(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/projects/Project2_288.ipynb)

Project 2

Fourier methods, Matched Filtering, and Differential Equations

This notebook is arranged in cells. Texts are usually written in the markdown cells, and here you can use html tags (make it bold, italic, colored, etc). You can double click on this cell to see the formatting.

The ellipsis (...) are provided where you are expected to write your solution but feel free to change the template (not over much) in case this style is not to your taste.

Hit "Shift-Enter" on a code cell to evaluate it. Double click a Markdown cell to edit.

Imports

In [1]:
```
!pip3 install --upgrade pip && pip3 install h5py=='2.9.0'
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/
Requirement already satisfied: pip in /usr/local/lib/python3.8/dist-pa
Collecting pip
  Downloading pip-22.3.1-py3-none-any.whl (2.1 MB)
     |████████████████████████████████| 2.1 MB 5.1 MB/s
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 21.1.3
    Uninstalling pip-21.1.3:
      Successfully uninstalled pip-21.1.3
Successfully installed pip-22.3.1
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/
Collecting h5py==2.9.0
  Downloading h5py-2.9.0-cp38-cp38-manylinux1_x86_64.whl (2.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.8/2.8 MB 14.6 MB/s eta
Requirement already satisfied: numpy>=1.7 in /usr/local/lib/python3.8/d
Requirement already satisfied: six in /usr/local/lib/python3.8/dist-pa
Installing collected packages: h5py
  Attempting uninstall: h5py
    Found existing installation: h5py 3.1.0
    Uninstalling h5py-3.1.0:
      Successfully uninstalled h5py-3.1.0
Successfully installed h5py-2.9.0
WARNING: Running pip as the 'root' user can result in broken permissio
```

In [2]:
```
import numpy as np
import json
from scipy import signal
from scipy.interpolate import interp1d
from scipy.signal import butter, filtfilt, iirdesign, zpk2tf, freqz
from IPython.display import Audio
from scipy.io import wavfile
import h5py
import matplotlib.mlab as mlab
#For plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Mounting Google Drive locally

Mount your Google Drive on your runtime using an authorization code.

Note: When using the 'Mount Drive' button in the file browser, no authentication codes are necessary for notebooks that have only been edited by the current user.

In [3]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

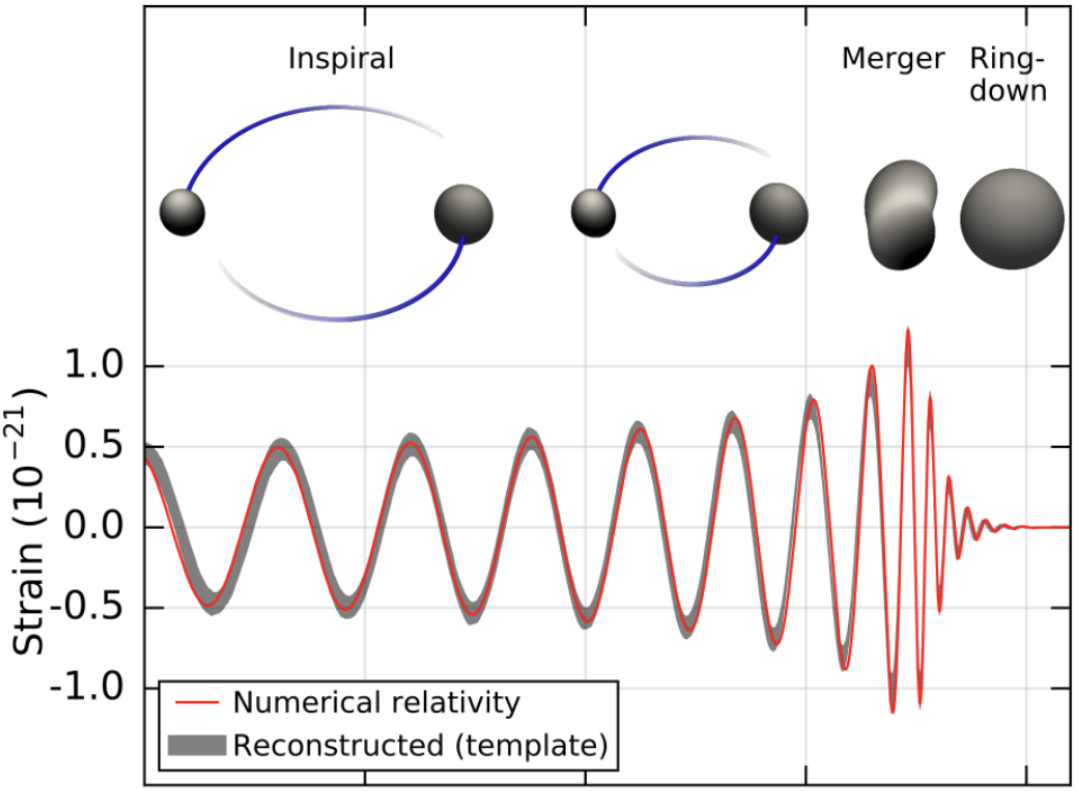
Problem - LIGO Gravitational Wave Detection

On February 11 2016, the LIGO (Laser Interferometer Gravitational-wave Observatory) and Virgo collaboration announced the first observation of gravitational waves from the collision and merger of a pair of black holes. Such observation was made on 14 September 2015 by the two

detectors of the LIGO (Hanford and Livington); hence, the signal was named "GW150914." This event took place in a distant galaxy more than one billion light years from the Earth. LIGO estimated that the peak gravitational-wave power radiated during the final moments of the black hole merger was more than ten times greater than the combined light power from all the stars and galaxies in the observable Universe. This remarkable discovery marks the beginning of an exciting new era of astronomy as we open an entirely new, gravitational-wave, window on the Universe. The 2017 Nobel Prize was awarded to researchers of the LIGO/Virgo collaboration for such discovery (https://www.nobelprize.org/nobel_prizes/physics/laureates/2017/press.html (https://www.nobelprize.org/nobel_prizes/physics/laureates/2017/press.html)).

Gravitational waves are "ripples" in space-time produced by some of the most violent events in the cosmos, such as the collisions and mergers of massive compact stars. Their existence was predicted by Einstein in 1916, when he showed that accelerating massive objects would shake space-time so much that waves of distorted space would radiate from the source.

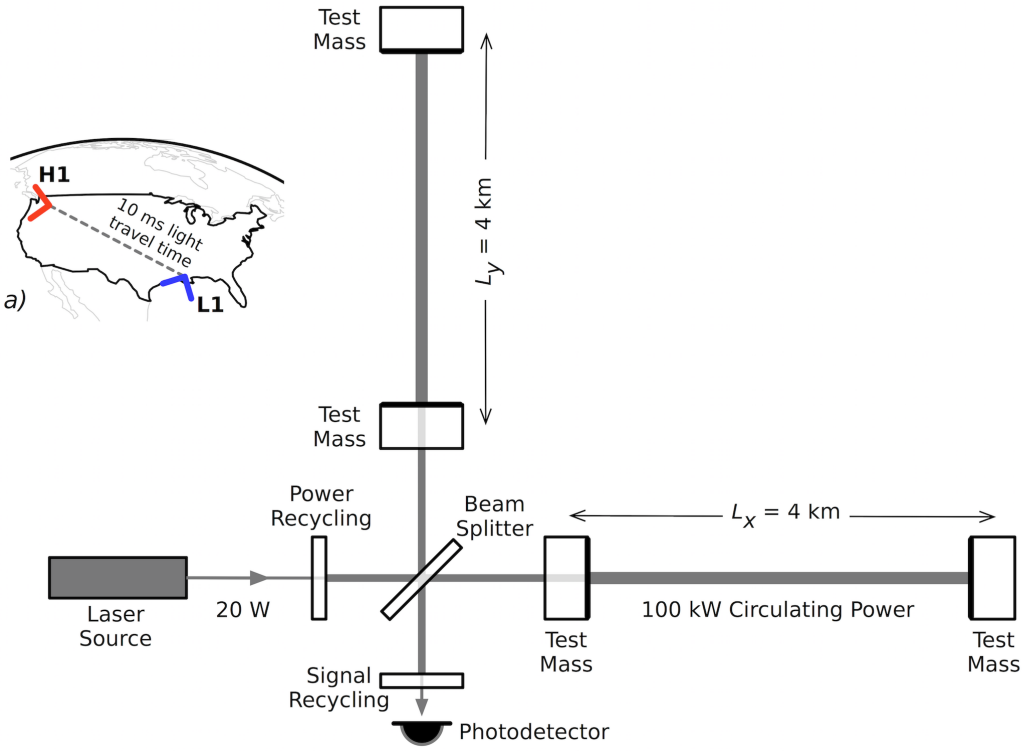
In the same year that Einstein predicted gravitational waves, the physicist Karl Schwarzschild showed that Einstein's work permitted the existence of black holes. There have been dramatic improvements in our theoretical understanding of these bizarre objects - including, over the past decade, some remarkable advances in modeling a pair of black holes (referred to as a binary) through several close orbits before they finally merge. These computer models have allowed us to construct precise gravitational waveforms - i.e. the pattern of gravitational waves emitted by the black holes as they approach ever closer and finally merge into a single, larger black hole - in accordance with the predictions of general relativity. The direct observation of a binary black hole merger would therefore provide a powerful cosmic laboratory for testing Einstein's theory.



The above image shows the predictions of the best-matching waveform computed from general relativity, over the three stages of the event: inspiral, merger and ringdown. A visual simulation is available here: http://www.glowscript.org/#/user/rhilborn/folder/Public/program/BinaryInSpiral (http://www.glowscript.org/#/user/rhilborn/folder/Public/program/BinaryInSpiral)

LIGO comprised of two giant laser interferometers located thousands of kilometers apart, one in Livingston, Louisiana and the other in Hanford, Washington. As shown in the below simplified diagram, an interferometer like LIGO consists of two "arms" (each one 4km long) at right angles

to each other, along which a laser beam is shone and reflected by mirrors (suspended as test masses) at each end. When a gravitational wave passes by, the stretching and squashing of space causes the arms of the interferometer alternately to lengthen and shrink, one getting longer while the other gets shorter and then vice-versa. As the interferometers' arms change lengths, the laser beams take a different time to travel through the arms - which means that the two beams are no longer "in step" (or in phase) and what we call an interference pattern is produced. This is why we refer to the LIGO detectors as "interferometers".



The difference between the two arm lengths is proportional to the strength of the passing gravitational wave, referred to as the **gravitational-wave strain**, and this number is mind-bogglingly small. For a gravitational wave typical of what we can detect, we expect the strain to be about 1/10,000th the width of a proton! However LIGO's interferometers are so sensitive that they can measure even such tiny amounts.

In Project 2, we take the strain data from LIGO and use matched filtering to find hidden signals from noisy data. Ultimately, we aim to show that the waveform detected by LIGO matched the predictions of general relativity, confirming that LIGO has detected gravitational waves.

First, load the data from LIGO Hanford's **H1** detector and Livingston's **L1** detector.

Reference: http://www.ligo.org/science/Publication-GW150914/ (http://www.ligo.org/science/Publication-GW150914/)

In [4]:
```
# Load data
events = json.load(open("/content/drive/My Drive/P188_288/P188_288_Pro
event = events['GW150914']
```

In [5]:
```
# Module for reading LIGO data files
import sys
sys.path.append('/content/drive/My Drive/P188_288/P188_288_Project2/')
import readligo as rl
```

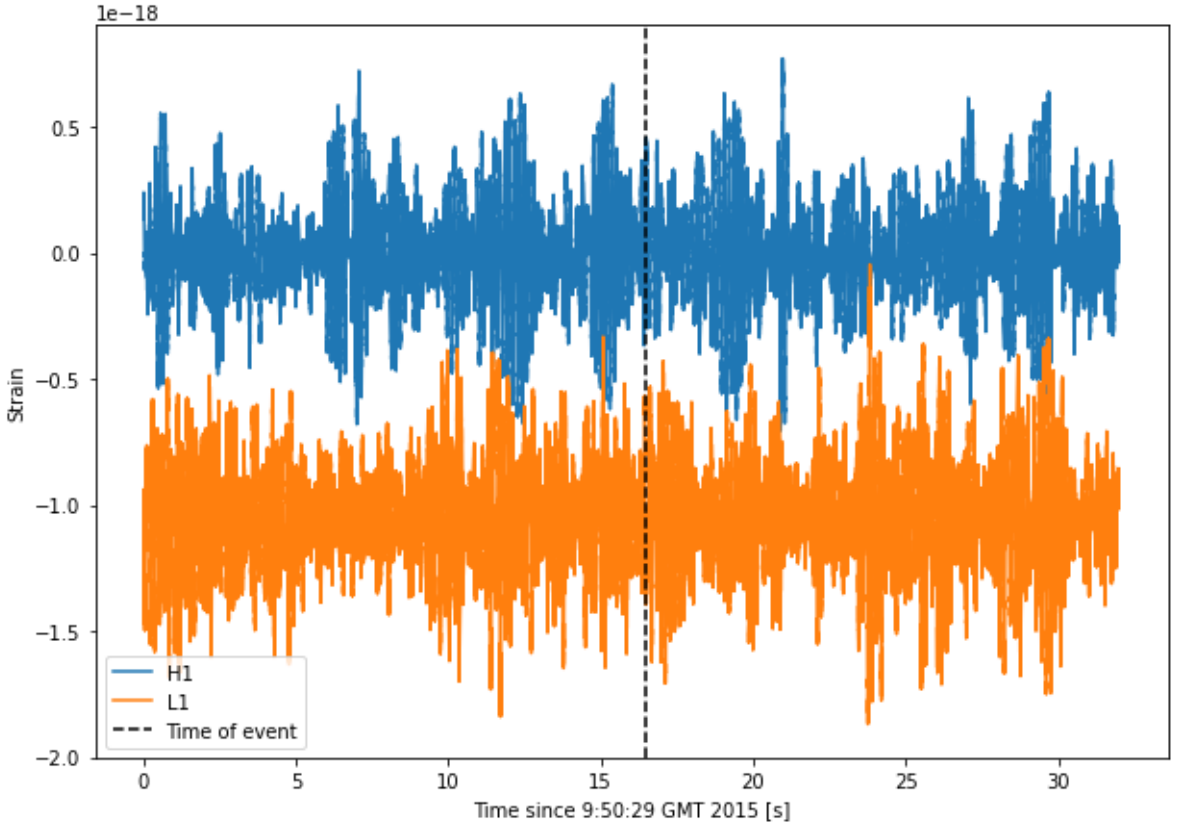
In [6]:
```
# H1 data
# "strain_H1" is a vector of strain values
# "time_H1" is a vector of time values to match the strain vector
# chan_dic_H1 is a dictionary of data quality channels. Ignore this.
fn_H1 = '/content/drive/My Drive/P188_288/P188_288_Project2/H-H1_LOSC_
strain_H1, time, chan_dict_H1 = rl.loaddata(fn_H1, 'H1')

