In [ ]:
```
# Created by Elia Gatti, UCL.
# This code takes inspiration and is heavily based on many web resources
# A shoutout to the github users curiores, emrebalak, and Thrifleganger
```

Modelling and designing embedded systems lab - Halloween Special

Spooktacular Signal Processing

WELCOME... To the creepy notebook. You are a well-known and respected IoT engineer by day, but it is at night that your true calling manifest itself. You hunt the streets at night with your patented IoT instruments, to look for mysteries and clues from beyond the veil.... Yesterday, your "spectrophone" (a microphone that records the screams of desperate disembodied souls) picked up something unusual..

You set off to analyze that mysterious signal.....

First, you need all the tools necessary to do your job of supernatural detective:

In [ ]:
```
import numpy as np
from scipy.io.wavfile import read
import matplotlib.pyplot as plt
from IPython.display import Audio
import scipy as sp
from scipy.io import wavfile
from scipy import signal
from scipy.signal import periodogram as periodogram_f
from scipy.fft import fftfreq, fftshift
from scipy.fft import fft, ifft, fft2, ifft2
import math
```

Now it's time to get down to work. First load the signal:

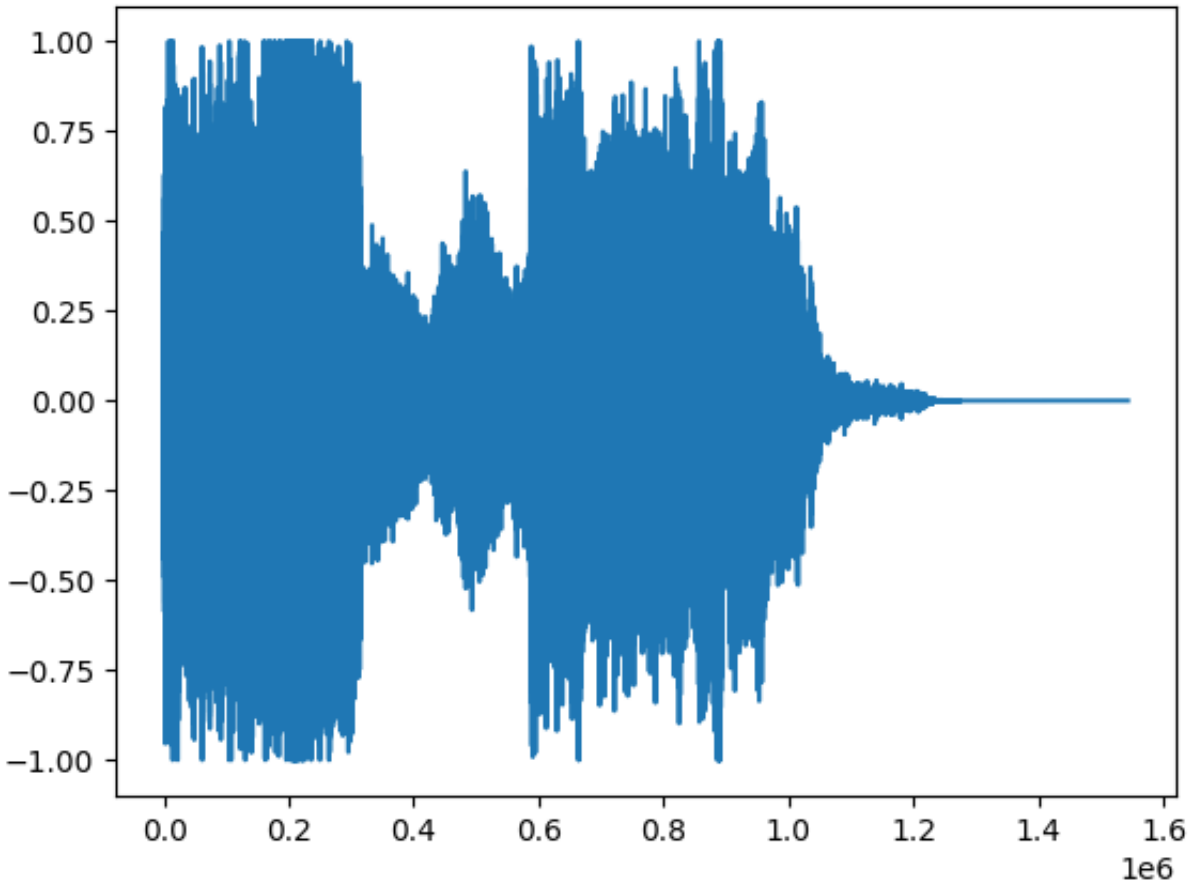
In [ ]:
```
SAMPLE_RATE = 44100
song = read("aphex_twin_equation.wav")[1][:,0]  # read song left stereo
song = song/np.max(np.abs(song)) #normalize song
Audio("aphex_twin_equation.wav")
```

Out[ ]: 0:23 0:35

Oh golly! That sounds sooooo spooky! It must certainly be a message from the another dimension! You suspect there is more to it than it meets the eye... It would be a good idea to find a way to visualize that signal...

In [ ]:
```
plt.plot(song)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x12bcf0b50>]
```



