Fast Fourier Transforms

Fast Fourier Transforms

- Fourier transforms are useful
- Can we calculate them *efficiently*?

The Fourier transform treats continuous functions defined over infinite domains

$$
\begin{aligned}
H(f) &= \int_{-\infty}^{\infty} h(t) \exp(-2\pi \mathrm{i} f t) \mathrm{d}t \\
h(t) &= \int_{-\infty}^{\infty} H(f) \exp(2\pi \mathrm{i} f t) \mathrm{d}f
\end{aligned}
$$

We can represent a signal over a finite domain with discrete coefficients

If our signal is measured over a period $0 \leq t \leq T$, and if we assume $h(t)$ is periodic, the Fourier transform becomes a Fourier series with components at discrete frequencies $f_n = n/T$.

$$
\begin{aligned}
H_n &= \int_{0}^{T} h(t) \exp(-2\pi \mathrm{i} n t/T) \mathrm{d}t \\
h(t) &= \frac{1}{T} \sum_{n=-\infty}^{\infty} H_n \exp(2\pi \mathrm{i} n t/T)
\end{aligned}
$$

i.e. an **infinite** number of discrete frequencies can represent a **continuous** periodic signal.

Can sample the signal in the time domain to make the time domain discrete

Consider sampling the signal $h(t)$ at N **uniformly-spaced** points:

$$t_k = k\Delta, k = 0, 1, \ldots (N-1)$$

with $T = N\Delta$

The **sampling rate** is $f_s = 1/\Delta$.

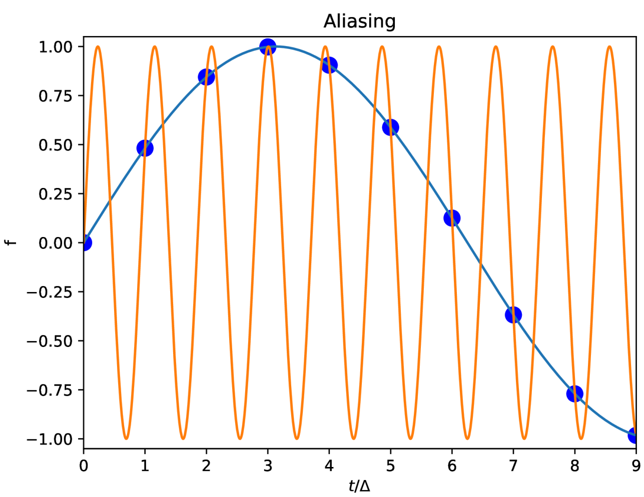
There is a **maximum representable frequency** in such a signal, called the **Nyquist critical frequency** f_c, where we have two samples per cycle:

$$f_c = \frac{1}{2\Delta}$$

Nyquist-Shannon sampling theorem: If $h(t)$ is **band-limited** to frequencies $|f| < f_c$ (i.e. if $H(f) = 0$ for $|f| > f_c$) then $h(t)$ is completely determined by h_n.

Higher frequencies than the Nyquist frequency are aliased because the frequency domain is now periodic

The frequency $f + 2f_c = f + (1/\Delta)$ produces exactly the same samples as f:



Ideally we **bandpass filter** the signal before sampling to ensure it is bandwidth-limited and then no aliasing can occur.

The Discrete Fourier Transform (DFT) is periodic and discrete in both domains

Given our N samples h_k, we can construct N frequencies which approximate the continuous Fourier transform, with the highest frequency being the critical frequency f_c. It's simplest for now to assume that N is even. We define the **discrete Fourier transform** as

$$H_n = \sum_{k=0}^{N-1} h_k \, e^{-2\pi \mathrm{i}\, k\, n/N}$$

which maps N time-domain samples into N frequencies, which are

$$f_n = \frac{n}{N\Delta} = \frac{2n}{N} f_c$$

We now have a discrete signal in the time and frequency domains, with the functions **periodic in both domains**: $h_{k+N} = h_k$ and $H_{n+N} = H_n$.