# L1 data
fn_L1 = '/content/drive/My Drive/P188_288/P188_288_Project2/L-L1_LOSC_
strain_L1, time, chan_dict_L1 = rl.loaddata(fn_L1, 'L1')

# Approximate time of a binary black hole merger (Mon Sep 14 09:50:45 
tevent = event['tevent'] - time[0]

# Time since 9:50:29 GMT 2015
time = time - time[0]
tevent = tevent - time[0]

# Template waveform
fn_template = '/content/drive/My Drive/P188_288/P188_288_Project2/GW15

# Sampling rate
fs = event['fs']

# frequency band for bandpassing signal
fband = event['fband']
```

```
/usr/local/lib/python3.8/dist-packages/h5py/_hl/dataset.py:312: H5pyDe
  warn("dataset.value has been deprecated. "
```

We define the vector "time" as the time since Sep 14 9:50:29 GMT 2015. The event of a binary black hole merger occurred around Sep 14 09:50:45 GMT 2015 ("tevent" - about 16s after time[0]). The strain values of H1 and L1 detectors are stored as "strain_H1" and "strain_L1" respectively.

1. Plot the time strain data for both H1 and L1. Mark the event of a merger on the plot.

In [7]:
```
plt.figure(figsize=(10, 7))
plt.plot(time, strain_H1, label="H1")
plt.plot(time, strain_L1, label="L1")
plt.axvline(tevent, ls="--", c="k", label="Time of event")
plt.legend()
plt.xlabel("Time since 9:50:29 GMT 2015 [s]")
plt.ylabel("Strain")
plt.show()
```



The FFT $y[k]$ of length N of the length-N sequence $x[n]$ is defined as

$$y[k] = \sum_{n=0}^{N-1} e^{-2\pi j \frac{kn}{N}} x[n] \,,$$

and the inverse transform is defined as follows

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi j \frac{kn}{N}} y[k] \,.$$

These transforms can be calculated by means of np.fft.fft (np.fft.rfft) and np.fft.ifft. (https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.fft.html (https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.fft.html))

From the definition of the FFT it can be seen that

$$y[0] = \sum_{n=0}^{N-1} x[n]\,.$$

For N even, the elements $y[1]\ldots y[N/2-1]$ contain the positive-frequency terms, and the elements $y[N/2]\ldots y[N-1]$ contain the negative-frequency terms, in order of decreasingly negative frequency. For N odd, the elements $y[1]\ldots y[(N-1)/2]$ contain the positive-frequency terms, and the elements $y[(N+1)/2]\ldots y[N-1]$ contain the negative-frequency terms, in order of decreasingly negative frequency.

In case the sequence x is real-valued, the values of $y[n]$ for positive frequencies is the conjugate of the values y[n] for negative frequencies (because the spectrum is symmetric). Typically, only the FFT corresponding to positive frequencies is plotted.

np.fft.fft gives $y[0], y[1]\ldots y[N-1]$ while np.fft.rfft gives $y[0], y[1]\ldots y[N/2-1]$ (real FFT).

Now, let us plot the FFT of the sum of two sines as an example.

Using $N = 600$ samples with the sample rate $f_s = 800$, assume that t runs from 0 to N/f_s.

We define the sum of two sines as:

$$f(t) = a_1 \sin(2\pi w_1 t) + a_2 \sin(2\pi w_2 t)$$

where $a_1 = 0.5, a_2 = 1.6, w_1 = 50, w_2 = 130$.

1. Using np.fft.rfft:
Now, compute the FFT of $f(t)$ by doing np.fft.rfft(f(t)). Then, take the absolute value of it and then multiply by a normalization factor $2/N$.

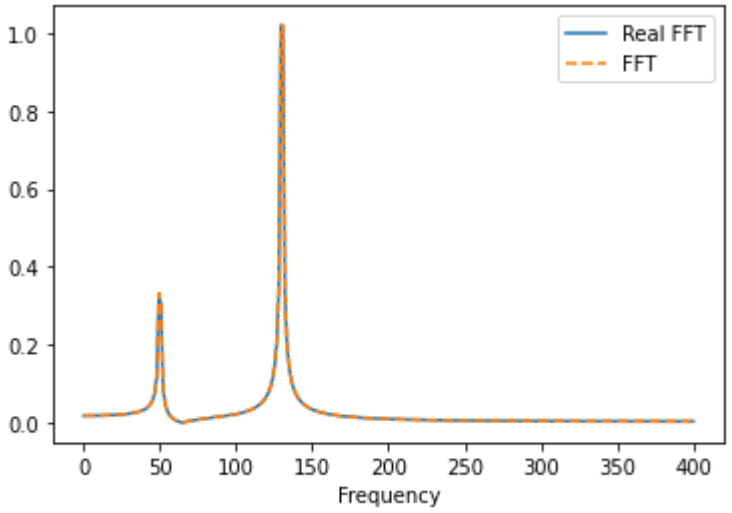
You can get the frequency using np.fft.rfftfreq. ("np.fft.rfftfreq(len(y), d = 1./fs_example)")

2. Using np.fft.fft:
Similarly, compute the FFT of $f(t)$ by doing np.fft.fft(f(t)) and take the absolute value of it and then multiply by a normalization factor $2/N$. You can also get the frequency using np.fft.fftfreq. ("np.fft.fftfreq(len(y), d = 1./fs_example)")

Now, select the first $N/2$ values in order to only plot the positive-frequency terms.

2. Plot the FFT of $f(t)$ using both np.fft.rfft and np.fft.fft.

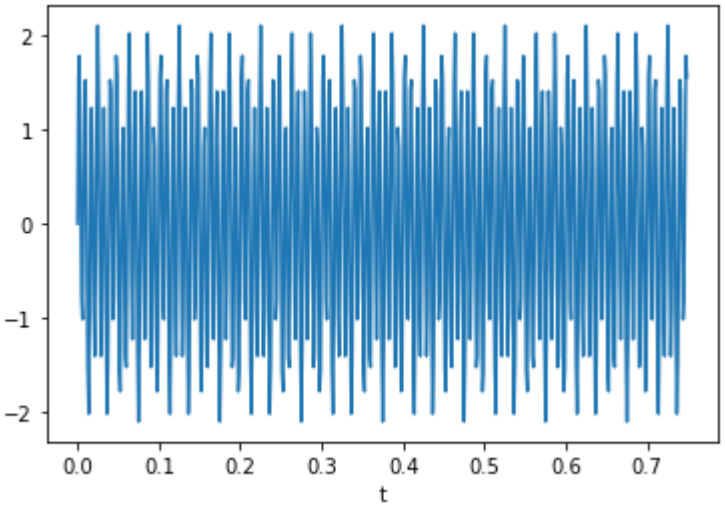
In [8]:
```
# Define f(t)
def f(t):
  a1 = .5
  a2 = 1.6
  w1 = 50
  w2 = 130
  return a1 * np.sin(2 * np.pi * w1 * t) + a2 * np.sin(2 * np.pi * w2