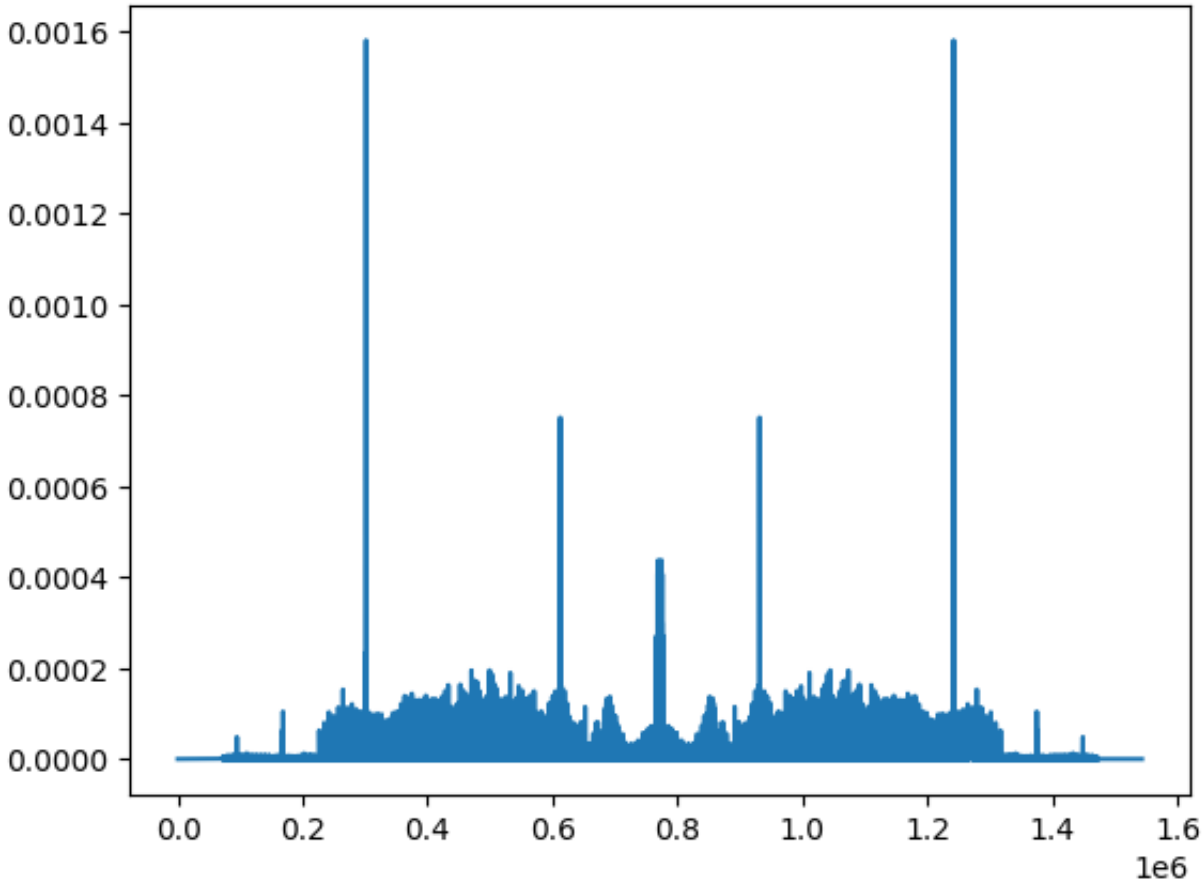
It doesn't look there is much there... your run of the mill representation in the temporal domain... But... Wait a minute! For the souls that passed, is time all that important? What does it matter time when ones has all the eternity? Not to mention this is clearly a sound from a SPECTRUM...ehm... spectre! We might have more luck looking at its spectral density!

Your tool of choice to estimate the spectral density of a stimulus is the Periodogram!

$$\hat{S}^{(p)}(f) = \frac{\Delta t |\mathrm{FT}[X_t](f)|^2}{N}$$

In [ ]:
```
periodogram = np.abs(fft(song))**2 / (SAMPLE_RATE * len(song))
plt.plot(fftshift(periodogram))
```

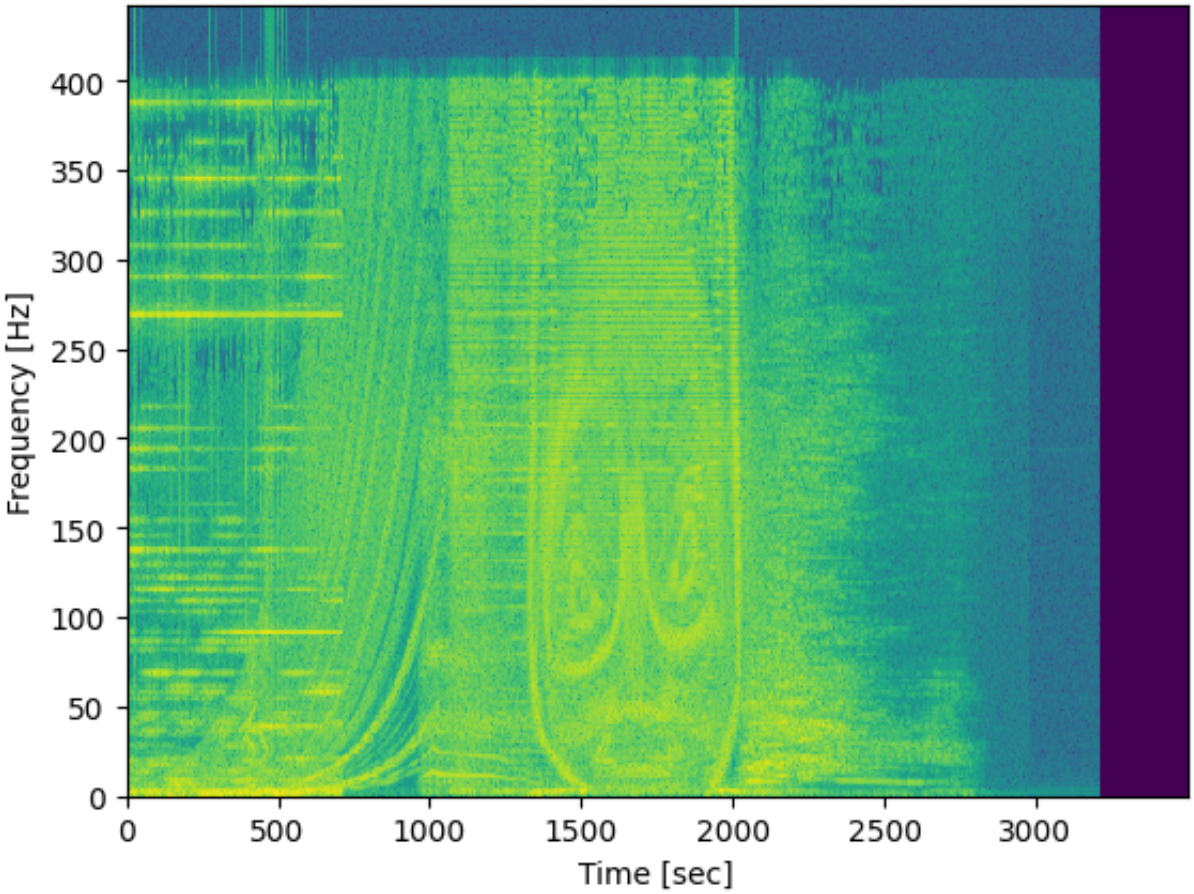
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x12c993940>]
```



Darn it! still no luck! Even the frequency domain does not hold the secret.... but wait a minute... you are in the real world... and time matters in your dimension! if only there was a way to visualize the frequency spectrum in time... You got it! You need a SPECTROGRAM!

In [ ]:
```
# The spectrogram is a spectral density estimation for "slices" of signal
# – Window Size: Size of the rectangular window.
# – Step: It is unnecessary to calculate the FFTs for every data point. T

window_size = int(0.02 * SAMPLE_RATE)
step_size = int(0.01 * SAMPLE_RATE)