We can represent negative frequencies using positive indices

The discrete frequencies are $f_n = n/(N\Delta) = 2nf_c/N$, with n running from $n = 0$ to $(N-1)$:

- $n = 0$ is zero frequency (sum of input values)
- $1 \le n \le (N/2)$ are positive frequencies, with $(N/2)$ being the highest (Nyquist critical frequency f_c)
- $(N/2) + 1 \le n \le (N-1)$ can be thought of as **negative frequencies**: we can subtract $2f_c = 1/\Delta$ from them and they are the same because H_n is periodic

There is an exact inverse

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \, e^{2\pi \mathrm{i}\, k\, n/N}$$

Note the division by N.

To get the right scaling with respect to the continuous transform we use $H(f_n) \approx \Delta H_n$.

The DFT generalizes to higher dimensions straightforwardly

- If data lives d dimensions with N_i datapoints along dimension $i = 1, \dots d$

  $$F_{\mathbf{n}} = \sum_{\mathbf{j}} f_{\mathbf{j}} e^{-i\eta_{\mathbf{n}} \cdot \mathbf{j}}$$

  $\mathbf{j} = (j_1, \dots j_d)$ with $j_i = 0, \dots N_i - 1$ and $\eta_{\mathbf{n}} = 2\pi(n_1/N_1, \dots n_d/N_d)$ $n_i = 0, \dots N_i - 1$

The Fast Fourier Transform (FFT) is an efficient method for calculating the DFT

- How much computation is involved in a DFT? We can write

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k$$

where

$$W \equiv \exp(-2\pi \mathrm{i}/N)$$

- This looks like a matrix multiplication with a square matrix W whose $N \times N$ elements W_{nk} multiply the vector h_k of length N. This is an $\mathcal{O}(N^2)$ **process** i.e. its compute time is dominated by a number of complex multiplications proportional to N^2

The FFT is many orders of magnitude faster

- In fact, **the FFT algorithm can do the same job in $\mathcal{O}(N \log_2 N)$ operations**.
- For $N = 10^6$, $N^2/(N \log_2 N) \approx 50,000$. (50,000s is 14 hours)
- "Discovered" by Danielson & Lanczos (1942), computer discovery by Cooley and Tukey (1965), but the original ideas goes back at least as far as Gauss (1802).

The key to the FFT is writing a DFT of length N as the sum of two DFTs of length $N/2$

Split the DFT into odd and even terms: we can write H_n as

$$\sum_k^{\text{even}} h_k \exp(-2\pi \mathrm{i} kn/N) + \sum_k^{\text{odd}} h_k \exp(-2\pi \mathrm{i} kn/N) =$$

$$\sum_{m=0}^{\text{N/2-1}} h_k \exp(-2\pi \mathrm{i}(2m)n/N) + \sum_{m=0}^{\text{N/2-1}} h_k \exp(-2\pi \mathrm{i}(2m+1)n/N) =$$

$$\sum_{m=0}^{\text{N/2-1}} h_k \exp(-2\pi \mathrm{i} mn/(N/2)) + \exp(-2\pi \mathrm{i} n/N) \sum_{m=0}^{\text{N/2-1}} h_k \exp(-2\pi \mathrm{i} mn/(N/2))$$

Separate a big FT into 2 smaller FT

If N is a power of 2, we can apply this theorem recursively

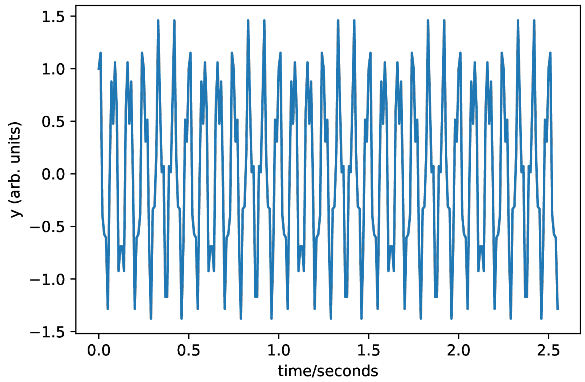
- Since $2(N/2)^2 = N^2/2$, two half-sized DFTs are $\sim 2\times$ quicker to calculate.
- After $\log_2(N)$ halvings we end up with $N = 1$.
- Highly recommended to use 2^N if efficiency important. Either design your experiment correctly, or **pad with zeros** until your data blocks have length 2^N!
- But fast techniques also exist when N can be factorised.
- Worst case: N is prime, but there exist algorithms to speed up even this case.