N = 600
fs = 800
t = np.arange(N) / fs

# Use rfft
yf = np.fft.rfft(f(t))
y_rfft = np.abs(yf) * 2 / N
rfreq = np.fft.rfftfreq(N, d=1./fs)

# Use fft
y_fft = np.abs(np.fft.fft(f(t))) * 2 / N
freq = np.fft.fftfreq(N, d=1./fs)

plt.figure()
plt.plot(rfreq, y_rfft, label="Real FFT")
plt.plot(freq[:int(N/2)], y_fft[:int(N/2)], ls="--", label="FFT")
plt.legend()
plt.xlabel("Frequency")
plt.show()
```



Now, transform back to the time domain. Let yf = np.fft.rfft(f(t)). Then, you can find the inverse FFT by doing np.fft.irfft(yf, len(f(t))).

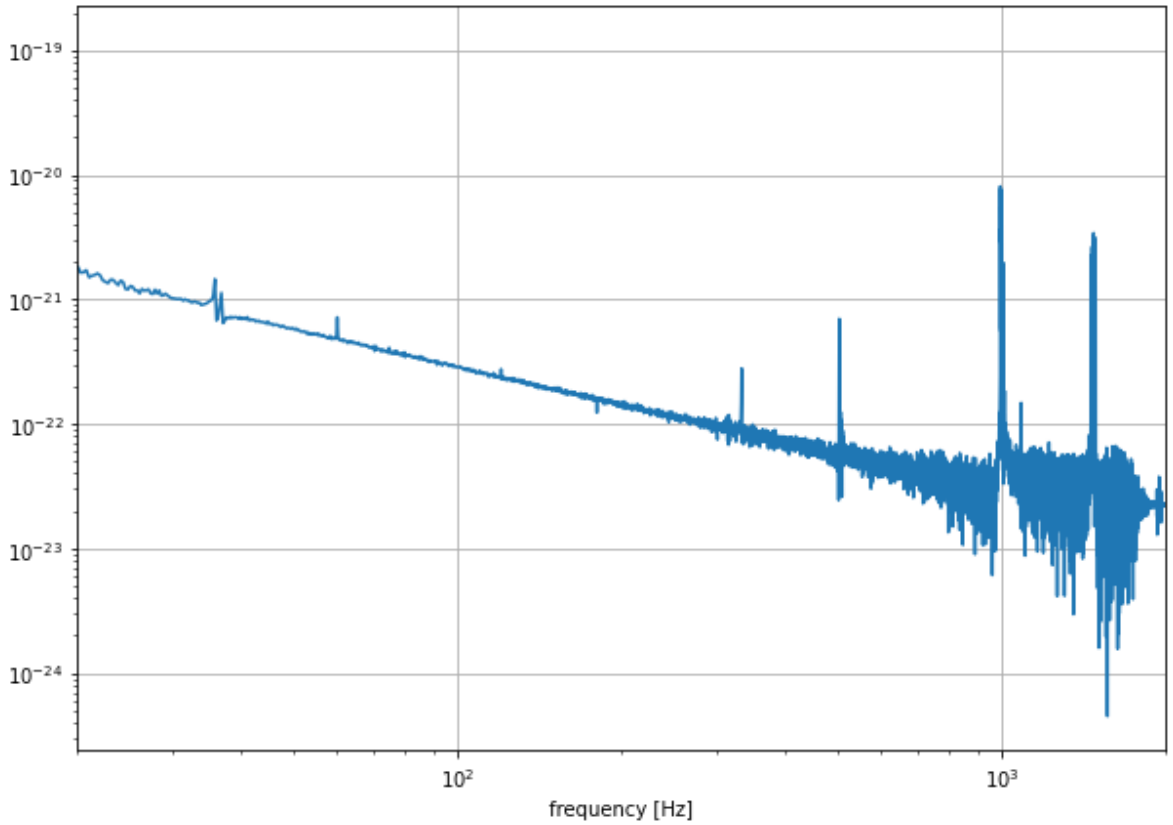
Plot the inverse FFT of yf (equivalent to ploting $f(t)$ vs. t).

In [9]:
```
plt.figure()
plt.plot(t, np.fft.irfft(yf, N))
plt.xlabel("t")
plt.show()
```



3. For LIGO strain data, $f_s = 4096$ Hz. Let $N = 4f_s$. Do the FFT of the H1 strain data using "np.fft.rfft" (compute the FFT of the strain data by doing np.fft.rfft(data). Then, take the absolute value of it and then multiply by a normalization factor $2/N$. You can get the frequency using np.fft.rfftfreq.) and plot it as a function of frequency in log-scale. We are not doing any binning, so the spectrum is expected to be quite noisy.

In [10]:
```
fs = event['fs']
N = 4*fs
dt = 1.0 / fs

h1_fft = np.abs(np.fft.rfft(strain_H1, n=N)) * 2 / N
freq = np.fft.rfftfreq(N, d=dt)

plt.figure(figsize = (10,7))

plt.loglog(freq, h1_fft)

plt.xlim(20, 2000)
plt.xlabel('frequency [Hz]')
plt.grid()
plt.show()
```



Plotting strain data in the Fourier domain gives us an idea of the frequency content of the data. Here, we visualize the frequency content of the data by plotting the amplitude spectral density, ASD.

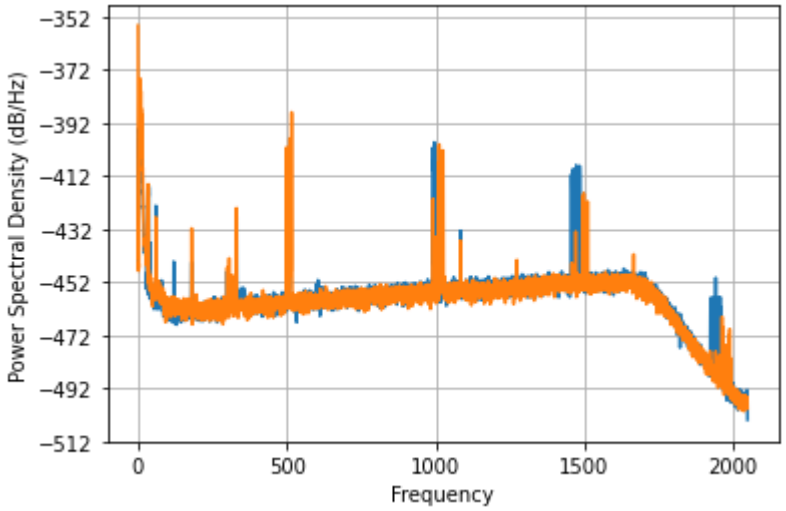
The ASDs are the square root of the power spectral densities (PSDs), which are averages of the square of the fast fourier transforms (FFTs) of the data.

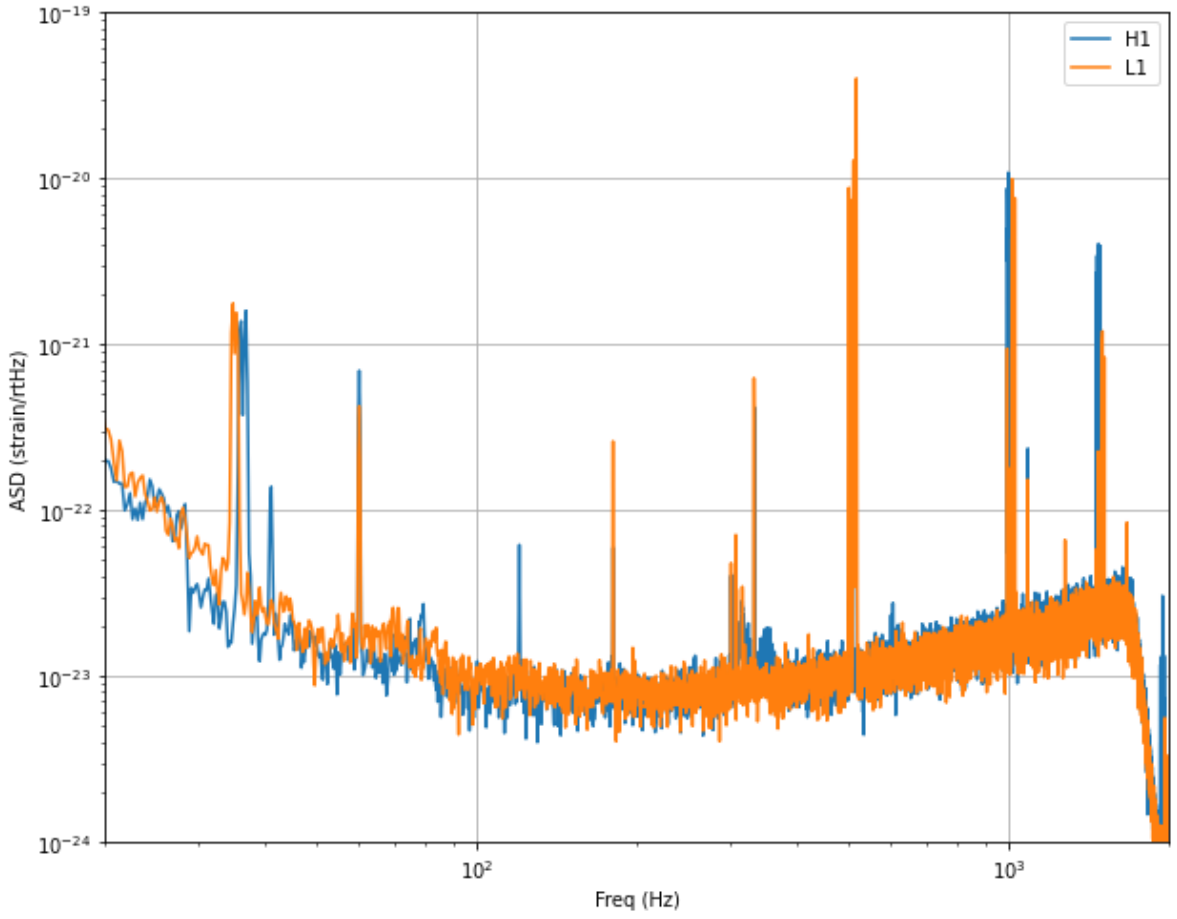
They are an estimate of the "strain-equivalent noise" of the detectors versus frequency, which limit the ability of the detectors to identify GW signals. (There's a signal in these data! For the moment, let's ignore that, and assume it's all noise.)

*4. mlab.psd (https://matplotlib.org/devdocs/api/_as_gen/matplotlib.pyplot.psd.html) gives PSDs. Use this to compute PSDs and take square roots to obtain ASDs. The input data are strain values, and let Fs = fs (which is 4096) and NFFT = 4*fs. (We are dividing the data into NFFT length segments) Plot ASDs of both H1 and L1 data. Label each plot.*

(Hint: Do "PSD, freqs = mlab.psd(data, Fs = fs, NFFT = 4∗fs)" to get the PSD values
Then, plot np.sqrt(PSD) as a function of freqs.)

In [11]:
```
fs = event['fs']
N = 4*fs
dt = 1.0 / fs

PSD_H1, freqs = plt.psd(strain_H1, Fs=fs, NFFT=N)
PSD_L1, freqs = plt.psd(strain_L1, Fs=fs, NFFT=N)

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

plt.loglog(freqs, np.sqrt(PSD_H1), label="H1")
plt.loglog(freqs, np.sqrt(PSD_L1), label="L1")