# Extract windows and make hanning window (https://en.wikipedia.org/wiki/
leftover = (len(song) - window_size) % step_size
song = song[:len(song) - leftover] # remove leftover 56 data points from
nshape = (window_size, (len(song) - window_size) // step_size + 1) # (882
nstrides = (song.strides[0], song.strides[0] * step_size) # (8, 3528)
```

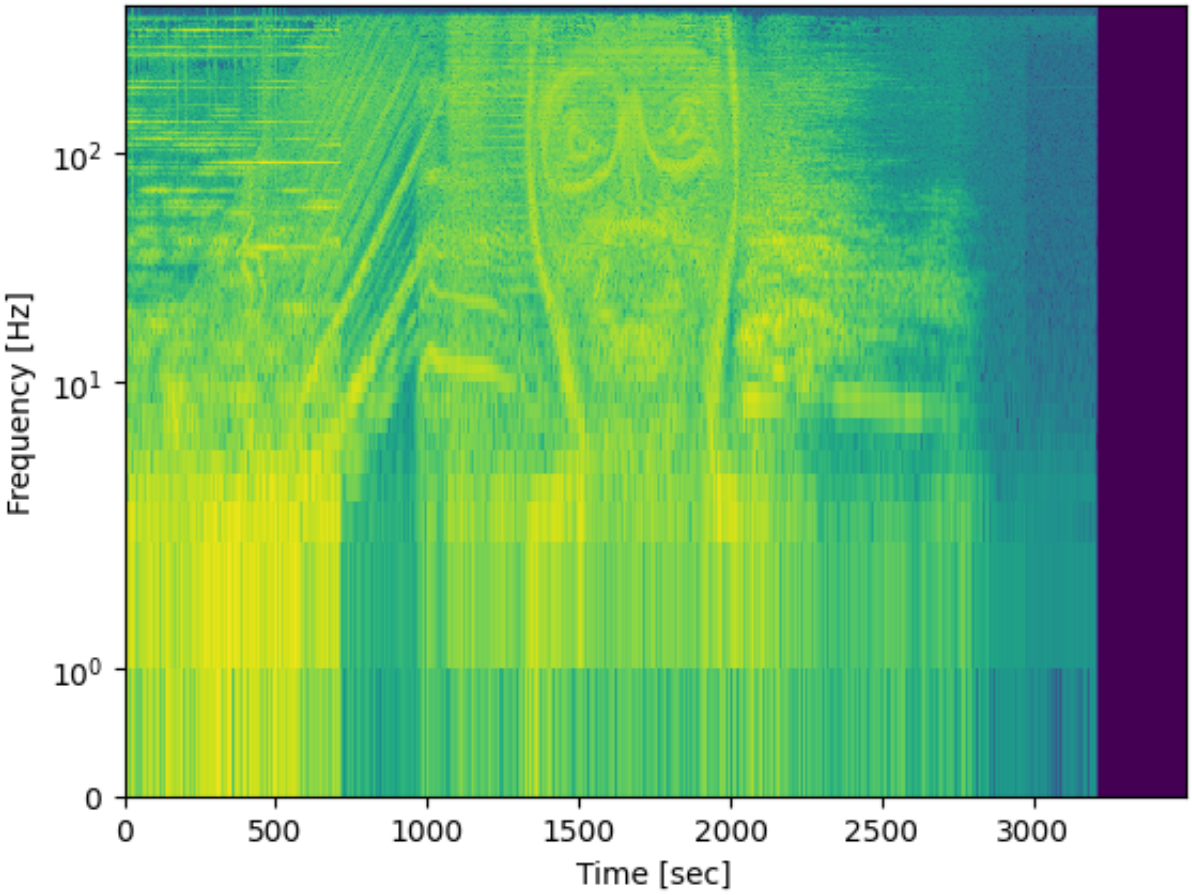
```
windows = np.lib.stride_tricks.as_strided(song, shape = nshape, strides =
# Window weighting
weighting = np.hanning(window_size)[:, None]


# Work on the first half of FFT and calculate absolute values
fft = np.fft.fft(windows * weighting, axis=0)
fft = fft[:442] # Since it's mirrored take first half
fft = np.absolute(fft) # absolute FFTs of the windowed data
fft = fft**2

# Use 10 * log10(x + c) to smooth the data for a better representation
spectrogram = 10 * np.log(fft + 1e-14) # 10 * log10(x + c) to smooth the

# The final 2D spectrogram matrix whichh rows and columns represent windo
plt.pcolormesh(spectrogram) # for obtaining a visual from the data
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.show()
```



Something seems to emerge..... When patterns do not seem clear, it is always a good idea to try different scales

In [ ]:
```
plt.pcolormesh(spectrogram) # for obtaining a visual from the data
plt.ylabel('Frequency [Hz]')
plt.yscale('symlog') # The message is hidden into the log scale.
plt.xlabel('Time [sec]')
```

