylab.yscale('log')
    pylab.xlabel('Time (s)')
    pylab.ylabel('Frequency (Hz)')
    pylab.xlim(time - 2, time + 1)
    pylab.show()
```


Original H1 Data
Frequency (Hz)
10^2
Time (s)


Signal Subtracted from H1 Data
Frequency (Hz)
10^2
Time (s)