plt.axis([20, 2000, 1e-24, 1e-19])
plt.grid('on')
plt.ylabel('ASD (strain/rtHz)')
plt.xlabel('Freq (Hz)')
plt.legend()
plt.show()
```



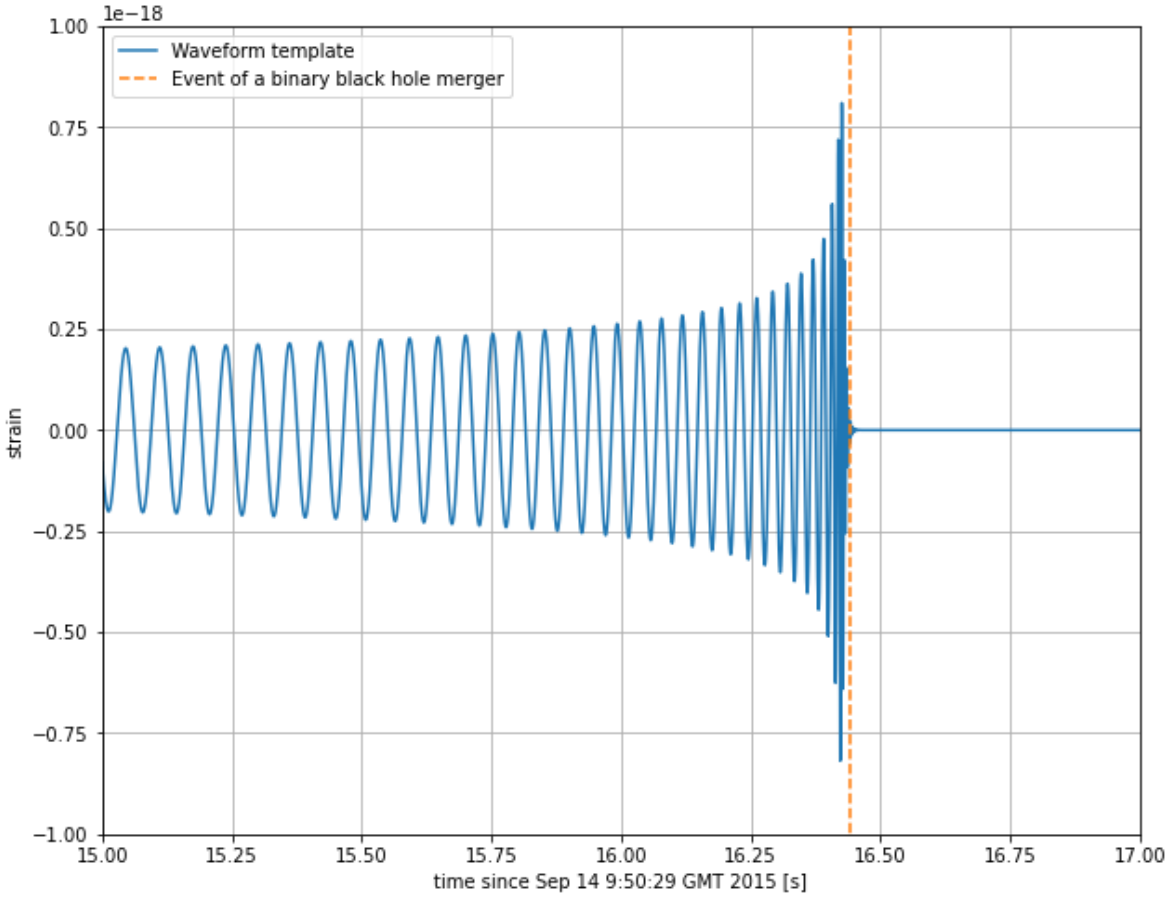


In the above plot, we only show the data between 20 Hz and 2000 Hz.

You see strong spectral lines in the data; they are all of instrumental origin. However, you can't see the signal in this plot, since it is relatively weak and less than a second long, while this plot averages over 32 seconds of data. So this plot is entirely dominated by instrumental noise.

Now, let's plot the best-matching gravitational waveform predicted by General Relativity.

In [12]:
```
# read in the template and parameters for the theoretical waveform
f_template = h5py.File(fn_template, "r")
template_p, template_c = f_template["template"][...]
time_template = time+(tevent-16)

plt.figure(figsize=(10,8))
plt.plot(time_template,template_p, label = 'Waveform template')
plt.plot(tevent*np.ones(2), [-2.0e-18, 1.e-18], '--', label = 'Event o

plt.xlim(15, 17); plt.ylim(-1.0e-18, 1.0e-18)
plt.xlabel('time since Sep 14 9:50:29 GMT 2015 [s]'); plt.ylabel('stra
plt.grid('on')
plt.legend()
plt.show()
```



Following https://arxiv.org/pdf/1710.04635.pdf (https://arxiv.org/pdf/1710.04635.pdf), we model the gravitational wave signal by making analogies to electromagnetic waves.

Assume that two masses orbit their common center of mass. The orbiting system emits gravitational waves, which carry energy away from the system, leading to a decay of the orbits, that is, there is an “in-spiral” of the masses.

Starting from an expression that indicates how the orbital frequency changes as the binary system radiates energy, we can find how the distance between two masses (denoted as r) changes with time:

$$\frac{dr}{dt} = \frac{-\eta N c}{4}\left(\frac{r_s}{r}\right)^3$$

where r_s is the Schwarzschild radius, the distance from the center of a spherically symmetric mass at which the escape speed, calculated from Newtonian mechanics in this case, is equal to the speed of light. For two binary masses, $r_s = 2GM/c^2$. $\eta = m_1 m_2/(m_1+m_2)^2$ is a dimensionless raio of the masses, and N is a numerical factor which depends on the details of the theory used to derive the result. Here, $\eta = 0.247, N = 6.4, M = 60.5 M_{\text{sun}}$.

We can now find the waveform with the following calculational steps:

1. Let $r_i = 5.3 r_s$ (initial separation of two masses) and t_i = time_template[64805] (16.26s since Sep 14 9:50:29 GMT 2015). Assume $t_f = 16.4$. Use 10000 steps, so $\Delta t = (t_f - t_i)/10000$

2. Solve the above differential equation in three different ways:

(1) Analytically: $r(t) = \left[r_i^4 - N\eta r_s^3 c(t-t_i)\right]^{1/4}$

(2) Euler's Method (numerically): Let $f(r) = \frac{dr}{dt}$. Then, $r(t+\Delta t) = r(t) + \Delta t \cdot f(r)$ for small Δt.

(3) Runge-Kutta Method (numerically): Let $f(r) = \frac{dr}{dt}$.
$k_1 = \Delta t \cdot f(r), k_2 = \Delta t \cdot f(r + \frac{1}{2}k_1)$. Then, $r(t+\Delta t) = r(t) + k_2$ for small Δt.

Yes, it is silly to solve this DEQ numerically since its analytic solution can be found very easily. But if you have nonlinear equations, (2) and (3) can be very useful.

3. Calculate r for the current time value. (Δt should be very small.)

4. Use Kepler's third law to find $w(t)$ (which shows orbital frequency) and calculate w for the current time value.

$$w^2 = \frac{GM}{r^3}$$

.

5. Use that value of w and calculate the interferometer strain signal $h(t)$:

$$h(t) = \frac{\Delta L_x(t) - \Delta L_y(t)}{L} \propto w^{2/3}\cos(2w(t-t_i))$$

5. For the next time value, repeat step 3-5 to find r, w, h.

6. Stop when t reaches 16.4 ($t_f = 16.4$).

7. Scale $h(t)$ so that $h(t_i)$ = template_p[64805]. (Remember that t_i = time_template[64805].)

5. Solving the given differential equation in three different ways (analytically, using Euler's method, using Runge-Kutta Method), find three different versions of $h(t)$. Then, show how our estimate of the waveform ($h(t)$) matches with the actual waveform ("template_p"); first plot "template_p" as a function of "time_template" and then plot three different versions of $h(t)$ in the same figure. Try different $M = 55M_{\text{sun}}, 60.5M_{\text{sun}}, 65M_{\text{sun}}$ and show that $M = 60.5M_{\text{sun}}$ fits best to the given waveform (for $M = 55M_{\text{sun}}, 65M_{\text{sun}}$, plot $h(t)$ using only the analytic solution. For $M = 60.5M_{\text{sun}}$, try all three methods to plot $h(t)$.) Label all plots. set the x limits of the axes as plt.xlim(ti, 16.45).

In [13]:
```
G = 6.67408e-11
Msun = 1.99e30
c = 2.99792e8

N = 32/5
eta = 0.247

ti = time_template[64805]
```

In [14]:
```
tf = 16.4
times, delta_t = np.linspace(ti, tf, num=int(1e4), retstep=True)
```

In [15]:
```
def r_sch(M):
  """
  Schwarzschild radius
  """
  return 2 * G * M / c**2

def r_init(rs):
  """
  Initial separation
  """
  return 5.3 * rs
```

In [16]:
```
def w(r, M):
  return np.sqrt(G * M / r**3)

def h(w, t):
  return w ** (2/3) * np.cos(2 * w * (t -ti))
```

In [17]:
```
# initialize arrays with r, w, h
# col0: analytical, 1: euler, 2: runge-kutta
rvals = np.empty((int(1e4), 3))
wvals = np.empty_like(rvals)
hvals = np.empty_like(rvals)

# mass
M = 60.5 * Msun
rs = r_sch(M)
ri = r_init(rs)
```

In [18]:
```
# Method 1: Analytic solution
def r_ana(t, rs, ri):
  return (ri ** 4 - N * eta * rs**3 * c * (t - ti))**(1/4)

rvals[:, 0] = r_ana(times, rs, ri)
wvals[:, 0] = w(rvals[:, 0], M)
hvals[:, 0] = h(wvals[:, 0], times)
```

In [19]:
```
# we need dr/dt for the other two methods
def dr_dt(r):
  return - eta * N * c / 4 * (rs/r)**3
```

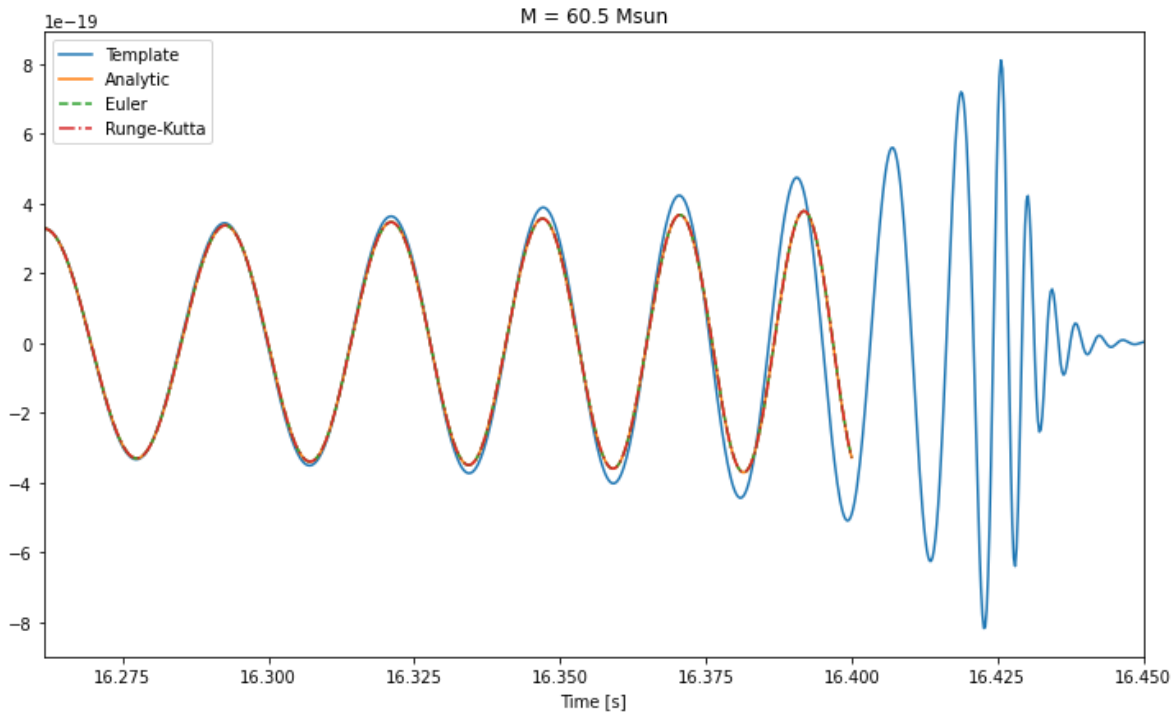
In [20]:
```
# Method 2: Euler's method
r = ri
for i, t in enumerate(times):
  if i > 0:
    r += dr_dt(r) * delta_t
  rvals[i, 1] = r
  wvals[i, 1] = w(r, M)
  hvals[i, 1] = h(w(r, M), t)
```

In [21]:
```
# Method 3: Runge-Kutta Method
r = ri
for i, t in enumerate(times):
  if i > 0:
    k1 = dr_dt(r) * delta_t
    k2 = dr_dt(r + k1/2) * delta_t
    r += k2
  rvals[i, 2] = r
  wvals[i, 2] = w(r, M)
  hvals[i, 2] = h(w(r, M), t)
```

In [22]:
```
# recale h, constant of proportionality = C
C = template_p[64805] / hvals[0, 0]  # C the same for all methods
hvals *= C
```

In [23]:
```
plt.figure(figsize=(12, 7))
plt.plot(time_template, template_p, label="Template")
plt.plot(times, hvals[:, 0], label="Analytic")
plt.plot(times, hvals[:, 1], ls="--", label="Euler")
plt.plot(times, hvals[:, 2], ls="-.", label="Runge-Kutta")
plt.legend()
plt.xlim(ti, 16.45)
plt.title("M = 60.5 Msun")
plt.xlabel("Time [s]")
```

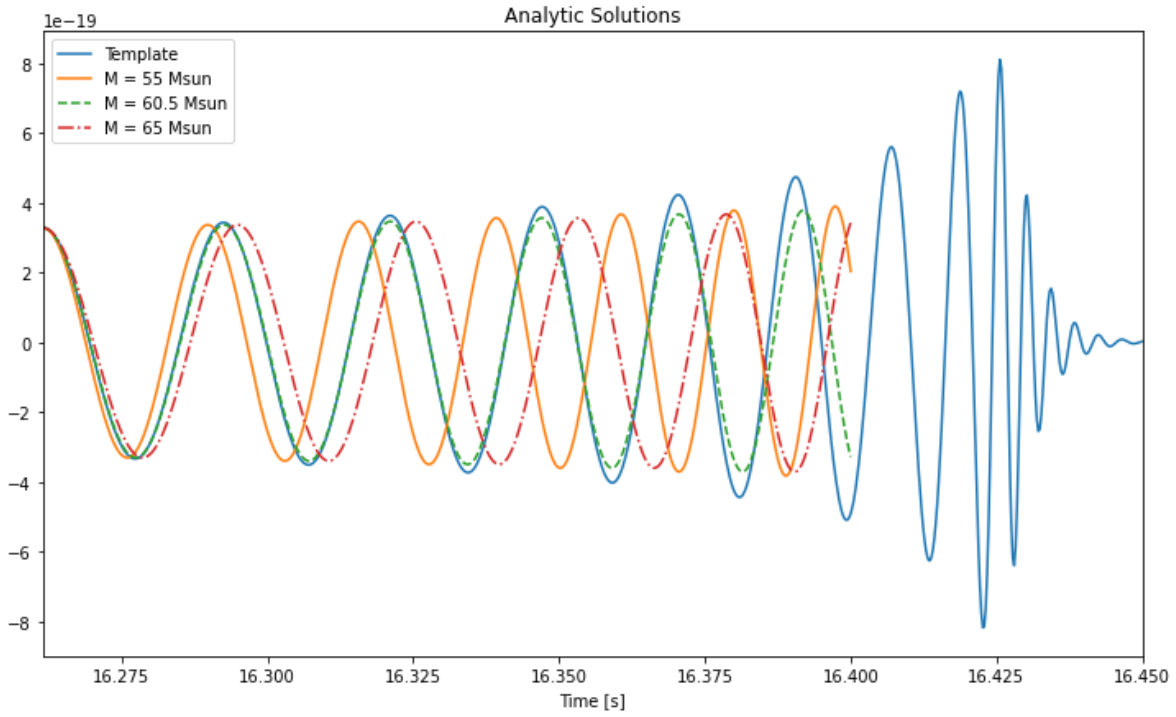
Out[23]:
```
Text(0.5, 0, 'Time [s]')
```



In [24]:
```
# different masses
# initialize empty h array, each col is for different mass
h_ana = np.empty_like(hvals)
h_ana[:, 1] = hvals[:, 0]  # already computed for M = 60.5 Msun