```
plt.show()
```

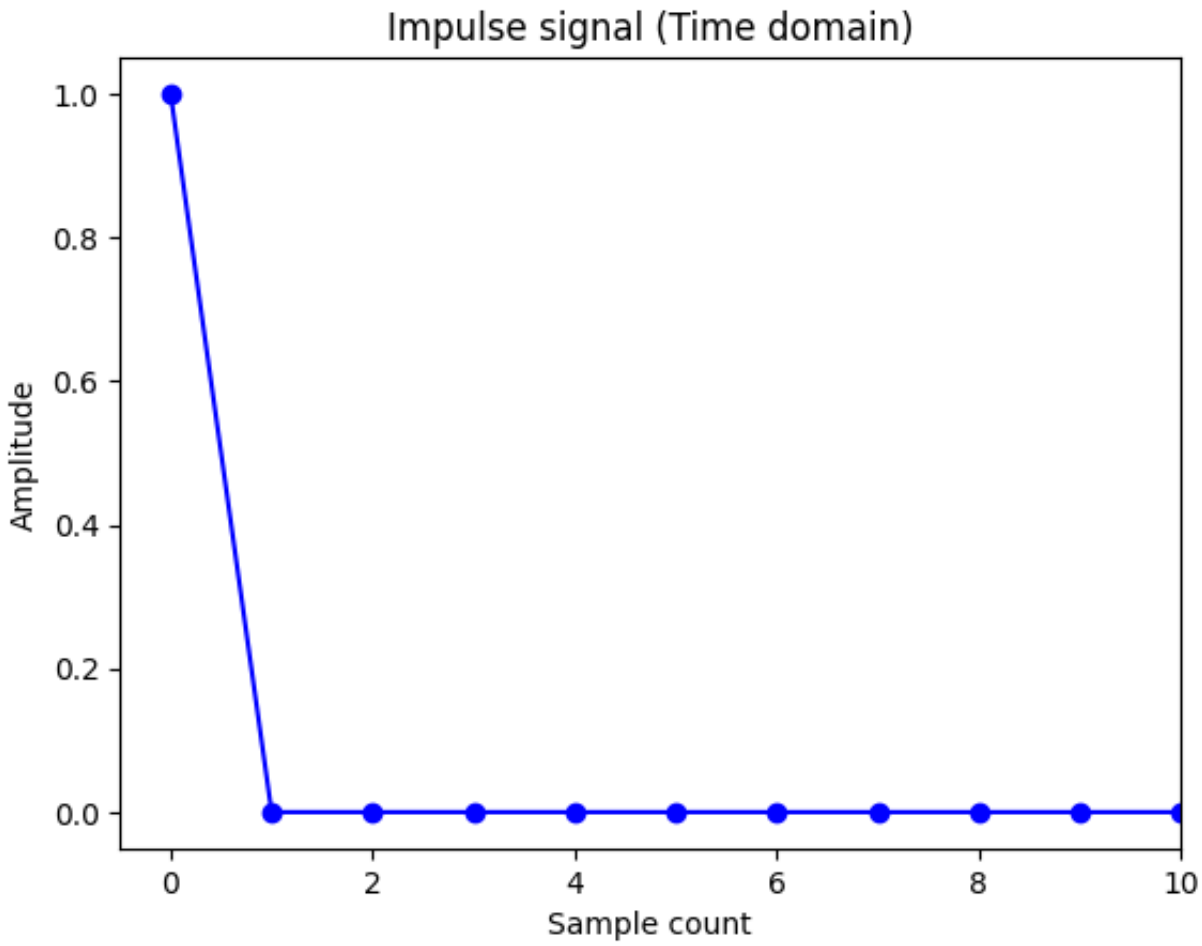


Ohhhhh!!! You got it! It is a real Ghost, and it is looking at you! Try to communicate with it... It is clearly a "digital" kind of ghost. You heard somewhere that sending impulse signals to digital ghost is a good way to understand their intentions. Impulse signals contain all possible frequencies after all.

Better generate some impulses then:

In [ ]:
```
frameSize = 1024

impulse = np.zeros(frameSize)
impulse[0] = 1;

plt.plot(impulse, 'bo-')
plt.xlabel('Sample count')
plt.ylabel('Amplitude')
plt.xlim(-0.5, 10)
plt.title("Impulse signal (Time domain)")

plt.show
```

Out[ ]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```

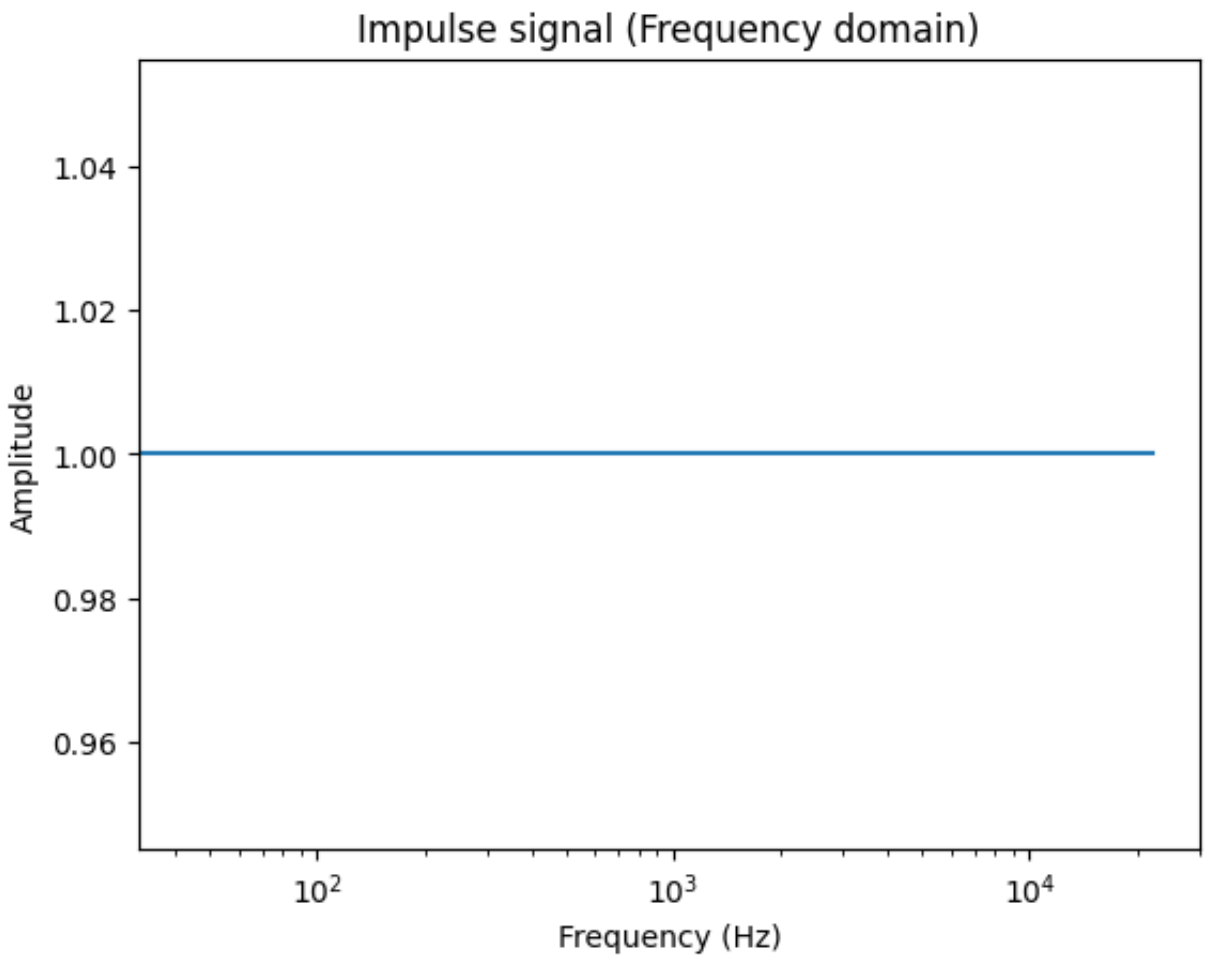


In [ ]:
```
from IPython.display import Audio