FFT Applications

1. Convolution of two signals: FFT each, multiply the FFTs, then inverse FFT back. For example, smooth an image using a Gaussian kernel.
2. **Filtering** a signal – closely related to convolution. We take a signal, FFT it, multiply the FFT by a function, then FFT back, e.g. low- or high-pass filtering.
3. Crystallography
4. Find the power spectrum (PSD) $|H_n|^2$
5. Optics: Fraunhofer (and Fresnel) diffraction, spatial/temporal coherence function
6. Signal processing. e.g. Freeview signals are transmitted using FFT: the data are cut into 8192-piece chunks, ($8192 = 2^{13}$), FFT'ed, transmitted, inverse FFT'ed on reception. In fact, FFTs are central algorithm in Digital Signal Processing (DSP)

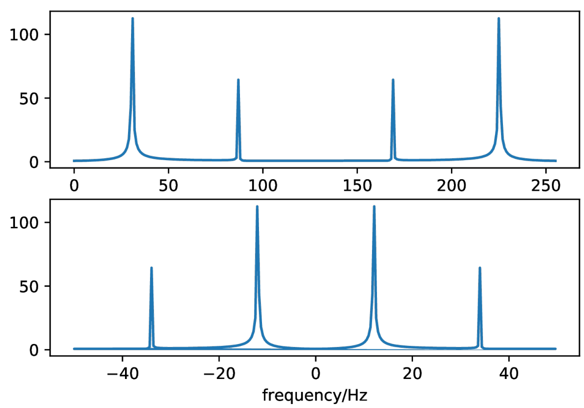
To test FFTs in Python, we first generate some fake data

```
import numpy as np
import matplotlib.pyplot as plt
dt=0.01
fftsize=256
t=np.arange(fftsize)*dt
#Generate some fake data at 12 Hz and 34 Hz
y=np.cos(2*np.pi*12*t)+0.5*np.sin(2*np.pi*34*t)
plt.plot(t,y)
```



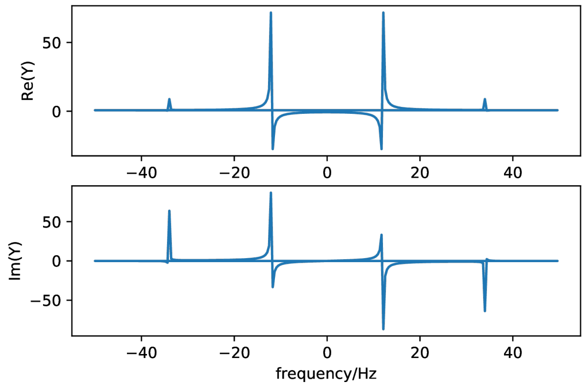
Negative frequencies come *after* positive frequencies

```
Y=np.fft.fft(y)
# Plot FFT modulus versus array index
plt.subplot(2,1,1); plt.plot(abs(Y))
# Now use the correct frequency coordinates
f=np.fft.fftfreq(fftsize,dt)
plt.subplot(2,1,2); plt.plot(f,abs(Y))
```



Negative frequencies are not always needed

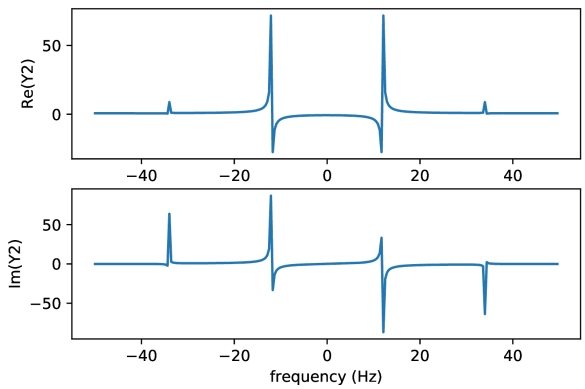
```
plt.subplot(2,1,1); plt.plot(f,Y.real)
plt.subplot(2,1,2); plt.plot(f,Y.imag)
```



Recall that the FT of a real signal has $H(f) = H^*(-f)$

Re-ordering the array makes plots tidier

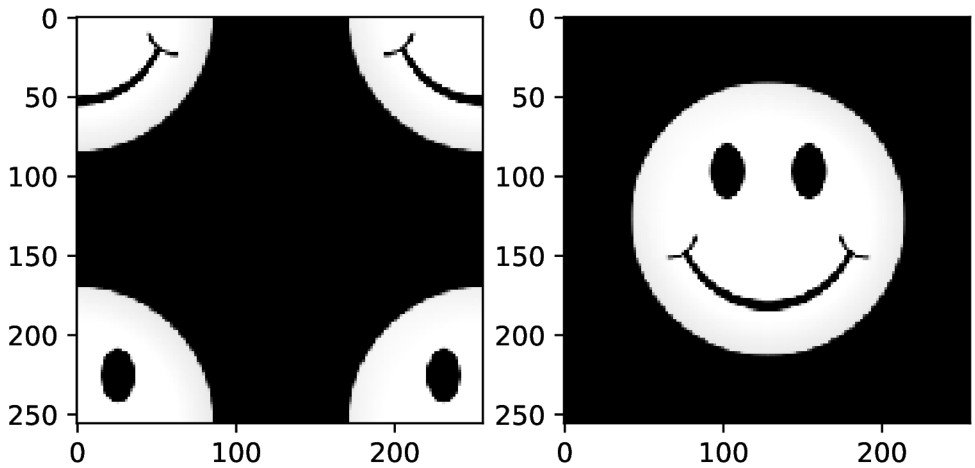
```
Y2=np.fft.fftshift(Y)
f2=np.fft.fftshift(f)
plt.subplot(2,1,1); plt.plot(f2,Y2.real)
plt.subplot(2,1,2); plt.plot(f2,Y2.imag)
```



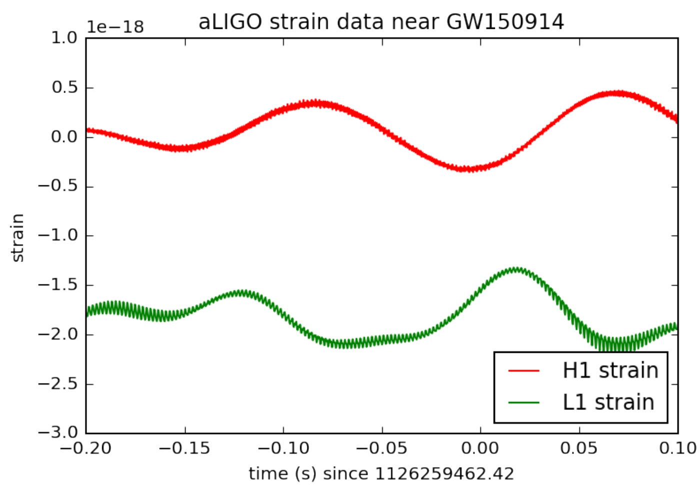
In 2-d negative-frequency re-ordering is even more helpful to visualise

Fourier results

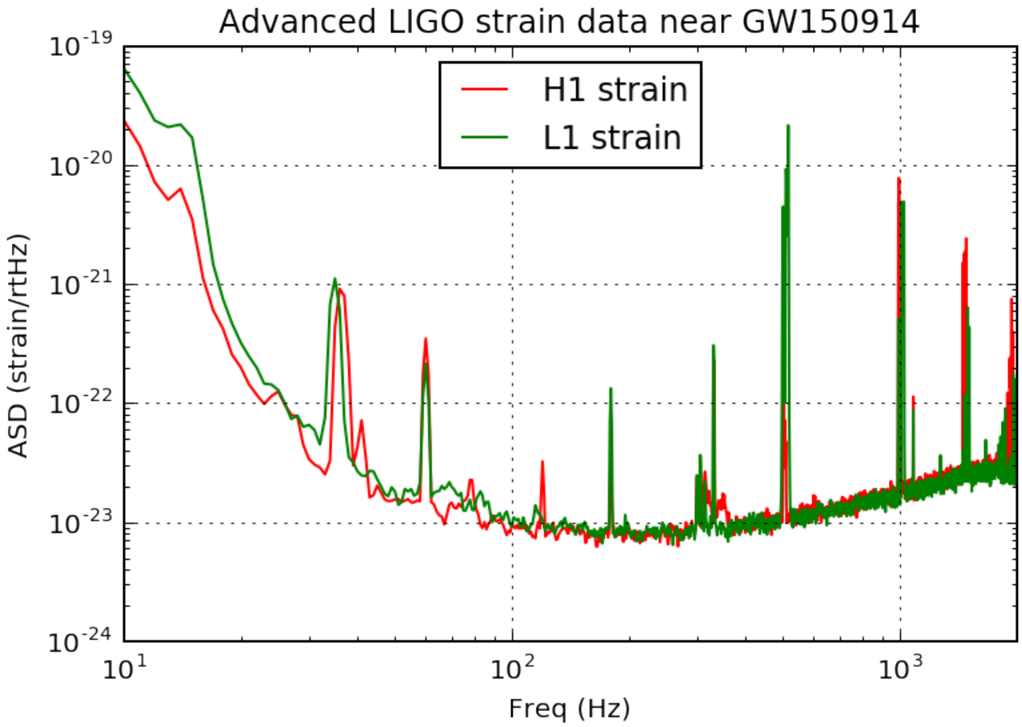
```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(smiley, cmap="gray")
ax2.imshow(np.fft.fftshift(smiley), cmap="gray")
```

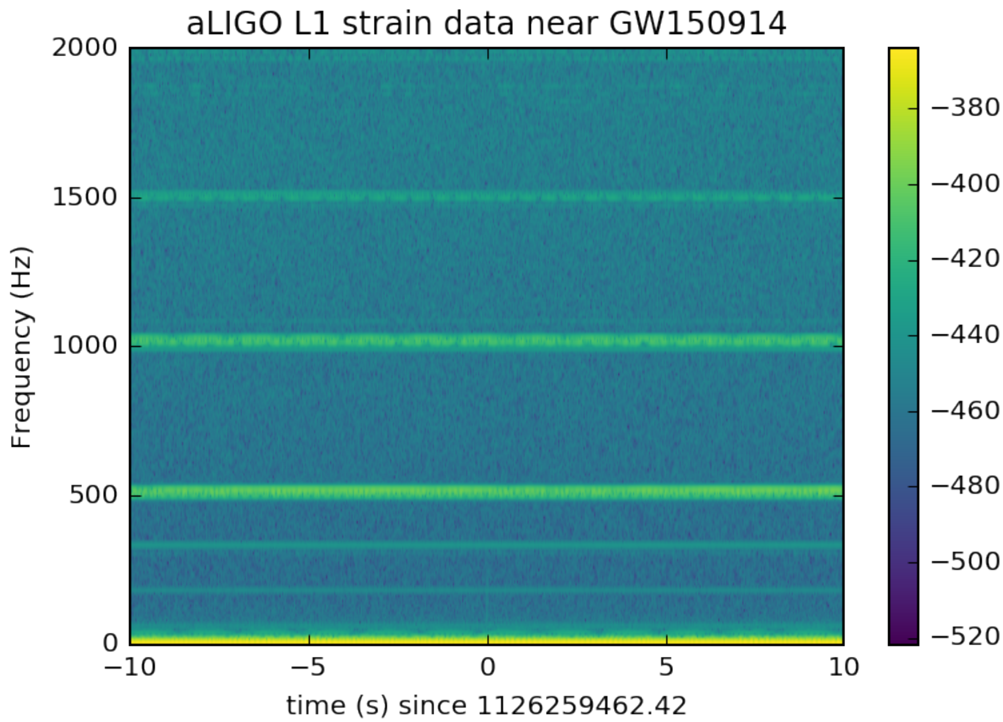


Signal Processing Example - LIGO

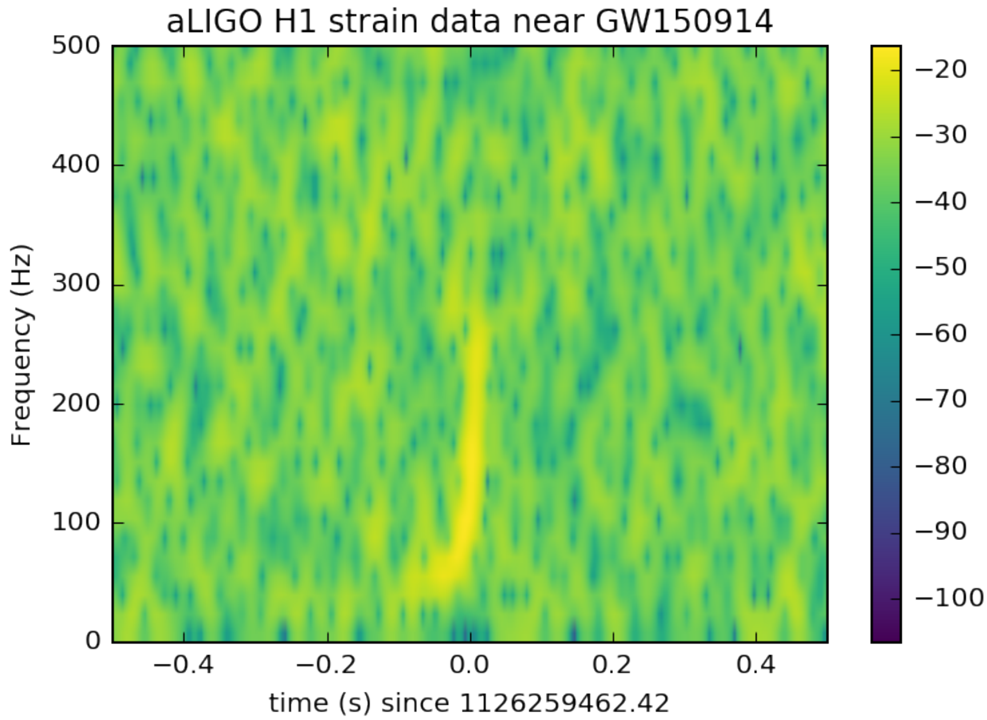


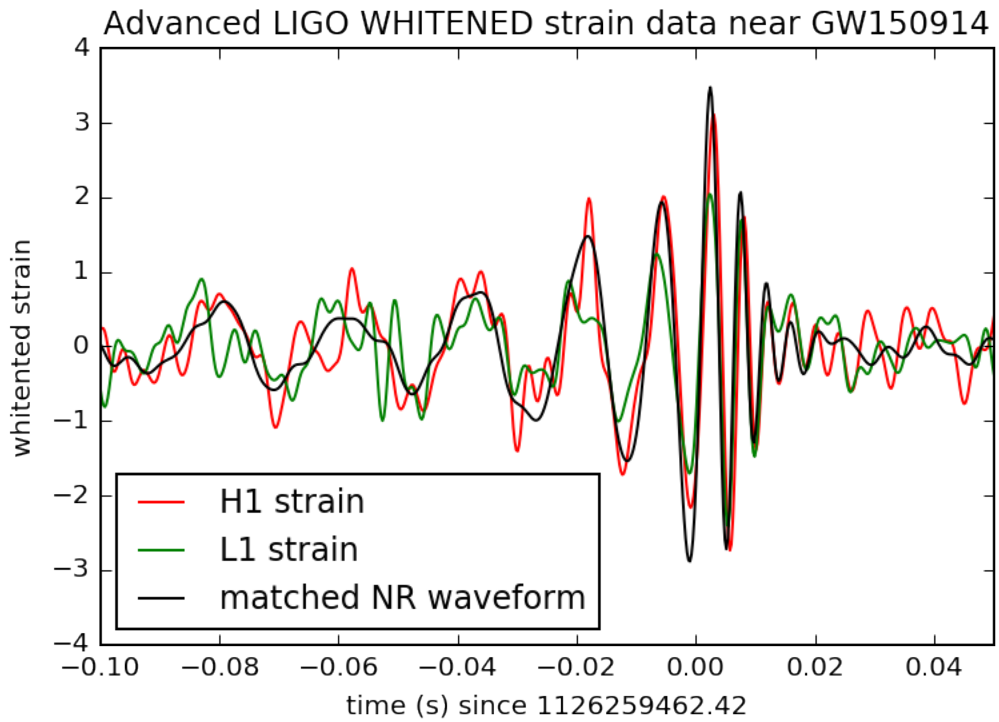
Plotting the data in the Fourier domain shows the noise sources




aLIGO L1 strain data near GW150914
2000
1500
1000
500
0
Frequency (Hz)
−10
−5
0
5
10
time (s) since 1126259462.42
−380
−400
−420
−440
−460
−480
−500
−520

We can Fourier filter the data and inverse transform to reveal the underlying signal




Advanced LIGO WHITENED strain data near GW150914
whitented strain
time (s) since 1126259462.42
H1 strain
L1 strain
matched NR waveform
4
3
2
1
0
−1
−2
−3
−4
−0.10
−0.08
−0.06
−0.04
−0.02
0.00
0.02
0.04