# M = 55 Msun
M = 55 * Msun
rs = r_sch(M)
ri = r_init(rs)
r55 = r_ana(times, rs, ri)
h_ana[:, 0] = h(w(r55, M), times)
h_ana[:, 0] *= template_p[64805] / h_ana[0, 0]

# M = 65 Msun
M = 65 * Msun
rs = r_sch(M)
ri = r_init(rs)
r65 = r_ana(times, rs, ri)
h_ana[:, 2] = h(w(r65, M), times)
h_ana[:, 2] *= template_p[64805] / h_ana[0, 2]
```

In [25]:
```
plt.figure(figsize=(12, 7))
plt.plot(time_template, template_p, label="Template")
plt.plot(times, h_ana[:, 0], label="M = 55 Msun")
plt.plot(times, h_ana[:, 1], ls="--", label="M = 60.5 Msun")
plt.plot(times, h_ana[:, 2], ls="-.", label="M = 65 Msun")
plt.legend()
plt.xlim(ti, 16.45)
plt.title("Analytic Solutions")
plt.xlabel("Time [s]")
```

Out[25]:
```
Text(0.5, 0, 'Time [s]')
```



You should find that with $M = 60.5 M_{\rm sun}$, $h(t)$ agrees reasonably well with the actual waveform which uses numerical GR. Our estimate breaks down as we get closer to the merger event.

From the ASD in Part 4, we can see that noise fluctuations are much larger at low and high frequencies and near spectral lines, reaching a roughly flat ("white") minimum in the band

around 80 to 300 Hz.

We can "whiten" the data (dividing it by the noise amplitude spectrum, in the fourier domain), suppressing the extra noise at low frequencies and at the spectral lines, to better see the weak signals in the most sensitive band. Note that only the data are needed in this process.

To get rid of remaining high frequency noise, we will also bandpass the data.

Now, define a function called "whiten" which takes the strain data ("strain"), interpolated psd ("interp_psd"), and dt (1.0 / fs = 1.0 / 4096).

Note:

1. For L1 and H1, strain = strain_L1 and strain = strain_H1 respectively.

2. Let Pxx_H1 be the psd values you obtained in Part 4 for H1. Then, interpolate this as "psd_H1 = interp1d(freqs, Pxx_H1)" (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.interpolate.interp1d.html (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.interpolate.interp1d.html)). psd_H1 is a function; to get the psd values at the needed frequencies (say you have an array "freqs"), you should do psd_H1(freqs).

3. So "whiten(strain_H1,psd_H1,1.0/fs)" would whiten the strain data for H1.

6. Define a function "whiten". Then, whiten the H1 and L1 strain data. (Replace ellipsis)

In [26]:
```
# transform to freq domain, divide by asd, then transform back
def whiten(strain, interp_psd, dt):
    # Given strain data, first transform to frequency domain
    # Use "np.fft.rfftfreq" to get the frequencies and "np.fft.rfft" t
    freqs = np.fft.rfftfreq(len(strain), d=dt)
    hf = np.fft.rfft(strain)

    # Normalization factor
    norm = 1./np.sqrt(1./(dt*2))

    # Divide by the ASD and multiply by the normalization factor
    white_hf = hf / np.sqrt(interp_psd(freqs)) * norm

    # Transform back to the time domain using "np.fft.irfft" (Hint: n
    # https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/nump
    white_ht = np.fft.irfft(white_hf, n=len(strain))
    return white_ht
```

In [27]:
```
dt = 1.0 / fs

# Interpolate the PSD values
psd_H1 = interp1d(freqs, PSD_H1)
psd_L1 = interp1d(freqs, PSD_L1)

strain_H1_whiten = whiten(strain_H1,psd_H1,dt)
strain_L1_whiten = whiten(strain_L1,psd_L1,dt)
```

Remember that the sampling rate ("fs") is 4096 Hz. What is the nyquist frequency?
(Note: The data cannot capture frequency content above the Nyquist frequency, but that's OK, because GW150914 only has detectable frequency content in the range 20 Hz - 300 Hz.)

In [28]:
```
nyq = fs/2

print(f"The Nyquist frequency is {nyq:.0f} Hz")
```

```
The Nyquist frequency is 2048 Hz
```

Now, suppress the high frequency noise (no signal there) with some bandpassing. (We bandpass from 43 to 300 Hz, since we are looking for a signal in that frequency band and the detector noise is pretty low there. This will keep power in this frequency range, but suppress power away from this frequency.) We use scipy.signal.butter (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.signal.butter.html (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.signal.butter.html)) and scipy.signal.filtfilt (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.signal.filtfilt.html (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.signal.filtfilt.html))

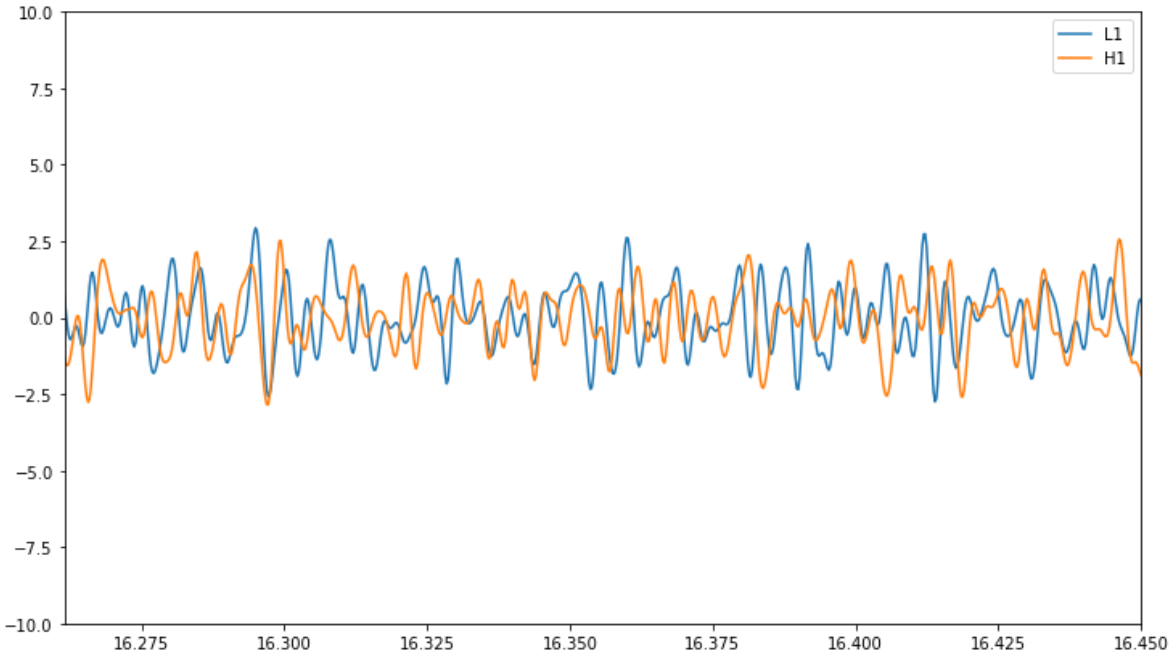
In [29]:
```
# bandpass filter coefficients
order = 4
lowcut= 43
highcut= 360
low = lowcut / nyq
high = highcut / nyq