sampleRate = 44100
audio = Audio(data=impulse, rate=sampleRate)
audio
```

Out[ ]: 0:00 -0:00

Doesn't sound like much right? Just doublecheck whether it is true that it contains all the possible freqeuncies ever:

In [ ]:
```
spectrum = np.fft.fft(impulse)
x = np.linspace(0, spectrum.size * sampleRate / frameSize / 2 , spectrum.
y = np.abs(spectrum)[:spectrum.size//2]
plt.plot(x, y)
plt.xscale('log')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title('Impulse signal (Frequency domain)')
plt.show()
```

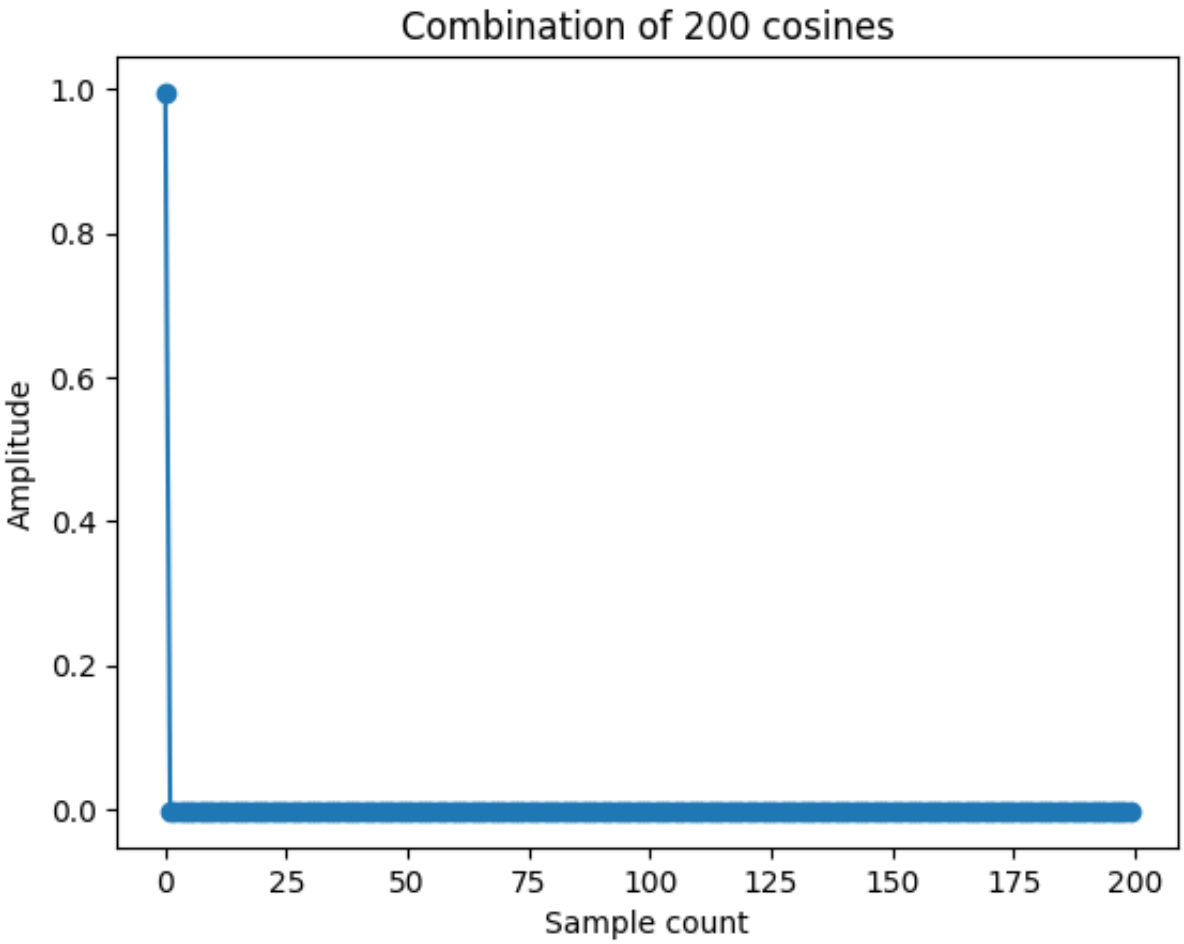


OK, now you are trying to understand why impulses are so powerful... Why do impulse signals contain all possible sinusoids? Let's try and think about it the other way around. What would a signal look like when you add all possible sinusoids together? let's sum some cosines and let's find out:

Let's take a sample rate of 400. And let's add sinusoids together starting from a sinusoid of frequency 1, all the way to a sinusoid of frequency 200 (Nyquist frequency). What do you think we'd get?

In [ ]:
```
sampleRate = 400
freq = 200

arr = np.linspace(0, freq, sampleRate)
combination = np.zeros(sampleRate)
for f in range(1, freq):
    combination += np.cos(2 * np.pi * f * arr)
combination /= freq

plt.plot(combination[:combination.size//2], 'o-')
plt.xlabel('Sample count')
plt.ylabel('Amplitude')
#plt.xlim(-0.5, 10)
plt.title("Combination of " + str(freq) + " cosines")
```

```
plt.show()
```



OK, now that's clear! You used your impulses to "knock at the doors of the other side" ... and something is happening! A light, purple mist is rising in front, and lowering, and rising, and lowering, periodically! of you... That is not a good sign! This mist clearly works on the high frequency spectrum!

Hurry up, there must be a way to filter the high frequency components of the mist!

The magical book of low pass filtering

This old scroll describes how to design a lowpass filter with a cutoff frequency ω_c and compute the discrete coefficients. Applying this filter to the magical mist will certainly work to exorcise the ghost for good!

In [ ]:
```
# Packages and adjustments to the figures