bb, ab = butter(order, [low, high], btype='band')
normalization = np.sqrt((highcut-lowcut)/nyq)
strain_H1_whitenbp = filtfilt(bb, ab, strain_H1_whiten) / normalizatio
strain_L1_whitenbp = filtfilt(bb, ab, strain_L1_whiten) / normalizatio
```

Shift L1 event by 7 ms (Signal arrived 7 ms earlier at L1)

In [30]:
```
strain_L1_shift = -np.roll(strain_L1_whitenbp,int(0.007*fs))
```

Plot "strain_L1_shift" (whitened L1 strain shifted by 7ms) and "strain_H1_whitenbp" (whitened H1 strain) as a function of time. Set "plt.xlim(ti, 16.45); plt.ylim(-10, 10)". Do you see a signal?

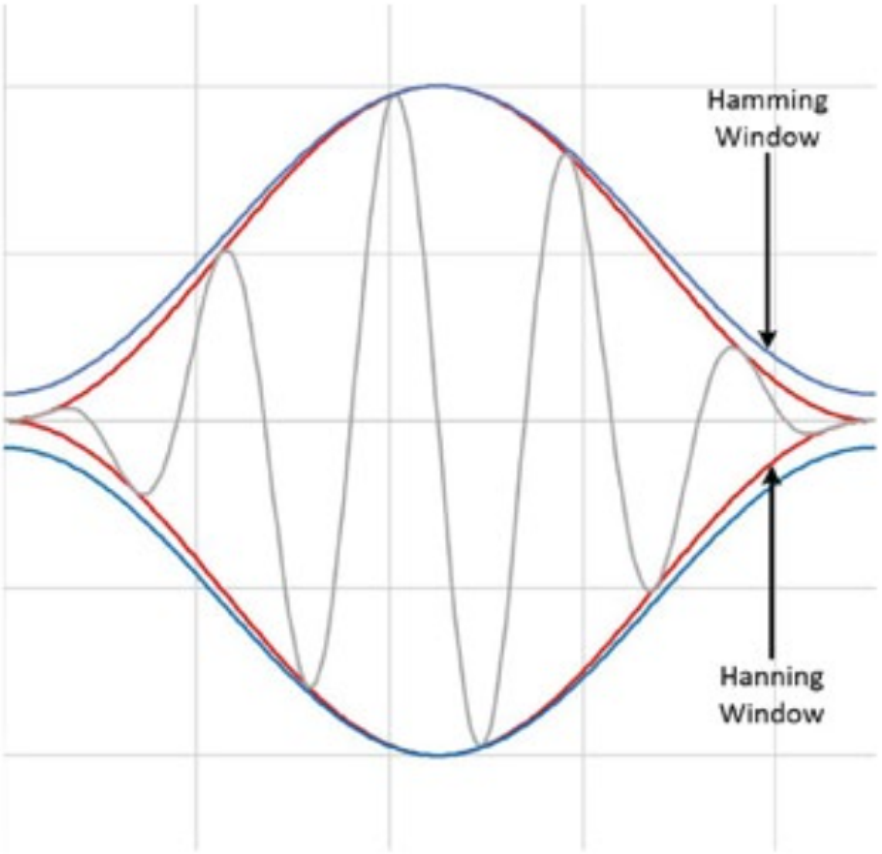
In [31]:
```
plt.figure(figsize=(12, 7))
plt.plot(time_template, strain_L1_shift, label="L1")
plt.plot(time_template, strain_H1_whitenbp, label="H1")
plt.xlim(ti, 16.45)
plt.ylim(-10, 10)
plt.legend()
plt.show()
```



The FFT input signal is inherently truncated. This truncation can be modelled as multiplication

of an infinite signal with a rectangular window function. In the spectral domain this multiplication becomes convolution of the signal spectrum with the window function spectrum, being of form sin(x)/x. This convolution is the cause of an effect called spectral leakage (https://en.wikipedia.org/wiki/Spectral_leakage (https://en.wikipedia.org/wiki/Spectral_leakage))

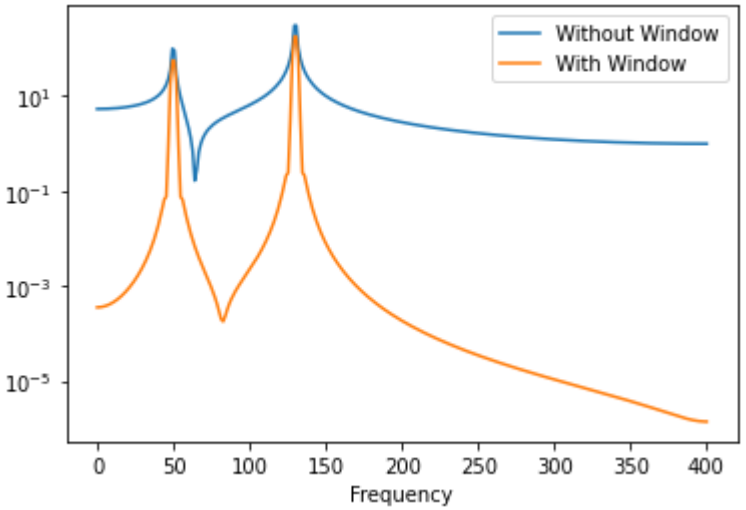
Hence, spectral leakage is caused by discontinuities in the original, noninteger number of periods in a signal. Windowing the signal with a dedicated window function helps mitigate spectral leakage by reducing the amplitude of the discontinuities at the boundaries



In this project, we use the Blackman window (https://docs.scipy.org/doc/numpy/reference/generated/numpy.blackman.html#numpy.blackman (https://docs.scipy.org/doc/numpy/reference/generated/numpy.blackman.html#numpy.blackman)).

7. Recall that in Part 2 we found the FFT of $f(t)$ using np.fft.rfft. Now, define a window function "w = np.blackman(N)" and multiply $f(t)$ by w and do the FFT. $\left(\text{np.fft.rfft(w*f(t))}\right)$. Plot the FFT of $f(t)$ with and without the window function. Set the y-axis as log-scale (plt.semilogy). Show that windowing reduces spectral leakage.

In [32]:
```
N = 600
fs = 800
t = np.arange(N) / fs

w = np.blackman(N)

# Use rfft without window
yf = np.fft.rfft(f(t))
# Use rfft with window
yf_win = np.fft.rfft(w*f(t))

rfreq = np.fft.rfftfreq(N, d=1./fs)

plt.figure()
plt.plot(rfreq, np.abs(yf), label="Without Window")
plt.plot(rfreq, np.abs(yf_win), label="With Window")
plt.legend()
plt.xlabel("Frequency")
plt.yscale("log")
plt.show()
```



It is clear that the FFT with the window concetrates the power into fewer Fourier modes (the ones corresponding to the frequency of the sines that make up f(t). Without the window, there's significant power (orders of magnitude more than in the case with the window) further away from the peaks.

Matched filtering is the optimal way to find a known signal buried in stationary, Gaussian noise. It is the standard technique used by the gravitational wave community to find GW signals from compact binary mergers in noisy detector data. LIGO scientists use matched filtering to find such "hidden" signals. A matched filter works by compressing the entire signal into one time bin (by convention, the "end time" of the waveform).

Here, we use only one template (the one identified in the full search as being a good match to the data). Assuming that the data around this event is fairly Gaussian and stationary, we'll use this simple method to identify the signal (matching the template) in our 32 second stretch of data.

8. Do the matched filtering. First, start with the L1 strain data.

In [33]:
```
# To calculate the PSD of the data, choose an overlap and a window
fs = event['fs']
dt = 1.0 / fs
NFFT = 4*fs
psd_window = np.blackman(NFFT)
# and a 50% overlap:
NOVL = NFFT/2

# Complex waveform template
template = (template_p + template_c*1.j)

# the length and sampling rate of the template MUST match that of the
datafreq = np.fft.fftfreq(template.size, d = 1./fs)
df = np.abs(datafreq[1] - datafreq[0])
```

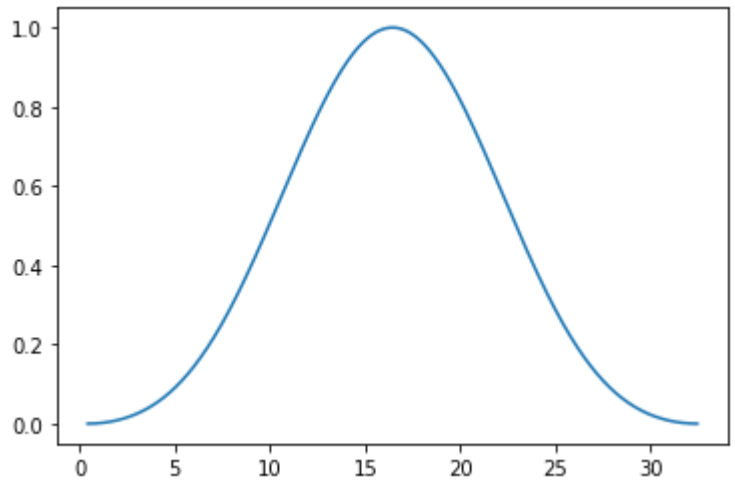
Window the data. (Replace ellipsis in the cell below)

1. Define the window: "np.blackman(template.size)"
2. Do the FFT of the template with window using "np.fft.fft": multiply "template" by "dwindow" and then do FFT. The normalization factor is "1/fs" (i.e. multiply the FFT of the template by 1/fs)
3. Do the FFT of the L1 strain data with window using "np.fft.fft": multiply the L1 strain data by "dwindow" and then do FFT. The normalization factor is "1/fs" (i.e. multiply the FFT of the template by 1/fs)

In [34]:
```
# window the data (Replace ellipsis)

# 1. Define the window
dwindow = np.blackman(template.size)

# 2. Do the FFT of the template (with dwindow)
template_fft = np.fft.fft(dwindow * template) / fs

# 3. Take the Fourier Transform (FFT) of the data (with dwindow)
data = strain_L1.copy()
data_fft = np.fft.fft(dwindow * data) / fs
```

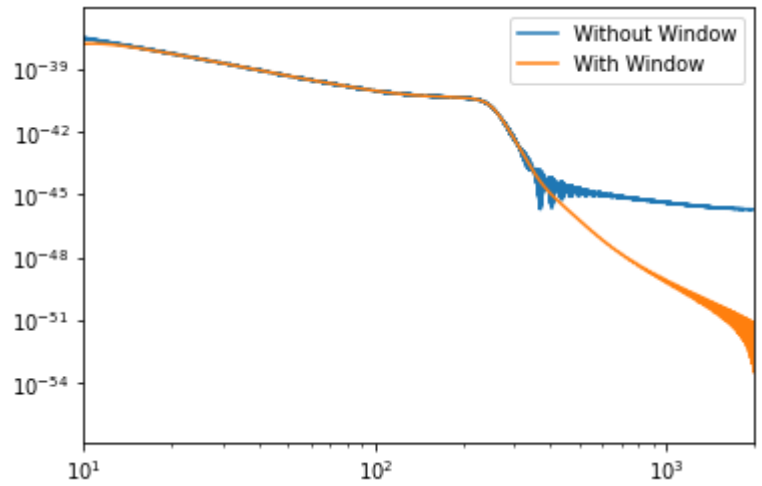
Plot the blackman window as a function of time. Do you think using this window is useful in our analysis? Note that the merger event occurred around 16s since Sep 14 9:50:29 GMT 2015, and the data stretches over 32s.

In [35]:
```
plt.figure()
plt.plot(time_template, dwindow)
plt.show()
```



Plot the FFT of the template with and without the window (as a function of frequency). Only use positive frequencies. Plot the squared magnitude of the FFT (power spectrum). Label all plots. Set the range of x-axis between 10 and 2000. Plot both axes in log scale.

In [36]:
```
# without window
template_no_win = np.fft.fft(template) / fs

plt.figure()
plt.plot(
  datafreq[:template.size//2],
  np.abs(template_no_win[:template.size//2])**2,
  label="Without Window",
)
plt.plot(
  datafreq[:template.size//2],
  np.abs(template_fft[:template.size//2])**2,
  label="With Window",
)
plt.legend()
plt.xscale("log")
plt.yscale("log")
plt.xlim(10, 2000)
plt.show()
```

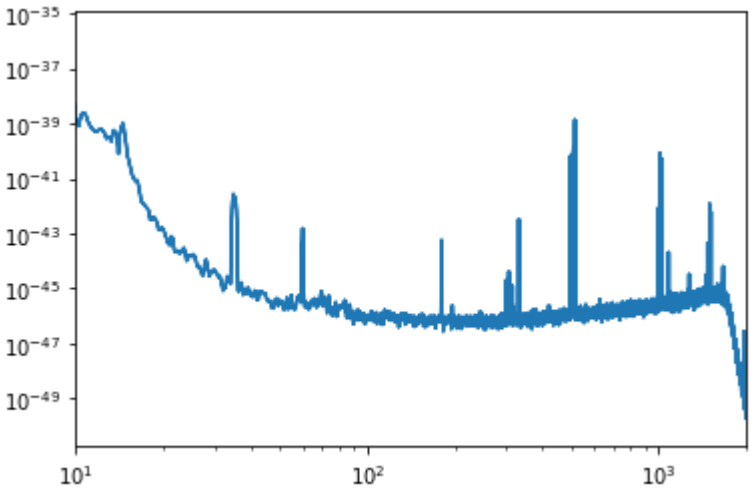


Next, we need an estimate of the noise power in each FFT bin.

In [37]:
```
# First, calculate the PSD of the data.  Also use an overlap, and wind
data_psd, freqs = mlab.psd(data, Fs = fs, NFFT = NFFT, window=psd_wind
# Interpolate to get the PSD values at the needed frequencies. This de
power_vec = np.interp(np.abs(datafreq), freqs, data_psd)
```

Plot the noise power spectral density (defined by "power_vec"). Set the range of x-axis between 10 and 2000. Plot both axes in log scale.

In [38]:
```
plt.figure()
plt.plot(np.abs(datafreq), power_vec)
plt.xscale("log")
plt.yscale("log")
plt.xlim(10, 2000)
plt.show()
```



Calculate the matched filter output in the time domain. (Replace ellipsis in the cell below)

FFT of data

Template; can be generated in frequency domain using stationary phase approximation

$$C(t) = 4\int_0^\infty \frac{\tilde{s}(f)\ \tilde{h}^*(f)}{S_n(f)}\ e^{2\pi i f t}\ df$$

Noise power spectral density

Look for maximum of $|C(t)|$ above some threshold ➔ trigger

1. Multiply the Fourier Space template and data, and divide by the noise power in each frequency bin: i.e. Multiply the FFT of the data with the conjugate (using ".conjugate()") of the FFT of the template and divide by the noise power ("power_vec")
2. Taking the Inverse Fourier Transform (IFFT) of the filter output (take it back to the time domain): i.e. Take the inverse FFT (using np.fft.ifft) of the step 1 output and multiply by the factor "2*fs"

Hence, the result will be plotted as a function of time off-set between the template and the data:

In [39]:
```
# 1. Multiply the Fourier Space template and data, and divide by the n
optimal = data_fft * template_fft.conjugate() / power_vec

# 2. Taking the Inverse Fourier Transform (IFFT) of the filter output
optimal_time = np.fft.ifft(optimal)
```

Normalize the matched filter output. (Replace ellipsis in the cell below)

1. Find $\sigma = \sqrt{\langle \frac{\tilde{h}(f)\tilde{h}^*(f)}{S_n(f)} \rangle}$ (Hint: First multiply the FFT of the template by its conjugate (using ".conjugate()") and then divide by the noise PSD. Basically you did $\frac{\tilde{h}(f)\tilde{h}^*(f)}{S_n(f)}$. You have an array then (each array element correspond to each element in the frequency array "datafreq" you defined earlier). Now, sum array elements and multiply by "df" which you defined earlier.
2. Normalize the matched filter output so that we expect a value of 1 at times of just noise: i.e. Divide the absolute value of "optimal_time" by sigma.

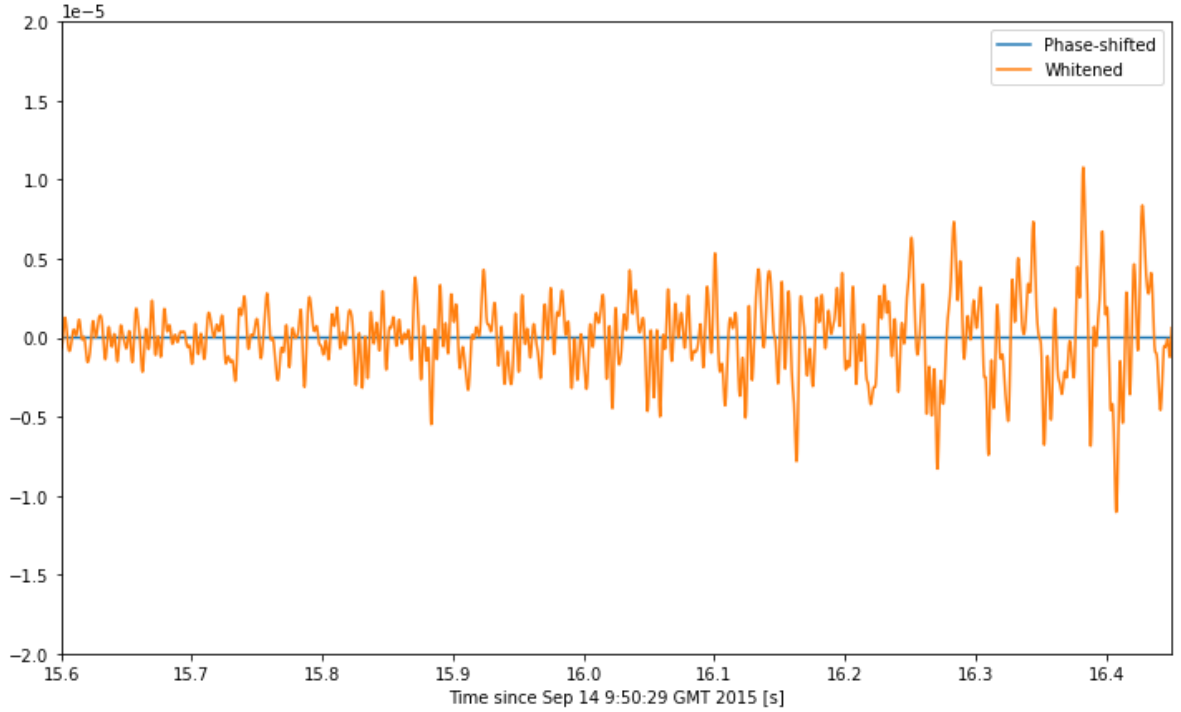
In [40]:
```
# 1. Find $\sigma$
sigmasq = np.sum(template_fft * template_fft.conjugate() / power_vec)
sigma = np.sqrt(sigmasq.real)

# 2. Normalize the matched filter output so that we expect a value of
SNR = np.abs(optimal_time) / sigma
```

Apply time offset, phase to template. Finally, whiten the template and band-pass it. (All routine given - nothing to do here, just run the cell.)

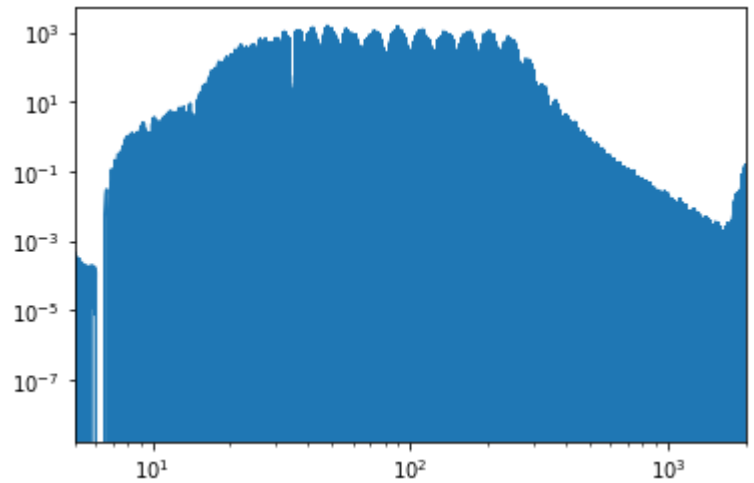
In [41]:
```
# shift the SNR vector by the template length so that the peak is at t
peaksample = int(data.size / 2)  # location of peak in the template
SNR_complex = optimal_time/sigma
SNR_complex = np.roll(SNR_complex,peaksample)
SNR = abs(SNR_complex)

# apply time offset, phase to template
# find the time and SNR value at maximum:
indmax = np.argmax(SNR)
timemax = time[indmax]
SNRmax = SNR[indmax]

d_eff = sigma / SNRmax
# Calculate optimal horizon distnace
horizon = sigma/8

# Extract time offset and phase at peak
phase = np.angle(SNR_complex[indmax])
offset = (indmax-peaksample)

# apply time offset, phase, and d_eff to template
template_phaseshifted = np.real(template*np.exp(1j*phase))    # phase
template_rolled = np.roll(template_phaseshifted,offset) / d_eff  # App

# Whiten and band-pass the template for plotting
template_whitened = whiten(template_rolled,interp1d(freqs, data_psd),d
template_match = filtfilt(bb, ab, template_whitened) / normalization #
```

Plot the original, phase-shifted template ("template_rolled") and the whitened template ("template_whitened"). Show the plot between the time range (15.6, 16.45) since Sep 14 9:50:29 GMT 2015. You will see that the waveform before ~ 16.3s is largely suppressed. That is because we divide the FFT of the template by the noise PSD in the Fourier space during whitening. Now, calculate FFT(template)/Noise PSD$^{1/2}$ and plot it as a function of frequency in the log-log plot. Set the x-axis range between 5 and 2000 Hz.

In [44]:
```
plt.figure(figsize=(12, 7))
plt.plot(time_template, template_rolled, label="Phase-shifted")
plt.plot(time_template, template_whitened, label="Whitened")
plt.xlabel("Time since Sep 14 9:50:29 GMT 2015 [s]")
plt.xlim(15.6, 16.45)
plt.ylim(-2e-5, 2e-5)
plt.legend()
plt.show()

plt.figure()
plt.loglog(datafreq, template_fft/np.sqrt(power_vec))
plt.xlim(5, 2e3)
plt.show()
```

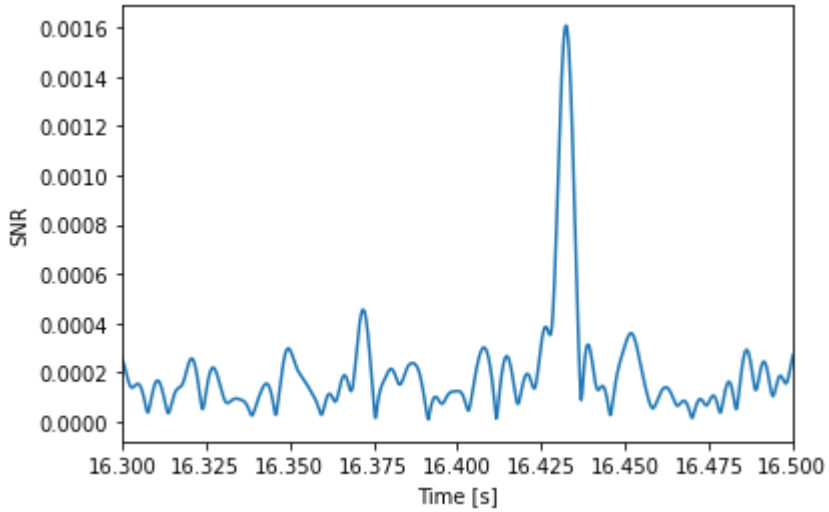