plt.rcParams["figure.figsize"] = 10,5
plt.rcParams["font.size"] = 16
# plt.rcParams.update({"text.usetex": True,"font.family": "sans-serif","f
```

Generate a magical mist (signal) and test your magic (filter)

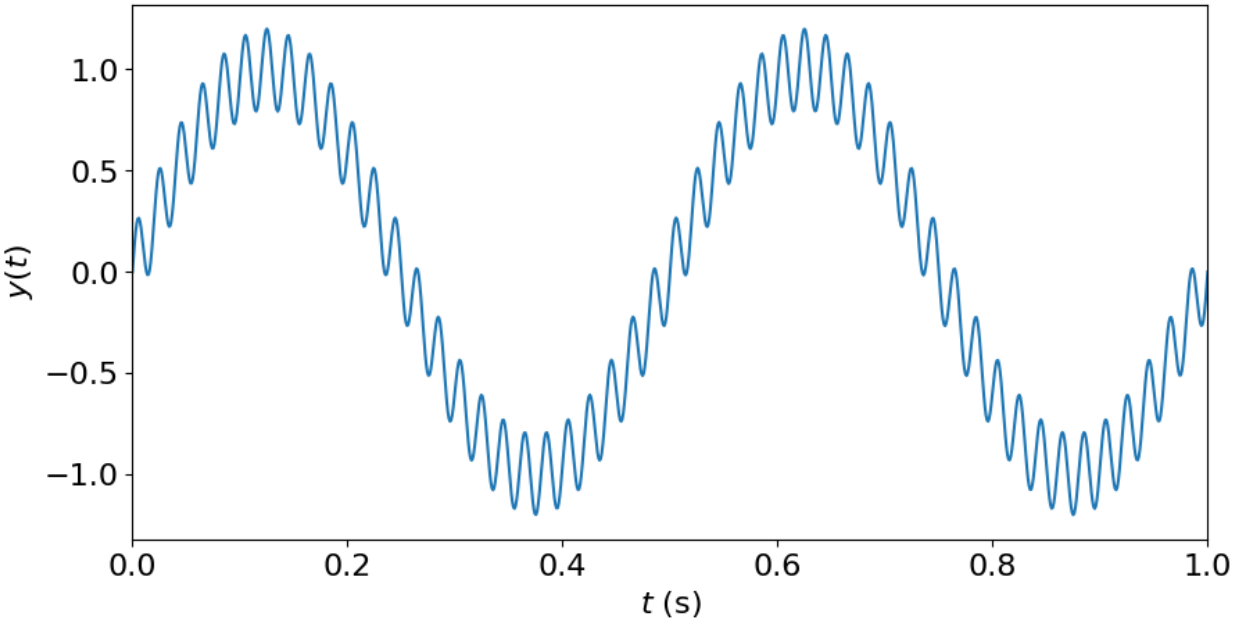
on it

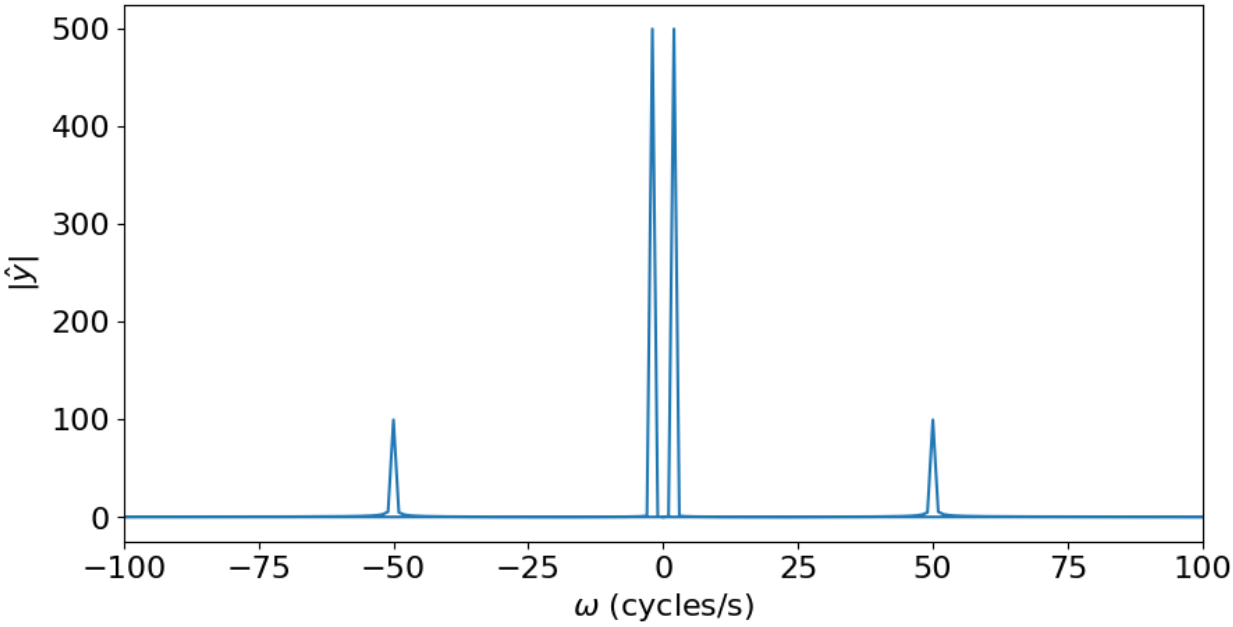
- A simple test signal $\boldsymbol{y} = \{y_i\}$ is generated with a fixed sampling frequency using the function:

$$y(t) = m_0 \sin(2\pi f_0 t) + m_1 \sin(2\pi f_1 t)$$

- The power spectrum is plotted as the magnitude of the discrete fourier transform (DFT): $|\hat{\boldsymbol{y}}|$

In [ ]:
```
# Generate a signal
samplingFreq = 1000; # sampled at 1 kHz = 1000 samples / second
tlims = [0,1]        # in seconds
signalFreq = [2,50]; # Cycles / second
signalMag = [1,0.2]; # magnitude of each sine
t = np.linspace(tlims[0],tlims[1],(tlims[1]-tlims[0])*samplingFreq)
y = signalMag[0]*np.sin(2*math.pi*signalFreq[0]*t) + signalMag[1]*np.sin(

# Compute the Fourier transform
yhat = np.fft.fft(y);
fcycles = np.fft.fftfreq(len(t),d=1.0/samplingFreq); # the frequencies in

# Plot the signal
plt.figure()
plt.plot(t,y);
plt.ylabel("$y(t)$");
plt.xlabel("$t$ (s)");
plt.xlim([min(t),max(t)]);

# Plot the power spectrum
plt.figure()
plt.plot(fcycles,np.absolute(yhat));
plt.xlim([-100,100]);
plt.xlabel("$\omega$ (cycles/s)");
plt.ylabel("$|\hat{y}|$");
```





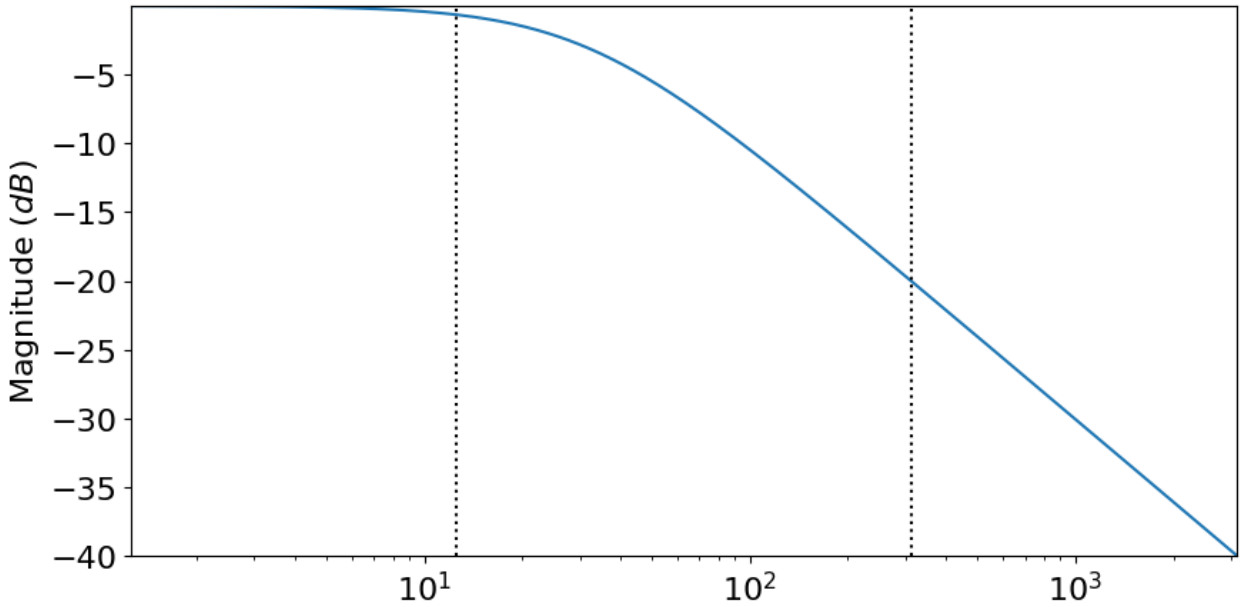
The transfer function is where the magic actually happens

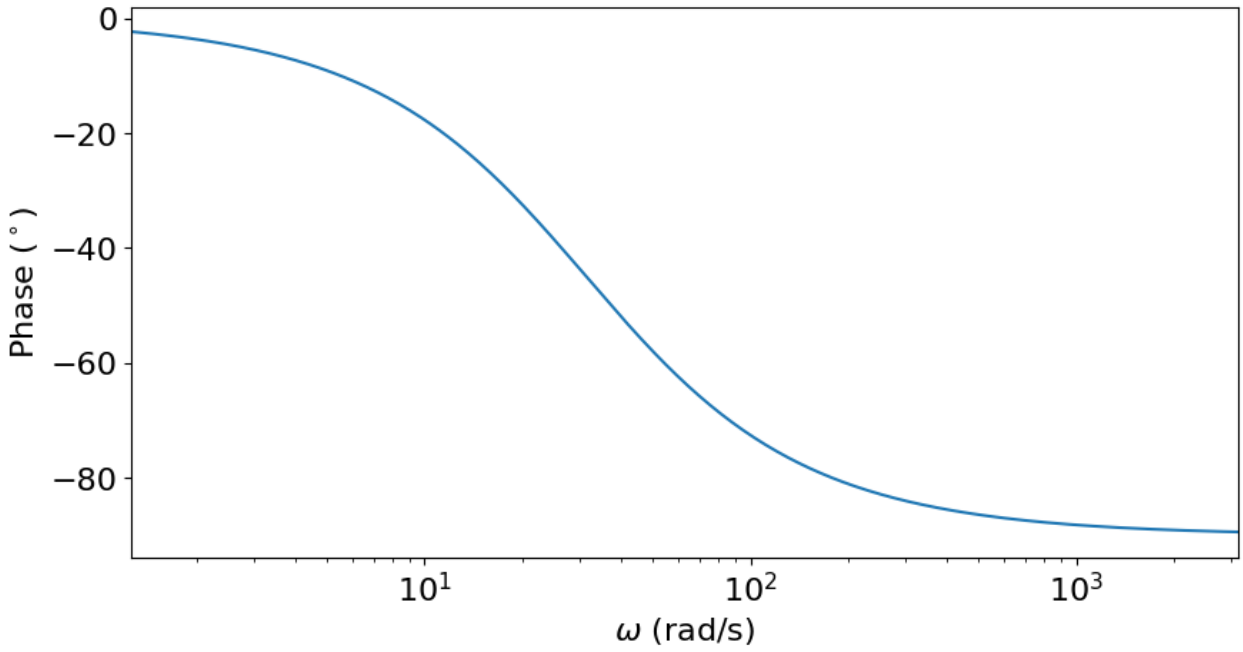
- A cutoff frequency is selected and the transfer function for the low-pass filter is computed using `signal.TransferFunction`
- The low-pass filter transfer function is

$$H(s) = \frac{\omega_0}{s + \omega_0}$$

- The Bode plot shows the frequency response of H by plotting the magnitude and phase of the frequency response
- Low frequencies are not attenuated (this is the *pass band*)
- High frequencies are attenutated (this is the *stop band*)

In [ ]:
```
# Low-pass filter
w0 = 2*np.pi*5; # pole frequency (rad/s)
num = w0        # transfer function numerator coefficients
den = [1,w0]    # transfer function denominator coefficients
lowPass = signal.TransferFunction(num,den) # Transfer function

# Generate the bode plot
w = np.logspace( np.log10(min(signalFreq)*2*np.pi/10), np.log10(max(signa
w, mag, phase = signal.bode(lowPass,w)

# Magnitude plot
plt.figure()
plt.semilogx(w, mag)
for sf in signalFreq:
    plt.semilogx([sf*2*np.pi,sf*2*np.pi],[min(mag),max(mag)],'k:')
plt.ylabel("Magnitude ($dB$)")
plt.xlim([min(w),max(w)])
plt.ylim([min(mag),max(mag)])

# Phase plot
plt.figure()
plt.semilogx(w, phase)  # Bode phase plot
plt.ylabel("Phase ($^\circ$)")
plt.xlabel("$\omega$ (rad/s)")
plt.xlim([min(w),max(w)])
plt.show()
```





we are talking about a digital ghost, that lives in discrete time, so we need some adjustments

To implement the low-pass filter in the digital, discretized domain (and when you will do that on hardware), you need to compute the discrete transfer function using the signal's sampling frequency.

- The time step is $\Delta t = 1/f_s$
- Computing the discrete transfer function using Tustin's method, set $s = \frac{2}{\Delta t}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)$, so

$$H(z) = \frac{\omega_0}{\frac{2}{\Delta t}\frac{1-z^{-1}}{1+z^{-1}} + \omega_0} = \frac{\Delta t \omega_0 (z+1)}{(\Delta t \omega_0 + 2)z + \Delta t \omega_0 - 2}$$

- You don't have to compute it by hand. The `to_discrete` method is used to compute the bilinear transform (Tustin's method)

In [ ]:
```
dt = 1.0/samplingFreq;
discreteLowPass = lowPass.to_discrete(dt,method='gbt',alpha=0.5)
print(discreteLowPass)
```

```
TransferFunctionDiscrete(
array([0.01546504, 0.01546504]),
array([ 1.        , -0.96906992]),
dt: 0.001
)
```

Filter coefficients

We want to find the filter coefficients for the discrete update:

$$y[n] = a_1 y[n-1] + a_2 y[n-2] + \ldots + b_0 x[n] + b_1 x[n-1] + \ldots$$

The coefficients can be taken directly from the discrete transfer function of the filter in the form:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots}{1 - a_1 z^{-1} - a_2 z^{-2} + \ldots}$$

(This is a result of taking the Z-transform which is not shown here)

Compare this to a transfer function with coefficients
`num = [b_0, b_1, b_2]`
`den = [1, a_1, a_2]`
is

$$H(z) = \frac{b_0 z^2 + b_1 z + b_2}{z^2 + a_1 z + a_2}$$

which is equivalent to

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

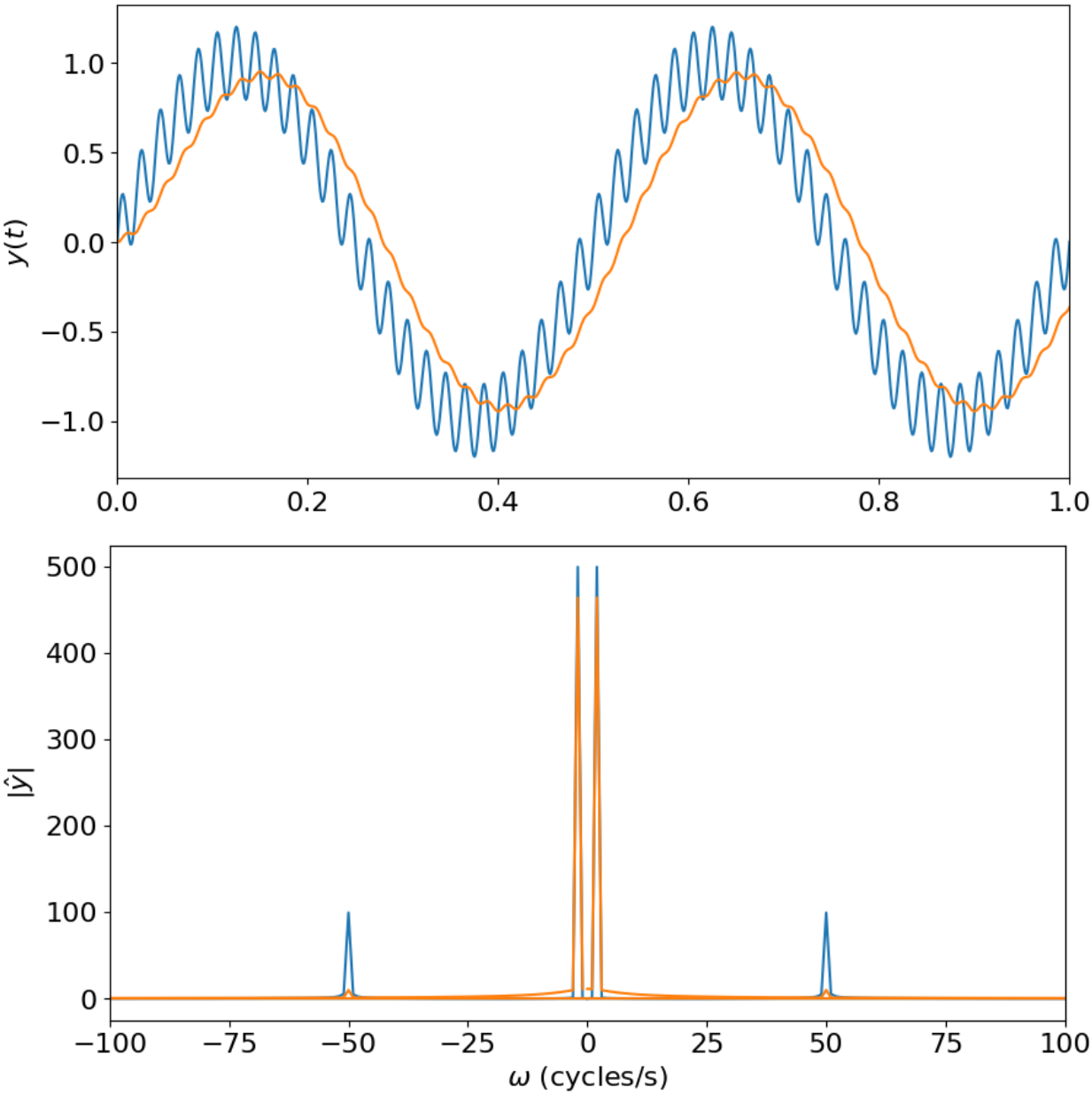
So you can take the coefficients in the same order that they are defined in the numerator and denominator of the transfer function object. The only difference is that the **coefficients in the denominator need a negative sign**.

- To filter the signal, apply the filter using the discrete update
- The filtered signal and filtered signal power spectrum are plotted alongside the unfiltered signal

In [ ]:
```
# The coefficients from the discrete form of the filter transfer function
b = discreteLowPass.num;
a = -discreteLowPass.den;
print("Filter coefficients b_i: " + str(b))
print("Filter coefficients a_i: " + str(a[1:]))

# Filter the signal
yfilt = np.zeros(len(y));
for i in range(3,len(y)):
    yfilt[i] = a[1]*yfilt[i-1] + b[0]*y[i] + b[1]*y[i-1];

# Plot the signal
plt.figure()
plt.plot(t,y);
plt.plot(t,yfilt);
plt.ylabel("$y(t)$")
```

```
plt.xlim([min(t),max(t)]);

# Generate Fourier transform
yfilthat = np.fft.fft(yfilt)
fcycles = np.fft.fftfreq(len(t),d=1.0/samplingFreq)

plt.figure()
plt.plot(fcycles,np.absolute(yhat));
plt.plot(fcycles,np.absolute(yfilthat));
plt.xlim([-100,100]);
plt.xlabel("$\omega$ (cycles/s)");
plt.ylabel("$|\hat{y}|$");
```

```
Filter coefficients b_i: [0.01546504 0.01546504]
Filter coefficients a_i: [0.96906992]
```



OK, our filters works!!! the mist dissipates and the evil spirit is exprcised!
HUZZAH!!!!!!


Happy
Halloween