```
/usr/local/lib/python3.8/dist-packages/matplotlib/cbook/__init__.py:13
  return np.asarray(x, float)
```



Plot SNR as a function of time. (Remember that the GW signal arrived 7 ms earlier at L1. So shift the L1 time accordingly - this is done conventionally.) Set plt.xlim(16.3, 16.5).

In [49]:
```
plt.figure()
plt.plot(time, SNR)
plt.xlim(16.3, 16.5)
plt.xlabel("Time [s]")
plt.ylabel("SNR")
plt.show()
```



In [50]:
```
# Shift L1 event by 7 ms (Signal arrived 7 ms earlier at L1)
template_match_shift_L1 = -np.roll(template_match,int(0.007*fs))
```

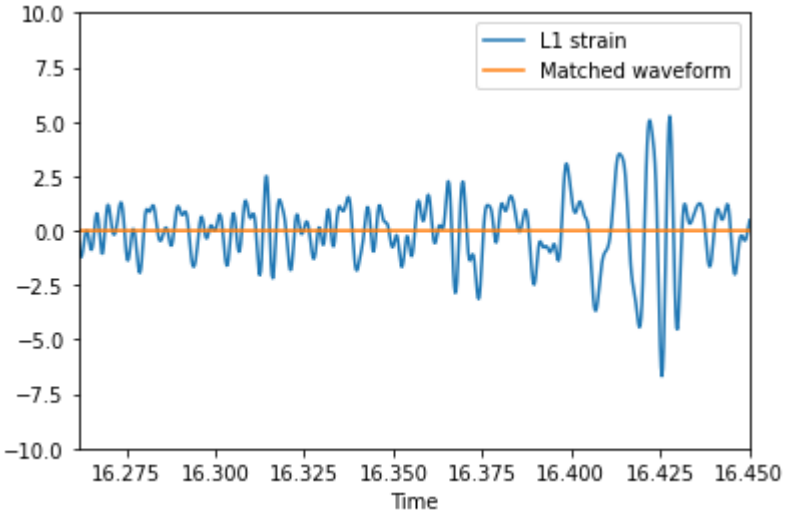
At what time does SNR peak? Print the result. (Remember that the GW signal arrived 7 ms earlier at L1. So shift the L1 time accordingly - this is done conventionally.)

In [56]:
```
print(f"SNR peaks at {time[np.argmax(SNR)]:.3f} s")
```

```
SNR peaks at 16.432 s
```

First, plot the whitened strain data (from Part 6) as a function of time. Then, plot the matched waveform ("template_match_shift_L1") in the same figure. Show that the whitened strain data match well to the waveform predicted by GR.

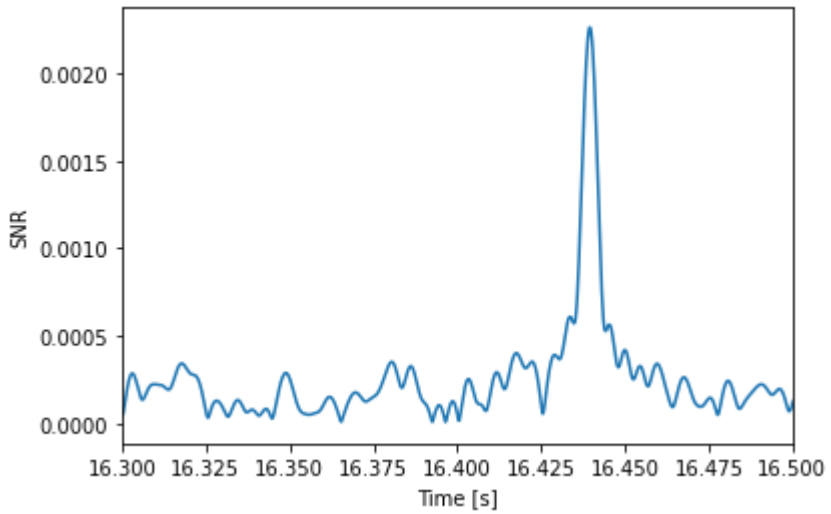
In [54]:
```
plt.figure()
plt.plot(time, strain_L1_shift, label="L1 strain")
plt.plot(time, template_match_shift_L1, label="Matched waveform")
plt.legend()
plt.xlabel("Time")
plt.xlim(ti, 16.45)
plt.ylim(-10, 10)
plt.show()
```



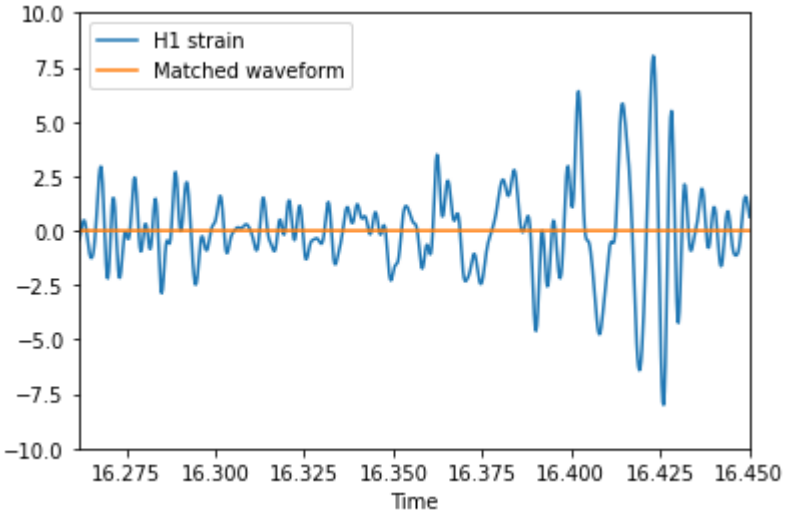
9. Repeat Part 8 for the H1 strain data and do: 1. Plot SNR vs. time, 2. plot the whitened strain data and matched waveform vs. time, 3. find the time at which SNR peaked.

In [60]:
```
data = strain_H1.copy()
data_fft = np.fft.fft(dwindow * data) / fs

data_psd, freqs = mlab.psd(data, Fs = fs, NFFT = NFFT, window=psd_wind
power_vec = np.interp(np.abs(datafreq), freqs, data_psd)

optimal = data_fft * template_fft.conjugate() / power_vec
optimal_time = np.fft.ifft(optimal)

sigmasq = np.sum(template_fft * template_fft.conjugate() / power_vec)
sigma = np.sqrt(sigmasq.real)


# shift the SNR vector by the template length so that the peak is at t
peaksample = int(data.size / 2)  # location of peak in the template
SNR_complex = optimal_time/sigma
SNR_complex = np.roll(SNR_complex,peaksample)
SNR = abs(SNR_complex)

# apply time offset, phase to template
# find the time and SNR value at maximum:
indmax = np.argmax(SNR)
timemax = time[indmax]
SNRmax = SNR[indmax]

d_eff = sigma / SNRmax
# Calculate optimal horizon distnace
horizon = sigma/8

# Extract time offset and phase at peak
phase = np.angle(SNR_complex[indmax])
offset = (indmax-peaksample)

# apply time offset, phase, and d_eff to template
template_phaseshifted = np.real(template*np.exp(1j*phase))    # phase
template_rolled = np.roll(template_phaseshifted,offset) / d_eff  # App

# Whiten and band-pass the template for plotting
template_whitened = whiten(template_rolled,interp1d(freqs, data_psd),d
template_match = filtfilt(bb, ab, template_whitened) / normalization #
```

In [62]:
```
plt.figure()
plt.plot(time, SNR)
plt.xlim(16.3, 16.5)
plt.xlabel("Time [s]")
plt.ylabel("SNR")
plt.show()
```



In [66]:
```
plt.figure()
plt.plot(time, strain_H1_whitenbp, label="H1 strain")
plt.plot(time, template_match, label="Matched waveform")
plt.legend()
plt.xlabel("Time")
plt.xlim(ti, 16.45)
plt.ylim(-10, 10)
plt.show()
```



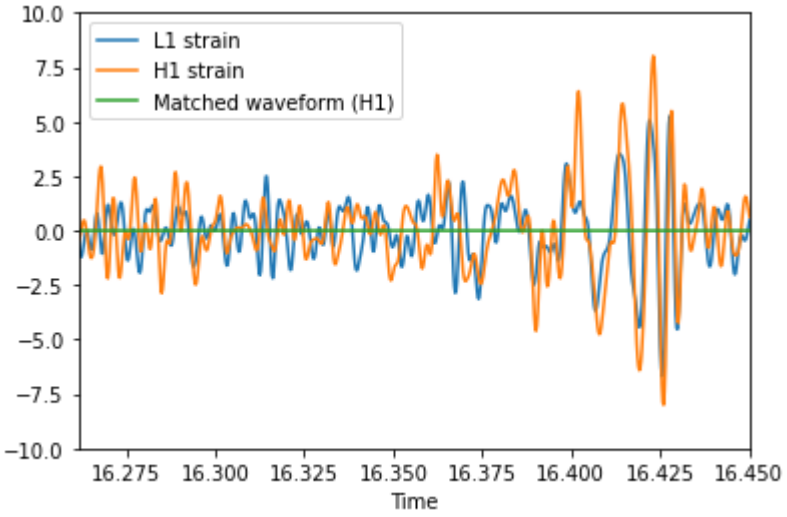
In [63]:
```
print(f"SNR peaks at {time[np.argmax(SNR)]:.3f} s")
```

```
SNR peaks at 16.439 s
```

10. First, compare the SNR peak time for both H1 and L1 data. (Print the results) Then, plot the whitened L1 and H1 strain in the same plot. Plot the waveform matched to the H1 strain data in the same plot as well.

We printed above that the peak SNR for H1 is at 16.439 s and the peak SNR for L1 is at 16.432 s. This lines up with the signal arriving 7 ms earlier at L1.

In [68]:
```
plt.figure()
plt.plot(time, strain_L1_shift, label="L1 strain")
plt.plot(time, strain_H1_whitenbp, label="H1 strain")
plt.plot(time, template_match, label="Matched waveform (H1)")
plt.legend()
plt.xlabel("Time")
plt.xlim(ti, 16.45)
plt.ylim(-10, 10)
plt.show()
```



So we conclude that the whitened strain data match well to the waveform predicted by General Relativity; we have detected the gravitational wave predicted by GR!

Extra (all the routine given; just run the cell!):

Make wav (sound) files from the filtered, downsampled data, +-2s around the event.

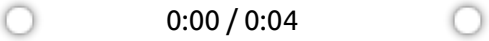
In [69]:
```
eventname = 'GW150914'

Pxx = (1.e-22*(18./(0.1+freqs))**2)**2+0.7e-23**2+((freqs/2000.)*4.e-2
psd_smooth = interp1d(freqs, Pxx)

template_offset = 16.

# function to keep the data within integer limits, and write to wavfil
def write_wavfile(filename,fs,data):
    d = np.int16(data/np.max(np.abs(data)) * 32767 * 0.9)
    wavfile.write(filename,int(fs), d)

deltat_sound = 2.                    # seconds around the event

# index into the strain time series for this time interval:
indxd = np.where((time >= tevent-deltat_sound) & (time < tevent+deltat_

# write the files:
write_wavfile(eventname+"_H1_whitenbp.wav",int(fs), strain_H1_whitenbp
write_wavfile(eventname+"_L1_whitenbp.wav",int(fs), strain_L1_whitenbp

# re-whiten the template using the smoothed PSD; it sounds better!
template_p_smooth = whiten(template_p,psd_smooth,dt)

# and the template, sooming in on [-3,+1] seconds around the merger:
indxt = np.where((time >= (time[0]+template_offset-deltat_sound)) & (t
write_wavfile(eventname+"_template_whiten.wav",int(fs), template_p_smo
```

With good headphones, you may be able to hear a faint thump in the middle; that's our signal!

In [70]:
```
fna = eventname+"_template_whiten.wav"
print(fna)
Audio(fna)
```

```
GW150914_template_whiten.wav
```

Out[70]:



In [71]:
```
fna = eventname+"_H1_whitenbp.wav"
print(fna)
Audio(fna)
```

```
GW150914_H1_whitenbp.wav
```

Out[71]: