Homework 3 - Question 1 - Luke Arend

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.fft import fft, fftshift
from hw3.unknownSystemsAll import unknown_systems as us
neuron1 = us.unknown_system_1
neuron2 = us.unknown_system_2
neuron3 = us.unknown_system_3
```

You are trying to experimentally characterize three auditory neurons, in terms of their responses to sounds. For purposes of this problem, the responses of these neurons are embodied in compiled matlab functions `unknownSystemX.p` with $X = 1, 2, 3$. If you are using Python, import the unknown systems module from the obfuscated Python file. Each takes an input column vector of length $N = 64$ whose elements represent sound pressure over time. In Python the response of each is a column vector (of the same length) representing the mean spike count over time. For each neuron,

a)

"Kick the tires" by measuring the response to an impulse in the first position of an input vector. Check that the system is consistent with shift-invariance by comparing this to the response to an impulse at positions $n = 2, 4, 8$. Check that the system is consistent with linearity by asserting that the response to any combination of two impulses is equal to the sum of their individual responses. Also examine respones to impulses at different n to determine how the system handles inputs near the boundary (i.e., whether the system does circular boundary-handling). Describe your findings.

In [2]:
```
def unit_impulse(idx=0):
    """ Produce a length-64 zero vector with a 1 at position `idx`. """
    signal = np.zeros(64)
    signal[idx] = 1
    return signal
```

In [3]:
```
def plot_impulse_response(system, positions=(0, 1, 3, 7)):
    """ Plot the system's response to up to four unit impulses. """
    fig, axs = plt.subplots(2, 2)
    for i, pos in enumerate(positions):
        plt.sca(axs.flat[i])
        impulse = unit_impulse(idx=pos)
        response = pd.Series(system(impulse))
        plt.title(f'Impulse(t={pos})')
        sns.lineplot(impulse, label='Input')
```

```
        sns.lineplot(response, label='Response')
        plt.legend()
        if i % 2 == 0:
            plt.ylabel('Mean spike count')
        if i >= 2:
            plt.xlabel('Time step')
    plt.tight_layout()
```

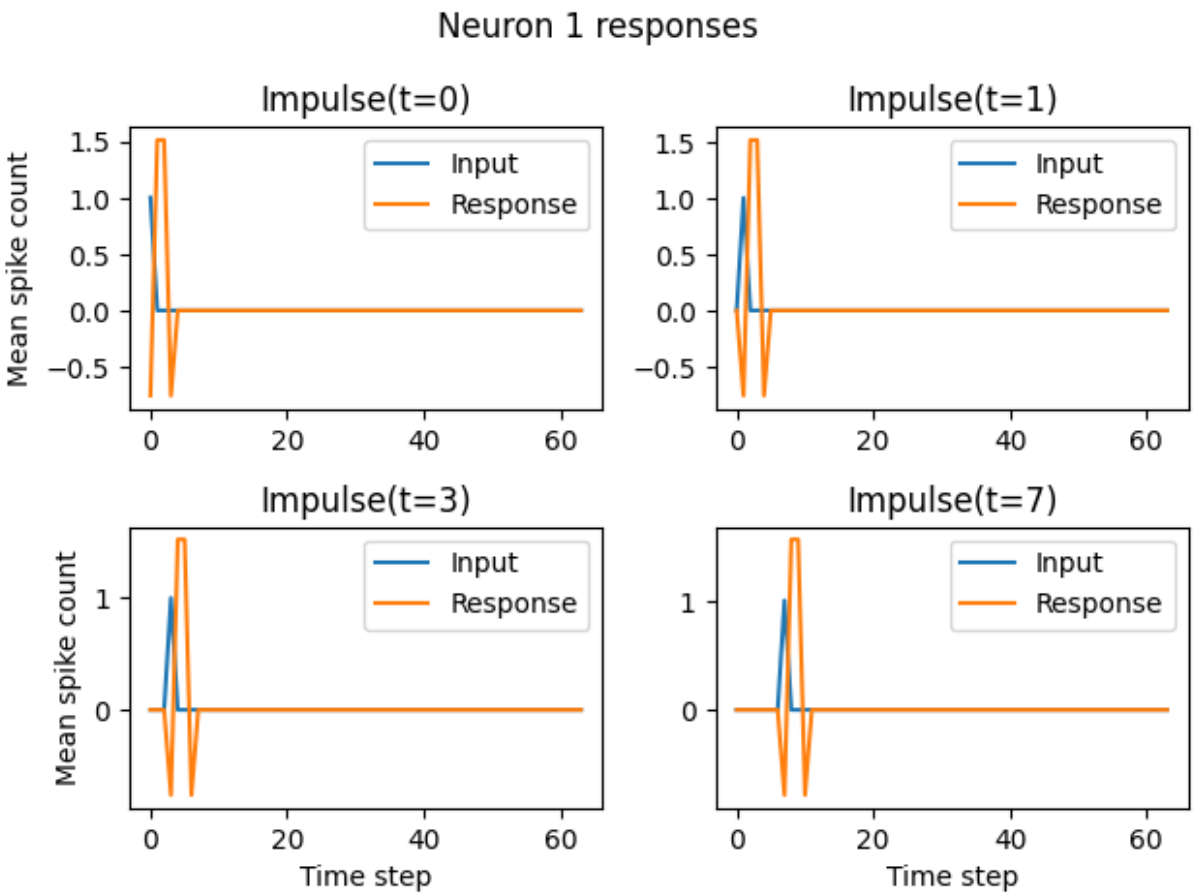
In [4]:
```
def obeys_shift_invariance(system):
    """ Return True if the system's response to a unit impulse at time t
    equals its response to the unit impulse at time 0 shifted by t. """
    response_0 = system(unit_impulse(0))
    for i in range(1, 64):
        response = system(unit_impulse(idx=i))
        shifted = np.concatenate([response[i:], response[:i]])
        if not np.allclose(shifted, response_0):
            return False
    return True
```

In [5]:
```
response = neuron1(unit_impulse(idx=3))
```

In [6]:
```
def obeys_linear_superposition(system):
    """ Return True if the system's response to a sum of any two
    impulses equals the sum of its responses to each impulse. """
    for i in range(64):
        for j in range(64):
            signal1 = unit_impulse(i)
            signal2 = unit_impulse(j)
            response1 = system(signal1)
            response2 = system(signal2)
            response3 = system(signal1 + signal2)
            if not np.allclose(response1 + response2, response3):
                return False
    return True
```

Neuron 1

In [7]:
```
plot_impulse_response(neuron1, positions=[0, 1, 3, 7])
plt.suptitle(f'Neuron 1 responses')
plt.tight_layout()
```



Neuron 1 might seem shift-invariant if you check its responses to step functions in the first few time steps. But compare unit functions at `t=0` and `t=32`:

In [8]:
```
response1 = neuron1(unit_impulse(0))
response2 = neuron1(unit_impulse(32))
np.max(response1), np.max(response2)
```

Out[8]:
```
(1.5099980968844182, 2.4000000000000004)
```

In [9]:
```
obeys_shift_invariance(neuron1)
```

Out[9]:
```
False
```

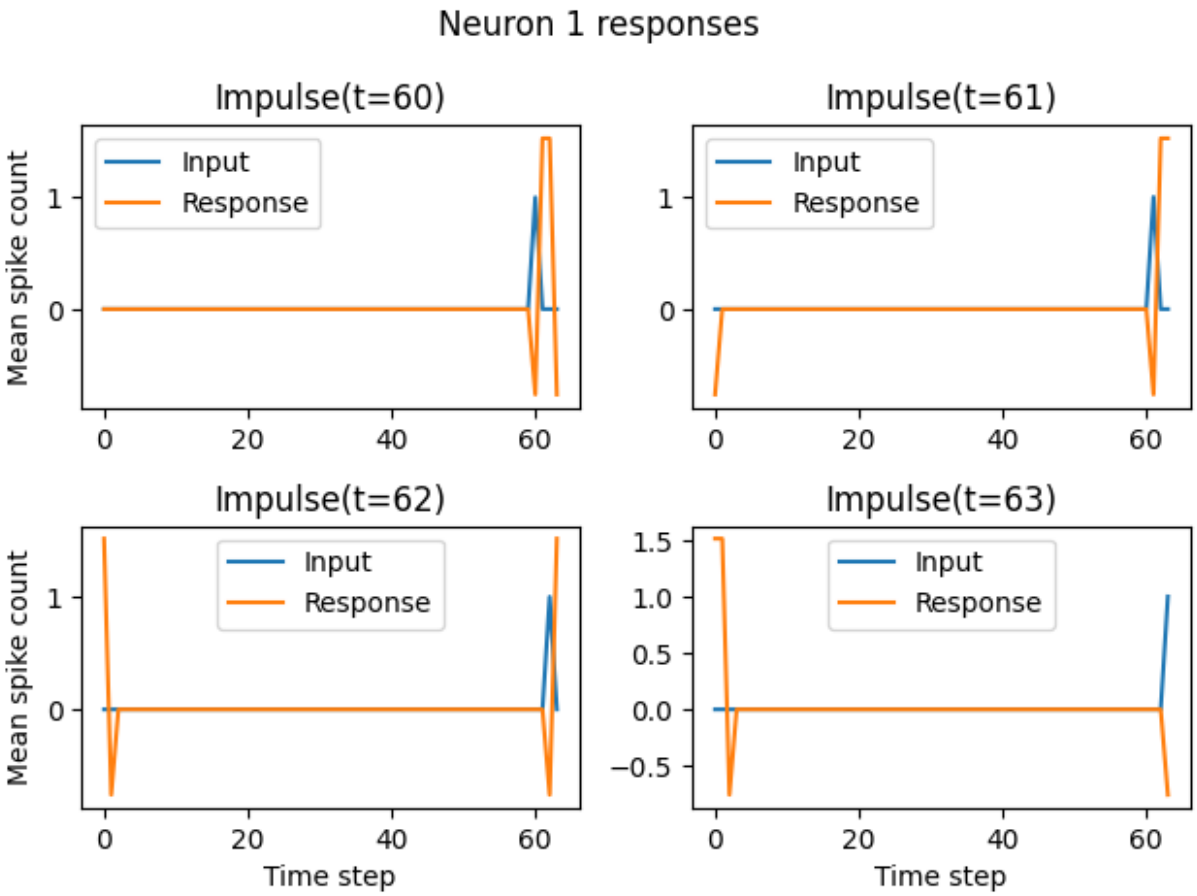
The responses to an impulse function changes amplitude as we shift the input in time. So Neuron 1 **is not shift-invariant**.

In [10]:
```
obeys_linear_superposition(neuron1)
```

Out[10]:
```
True
```

Neuron 1 **is linear** in that its response obeys linear superposition.

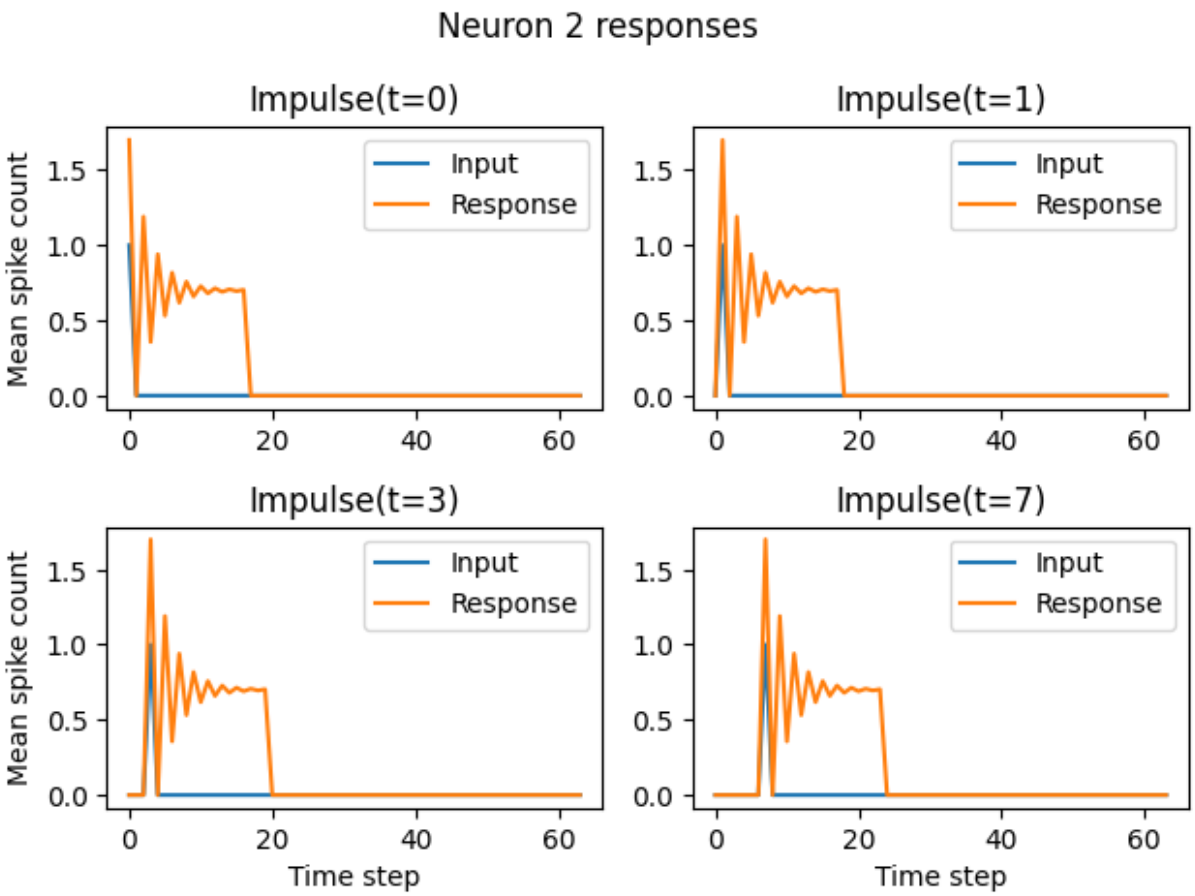
In [11]:
```
plot_impulse_response(neuron1, positions=[60, 61, 62, 63])
plt.suptitle(f'Neuron 1 responses')
plt.tight_layout()
```



The response to impulses at timesteps `t = [60, 61, 62, 63]` shows that Neuron 1 **does circular boundary-handling** (it views the end of its input range as wrapping around to connect with the beginning).

Neuron 2

In [12]:
```
plot_impulse_response(neuron2, positions=[0, 1, 3, 7])
plt.suptitle(f'Neuron 2 responses')
plt.tight_layout()
```



In [13]:
```
obeys_shift_invariance(neuron2)
```

Out[13]: True

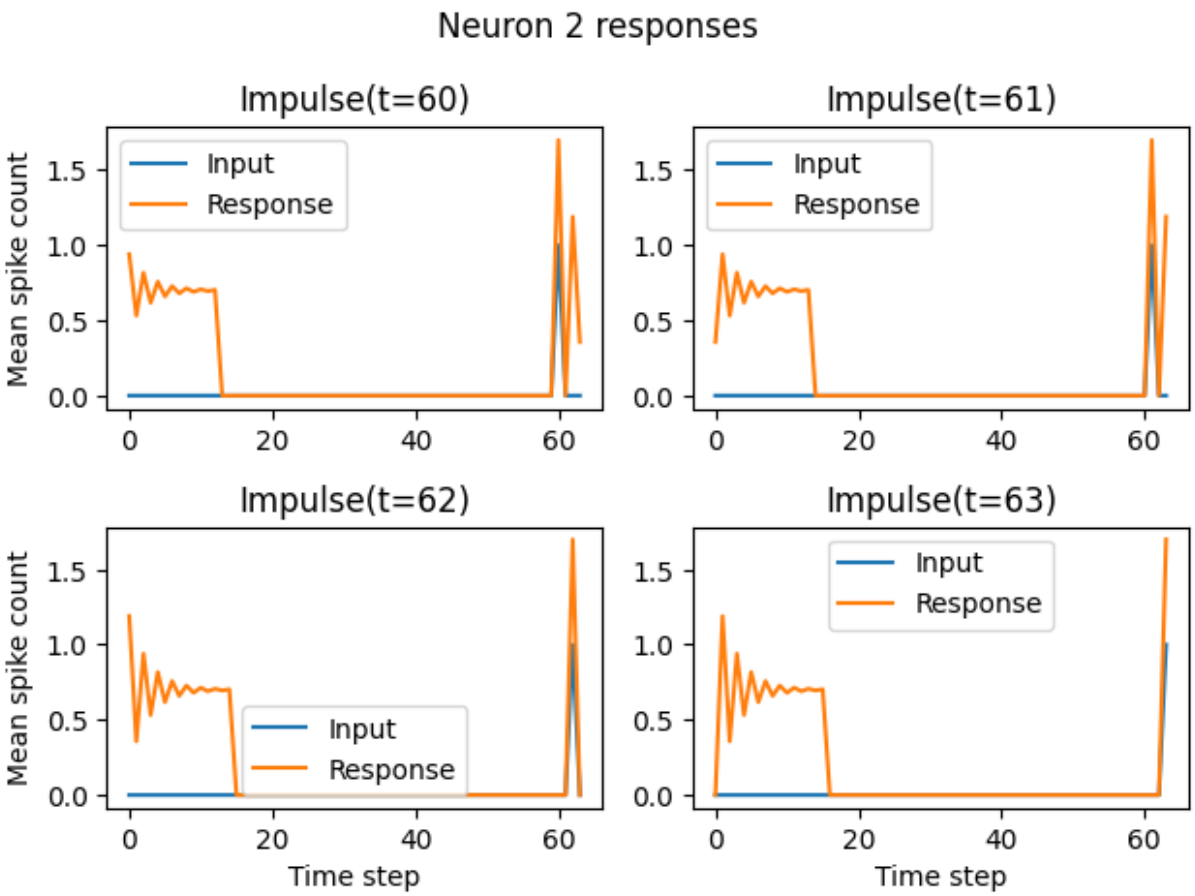
Neuron 2 **may be shift-invariant**, since its response to an impulse function only shifts (and does not change otherwise) as we shift the input in time.

In [14]:
```
obeys_linear_superposition(neuron2)
```

Out[14]: True

Neuron 2 **may be linear**, since its response to impulses obeys linear superposition.

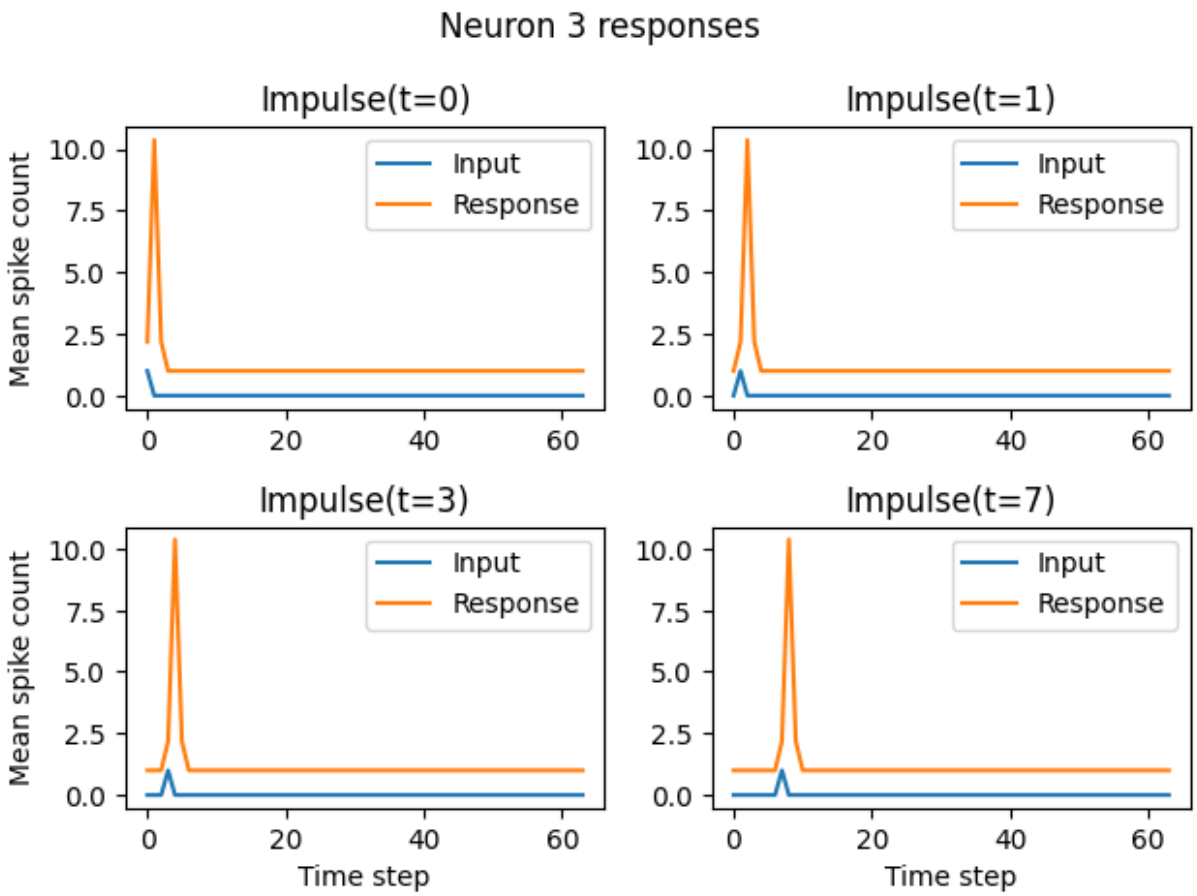
In [15]:
```
plot_impulse_response(neuron2, positions=[60, 61, 62, 63])
plt.suptitle(f'Neuron 2 responses')
plt.tight_layout()
```



The response to impulses at timesteps `t = [60, 61, 62, 63]` shows that Neuron 2 **does circular boundary-handling**.

Neuron 3

In [16]:
```
plot_impulse_response(neuron3, positions=[0, 1, 3, 7])
plt.suptitle(f'Neuron 3 responses')
plt.tight_layout()
```



In [17]:
```
obeys_shift_invariance(neuron3)
```

Out[17]: True

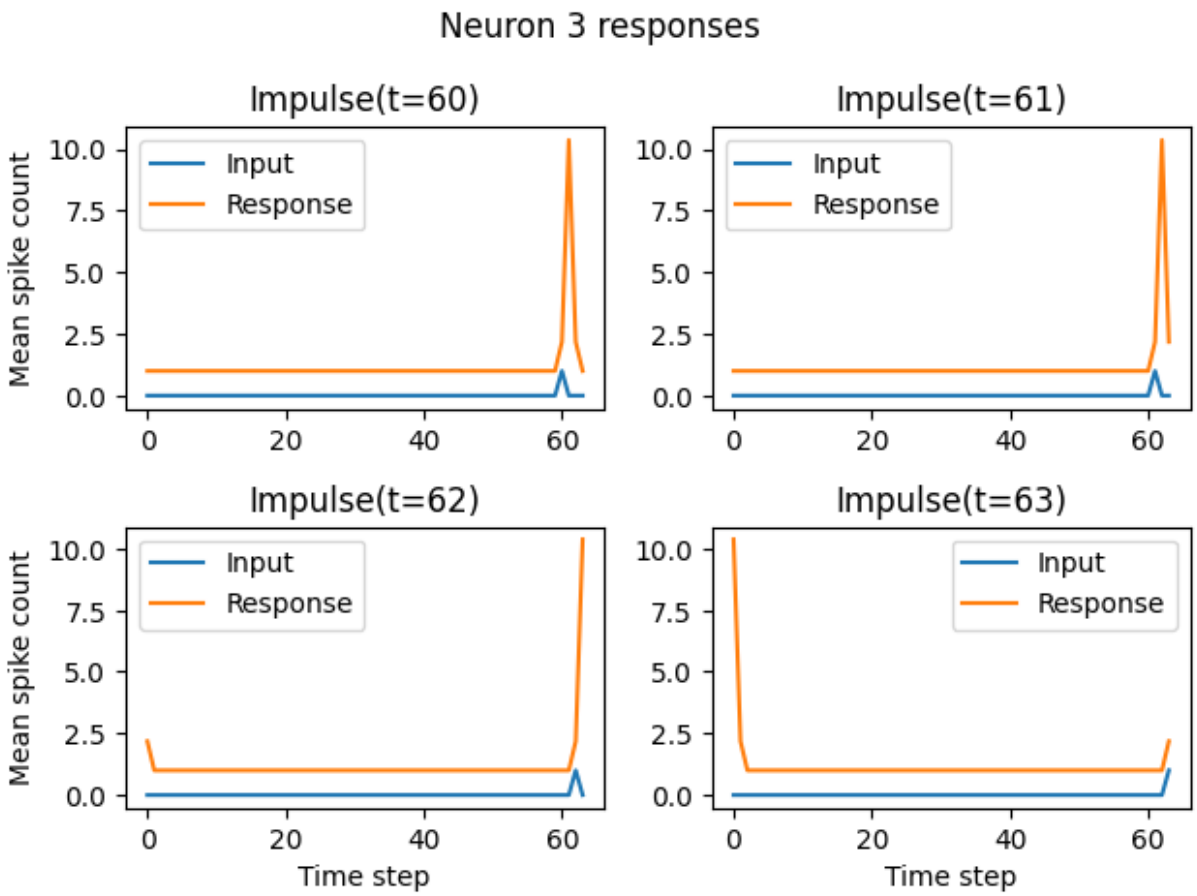
Neuron 3 **may be shift-invariant** in that its response to an impulse function merely shifts as we shift the input in time.

In [18]:
```
obeys_linear_superposition(neuron3)
```

Out[18]: False

Neuron 3 **is not linear** as its response does not obey linear superposition.

In [19]:
```
plot_impulse_response(neuron3, positions=[60, 61, 62, 63])
plt.suptitle(f'Neuron 3 responses')
plt.tight_layout()
```



The response to impulses at timesteps `t = [60, 61, 62, 63]` shows that Neuron 3 **does circular boundary-handling**.

b)

If the previous tests succeeded, examine the response of the system to sinusoids with frequencies $2\pi/N, 4\pi/N, 8\pi/N, 16\pi/N$, and random phases, and check whether the outputs are sinusoids of the same frequency (i.e., verify that the output vector lies completely in the subspace containing all the sinusoids of that frequency). [Note: make all elements of the the input stimuli positive, by adding one to each sinusoid. The responses will then also be positive (mean spike counts).]

In [20]:
```
def make_sinusoid(ncycles):
    x = np.arange(64) / 64
    return np.sin(ncycles * 2 * np.pi * x + np.random.rand())

def plot_responses(inputs, responses, fourier=False):
    """ Plot up to four input-response pairs. """
    fig, axs = plt.subplots(2, 2)
    for i, (signal, response) in enumerate(zip(inputs, responses)):
        plt.sca(axs.flat[i])
        x = np.arange(-32, 32) if fourier else np.arange(64)
        sns.lineplot(x=x, y=signal, label='Input')
```

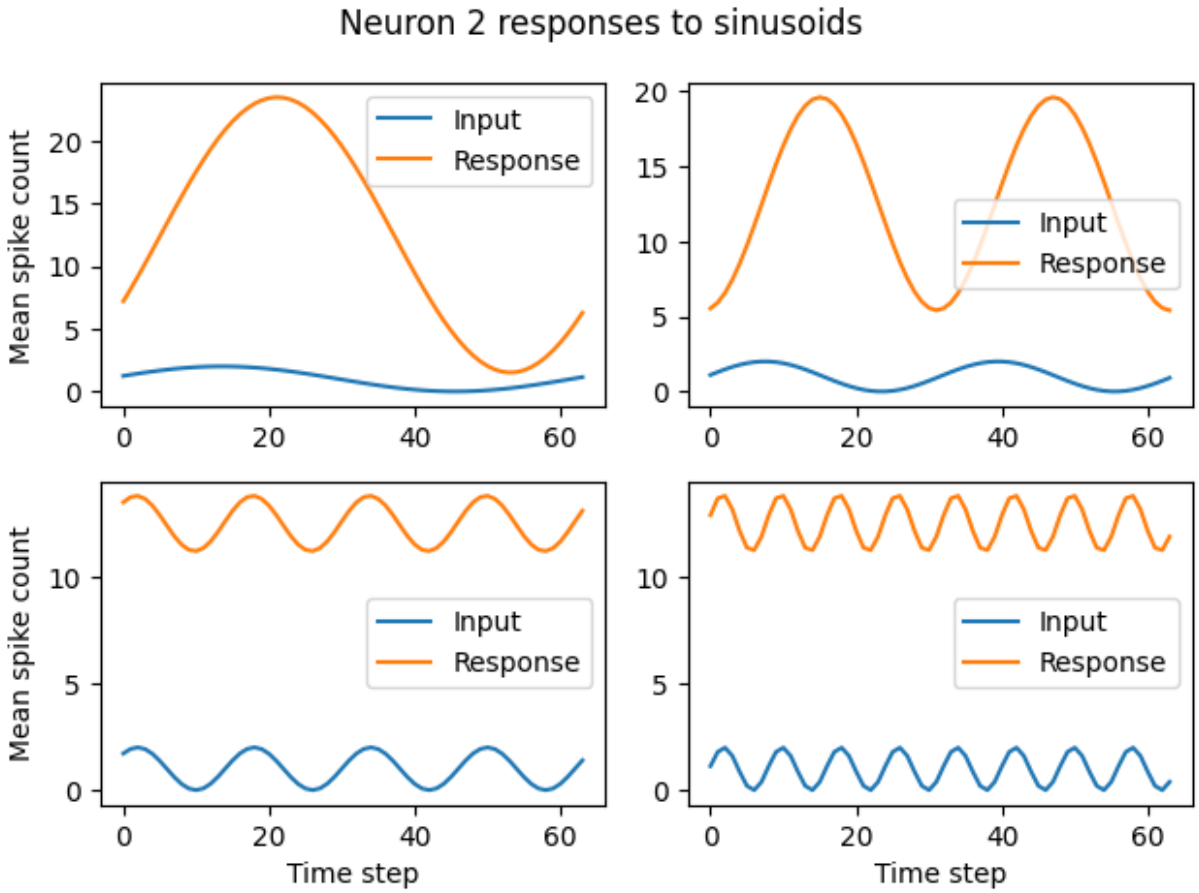
```
        sns.lineplot(x=x, y=response, label='Response')
        plt.legend()
        if i % 2 == 0:
            plt.ylabel('Amplitude' if fourier else 'Mean spike count')
        if i >= 2:
            plt.xlabel('Frequency' if fourier else 'Time step')
    plt.tight_layout()

ncycles = [1, 2, 4, 8]
sinusoids = [make_sinusoid(n) + 1.0 for n in ncycles]
```

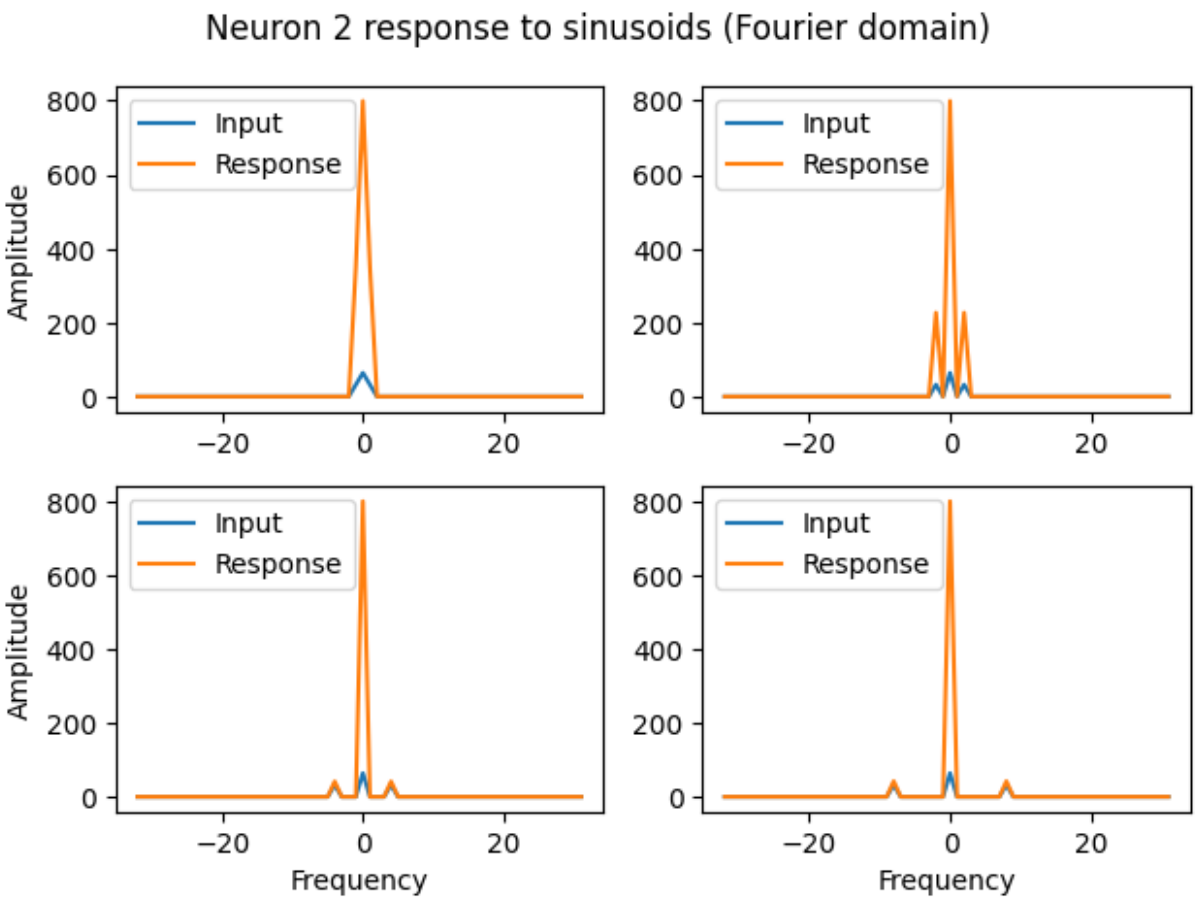
The previous tests indicated that **Neuron 2** was linear and shift-invariant.

Neuron 2

In [21]:
```
responses = [neuron2(s) for s in sinusoids]
plot_responses(sinusoids, responses)
plt.suptitle(f'Neuron 2 responses to sinusoids')
plt.tight_layout()
```



In [22]:
```
inputs_fft = [fftshift(fft(s)) for s in sinusoids]
responses_fft = [fftshift(fft(neuron2(s))) for s in sinusoids]
plot_responses(np.abs(inputs_fft), np.abs(responses_fft), fourier=True)
plt.suptitle(f'Neuron 2 response to sinusoids (Fourier domain)')
plt.tight_layout()
```



In the Fourier domain, we see that the response to each sinusoid contains only the frequency present in the input signal (as well as a DC bias at frequency 0 due to adding a constant to the input sinusoids). This means that each output vector lies completely in the subspace spanned by sinusoids with the input frequency.

c)

If the previous tests succeeded, verify that the change in amplitude and phase from input to output is predicted by the amplitude (`abs`) and phase (`angle`) of the corresponding terms of the Fourier transform of the impulse response. If not, explain which property (linearity, or shift-invariance, or both) seems to be violated by the system. If so, does the combination of all of your tests guarantee that the system is linear and shift-invariant? What set of tests would provide such a guarantee?

Neuron 1

Neuron 1 may be linear but is not shift-invariant. It obeys linear superposition for impulse functions but does not obey shift-invariance since the amplitude of a response depends on the impulse's position in time.

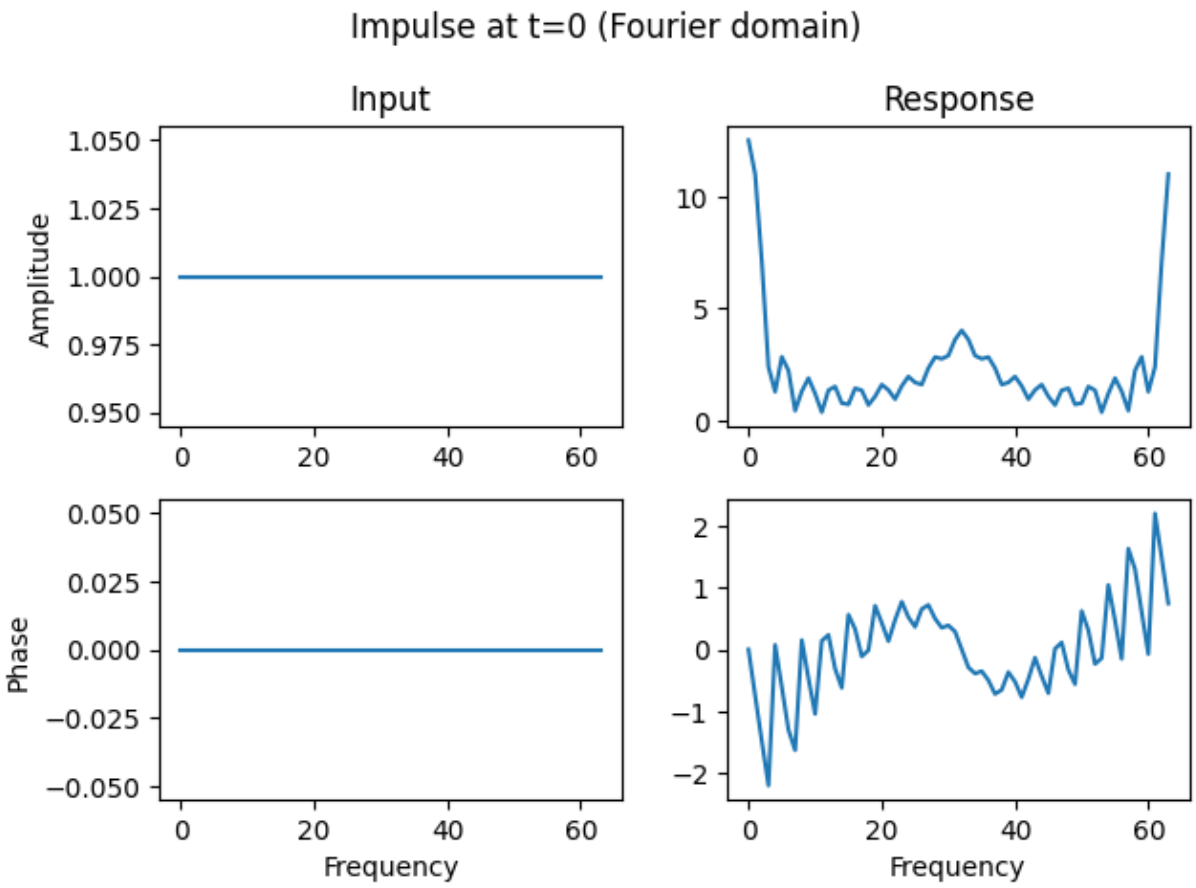
Neuron 2

Neuron 2 may be shift-invariant and may be linear.

First let's inspect the Fourier spectra (both amplitude and phase) for an impulse signal at `t=0` and its response.

In [23]:
```
signal = unit_impulse(0)
response = neuron2(signal)
signal_fft = fft(signal)
response_fft = fft(response)

fig, axs = plt.subplots(2, 2)
plt.suptitle('Impulse at t=0 (Fourier domain)')
plt.sca(axs[0][0])
plt.title('Input')
sns.lineplot(np.abs(signal_fft))
plt.ylabel('Amplitude')
plt.sca(axs[0][1])
plt.title('Response')
sns.lineplot(np.abs(response_fft))
plt.sca(axs[1][0])
sns.lineplot(np.angle(signal_fft))
plt.ylabel('Phase')
plt.xlabel('Frequency')
plt.sca(axs[1][1])
sns.lineplot(np.angle(response_fft))
plt.xlabel('Frequency')
plt.tight_layout()
```



Looking at indexes 1, 2, 4 and 8 in `signal_fft` and `response_fft` tells us the amplitude/phase of real components with 1, 2, 4 and 8 cycles in the input and response, respectively. Amplitude is given by the `abs` and phase by the `angle` of the complex number. So for the n-cycle frequency, we can get the amplitude change `np.abs(response_fft[n]) / np.abs(signal_fft[n])` and the phase shift as `np.angle(response_fft[n]) / np.angle(signal_fft[n])`.

We can compare these to the amplitude and phase differences measured from experiments on the system.

In [24]:
```
def predict_amplitude_change(system, signal, ncycles):
    response = system(signal)
    signal_amp = np.abs(fft(signal))[ncycles]
    response_amp = np.abs(fft(response))[ncycles]
    return response_amp / signal_amp

def predict_phase_shift(system, signal, ncycles):
    response = system(signal)
    signal_phase = np.angle(fft(signal))[ncycles]
    response_phase = np.angle(fft(response))[ncycles]
    return signal_phase - response_phase

def measure_amplitude_change(signal, response):
    return np.max(response) / np.max(signal)

def measure_phase_shift(signal, response):
```

```
    signal -= np.mean(signal)
    response -= np.mean(response)
    signal /= np.max(signal)
    response /= np.max(response)
    val = 2 * np.mean(signal * response)
    if val > 1:
        val = val - 2
        return np.arccos(val)
    return np.arccos(val)
```

In [25]:
```
df = pd.DataFrame()
responses = [neuron2(s) for s in sinusoids]
for n, signal, response in zip(ncycles, sinusoids, responses):
    row = f'{n}-cycle sinusoid'
    df.loc[row, 'Amplitude scaling (predicted)'] = predict_amplitude_change(
    df.loc[row, 'Amplitude scaling (measured)'] = measure_amplitude_change(s
    df.loc[row, 'Phase shift (predicted)'] = predict_phase_shift(neuron2, si
    df.loc[row, 'Phase shift (measured)'] = measure_phase_shift(signal, resp
df
```

Out[25]:

	Amplitude scaling (predicted)	Amplitude scaling (measured)	Phase shift (predicted)	Phase shift (measured)
1-cycle sinusoid	10.979091	11.740405	0.746050	0.744782
2-cycle sinusoid	7.083784	9.807816	1.487737	1.487369
4-cycle sinusoid	1.296524	6.890914	3.067542	3.096255
8-cycle sinusoid	1.319229	6.907909	2.990225	2.864089

We find very good agreement between predictions and experiment.

Our tests show that Neuron 2:

1. is invariant to shifts (for impulse functions),
2. obeys linear superposition (for impulse functions), and
3. respects the amplitude and phase scaling predicted by the Fourier transform (for impulse functions).

These tests **do not guarantee** that the system is linear nor shift-invariant in general. I can imagine the "black-box" system containing a rule that produces a nonlinear and/or non-shift-invariant response only for very particular input vectors which simply aren't tested here. The only way to conclusively demonstrate that the system is linear and shift-invariant is to test it on all possible inputs.

Neuron 3

Neuron 3 may be shift-invariant but is not linear. It obeys shift invariance for all impulse functions individually. However, we cannot say it is shift-invariant for general inputs (i.e. sums of impulse functions). Neuron 3 does not obey linear superposition for sums of impulse, so we have no guarantee that it will handle the individual components of the input in a linear way.

In [ ]:

Homework 3 - Question 2 - Luke Arend

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.fft import fft, fftshift
from scipy.signal import convolve
```

The response properties of neurons in primary visual cortex (area V1) are often described using linear filters. We'll examine a one-dimensional cross-section of the most common choice, known as a "Gabor filter" (named after Electrical Engineer/Physicist Denis Gabor, who developed it in 1946 for use in signal processing).

a)

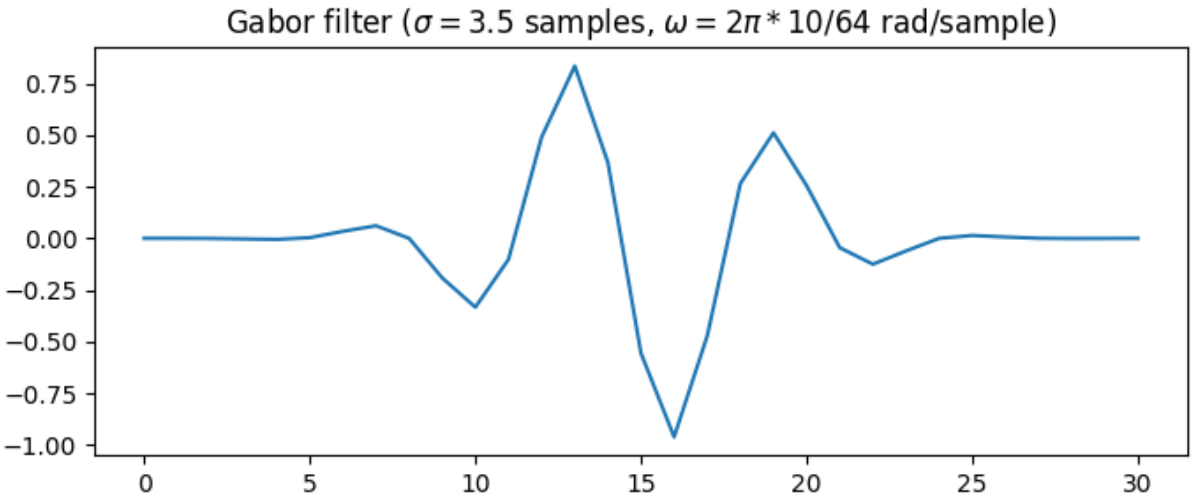
In [2]:
```
def make_gabor(sigma, omega, nsamples):
    center = nsamples // 2
    n = np.arange(nsamples) - center
    gaussian = np.exp(-(n ** 2) / (2 * sigma ** 2))

    n = np.arange(nsamples)
    sinusoid = np.cos(omega * n)
    return gaussian * sinusoid

def plot_fft(x, title=None, ax=None):
    if ax is None:
        fig, ax = plt.subplots(figsize=(8, 3))
    plt.sca(ax)
    plt.title(title)
    plt.plot(fftshift(np.abs(x)))
```

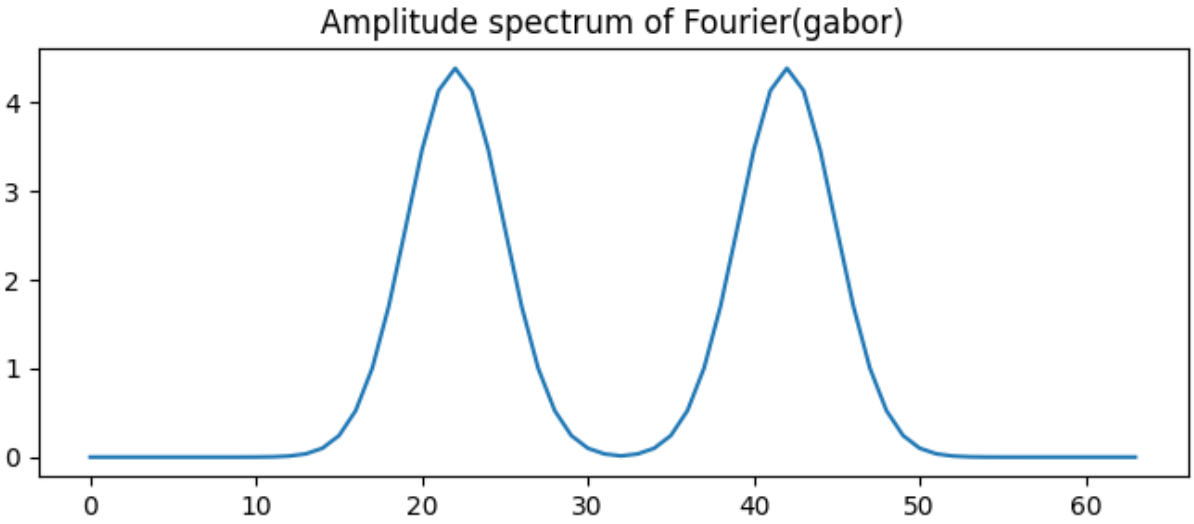
Create a one-dimensional linear filter that is a product of a Gaussian and a sinusoid, $e^{-\frac{n^2}{2\sigma^2}} \cos \omega n$, with parameters $\sigma = 3.5$ samples and $\omega = 2\pi * 10/64$ radians/sample. The filter should contain 31 samples, and the Gaussian should be centered on the middle (16th) sample. Plot the filter to verify that it looks like what you'd expect.

In [3]:
```
sigma = 3.5
omega = 2 * np.pi * 10 / 64
nsamples = 31
gabor = make_gabor(sigma, omega, nsamples)
plt.subplots(figsize=(8, 3))
plt.title('Gabor filter ($\sigma = 3.5$ samples, $\omega = 2\pi * 10/64$ rac
plt.plot(gabor)
plt.show()
```



Plot the amplitude of the Fourier transform of this filter, sampled at 64 locations (MATLAB's `fft` function takes an optional additional argument). What kind of filter is this?

In [4]:
```
gabor_fft = fft(gabor, n=64)
plot_fft(gabor_fft, title='Amplitude spectrum of Fourier(gabor)')
```



This is a bandpass filter. It passes frequencies in a band centered on ω and rejects others. The width of the band is related to σ.

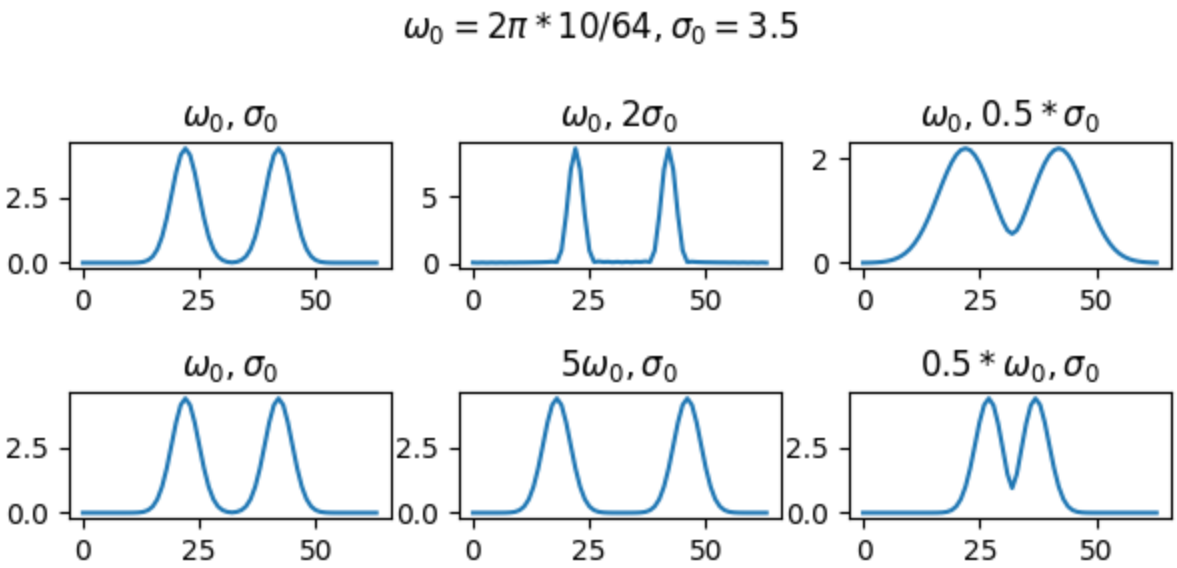
Why does it have this shape, and how is the shape related to the choice of parameters (σ, ω)?

The input to the Fourier transform is $x \odot y$ where x is a Gaussian and y is a sinusoid. The Gaussian depends on σ and the sinusoid depends on ω. Multiplying x and y pointwise in the input domain maps to convolving x with y in the output of the Fourier transform. In the Fourier domain, x looks like a pair of delta functions. The distance between them is proportional to the frequency (higher ω pushes the delta functions outwards). y in the Fourier domain looks like a Gaussian. In the limit of $\sigma \to 0$ the input

Gaussian becomes a delta function and the Fourier amplitude spectrum is a uniform distribution. As $\sigma \to \infty$ the input Gaussian becomes a uniform distribution whose Fourier amplitude spectrum is a delta function at 0.

Specifically, how does the Fourier amplitude change if you alter each of these parameters?

In [5]:
```
fig, axs = plt.subplots(2, 3, figsize=(6, 3))
plt.suptitle('$\omega_0 = 2\pi * 10/64, \sigma_0 = 3.5$')
x = fft(make_gabor(sigma, omega, nsamples), n=64)
plot_fft(x, '$\omega_0, \sigma_0$', ax=axs[0][0])
x = fft(make_gabor(2 * sigma, omega, nsamples), n=64)
plot_fft(x, '$\omega_0, 2\sigma_0$', ax=axs[0][1])
x = fft(make_gabor(sigma / 2, omega, nsamples), n=64)
plot_fft(x, '$\omega_0, 0.5 * \sigma_0$', ax=axs[0][2])
x = fft(make_gabor(sigma, omega, nsamples), n=64)
plot_fft(x, '$\omega_0, \sigma_0$', ax=axs[1][0])
x = fft(make_gabor(sigma, 5 * omega, nsamples), n=64)
plot_fft(x, '$5\omega_0, \sigma_0$', ax=axs[1][1])
x = fft(make_gabor(sigma, 0.5 * omega, nsamples), n=64)
plot_fft(x, '$0.5*\omega_0, \sigma_0$', ax=axs[1][2])
plt.tight_layout()
```



The output is two delta functions convolved with a Gaussian, which looks like two Gaussians separated by equal distance from the origin. ω controls the distance between Gaussians. As ω increases they grow further apart. σ controls how wide the peaks are. As σ grows their variance shrinks and they become delta functions as $\sigma \to \infty$.

b)

If you were to convolve this filter with sinusoids of different frequencies, which of them would produce a response with the largest amplitude? Obtain this answer by reasoning

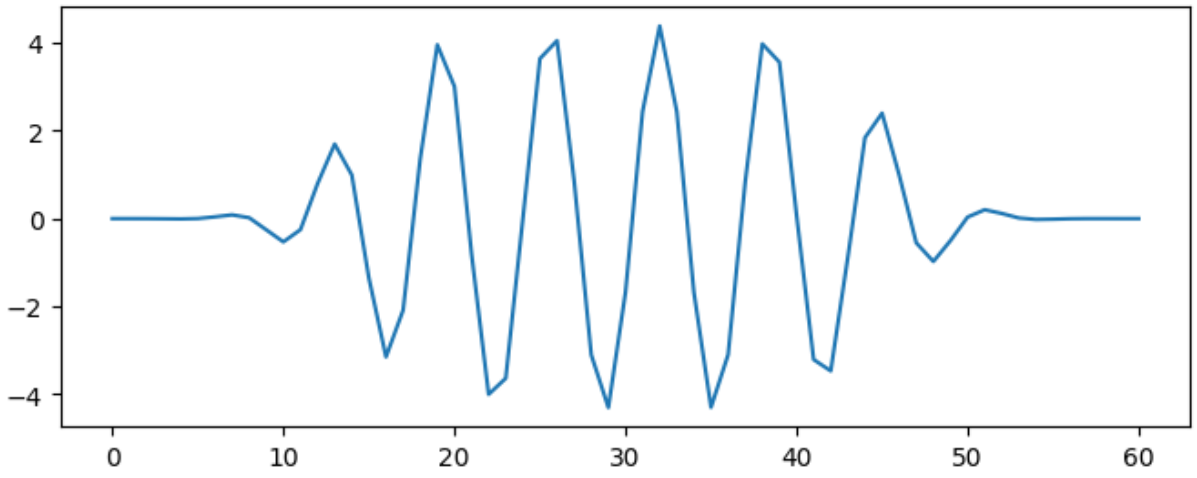
about the equation defining the filter (above), and also by finding the maximum of the computed Fourier amplitudes (using the `max` function), and verify that the answers are the same.

You would receive the largest amplitude response from a **sinusoid of frequency** ω.

When the sinusoid is dotted with $e^{-\frac{n^2}{2\sigma^2}} \cos \omega n$, the $e^{-\frac{n^2}{2\sigma^2}}$ part doesn't depend on ω, so we can look for the sinusoid that maximizes the dot product with $\cos \omega n$. This will be a cosine with frequency ω.

In [6]:
```
n = np.arange(31)
sinusoid = np.cos(omega * n)
out = convolve(sinusoid, gabor)
plt.subplots(figsize=(8, 3))
plt.plot(out)
np.max(out)
```

Out[6]: 4.386364050882267



In [7]:
```
amplitudes = np.abs(fft(gabor, n=64))
np.max(amplitudes)
```

Out[7]: 4.386570296827553

Compute the *period* of this sinusoid, measured in units of sample spacing, and verify by eye that this is matched to the oscillations in your plot of the filter.

In [8]:
```
period = 2 * np.pi / omega
period
```

Out[8]: 6.4

$T = \frac{2\pi}{f}$. One sine cycle in the Gabor filter plot above takes up about 6 and a half line segments. So **it checks**.

Which two sinusoids would produce responses with about 25% of this maximal amplitude?

In [9]:
```
def measure_response(f):
    n = np.arange(nsamples)
    sinusoid_opt = np.cos(2 * np.pi * 10/64 * n)
    sinusoid_f = np.cos(2 * np.pi * f * n)
    amp_opt = np.max(convolve(sinusoid_opt, gabor))
    amp_f = np.max(convolve(sinusoid_f, gabor))
    return amp_f / amp_opt, 1 / f
```

In [10]:
```
measure_response(f=10/64)
```

Out[10]:
```
(1.0, 6.4)
```

Find by iterative checking:

In [11]:
```
measure_response(16.43/64)
```

Out[11]:
```
(0.25003277348849984, 3.8953134510042604)
```

In [12]:
```
measure_response((64 - 16.43)/64)
```

Out[12]:
```
(0.2500327734884952, 1.3453857473197393)
```

So the sinusoids $\cos\left(2\pi\frac{16.43}{64}n\right)$ and $\cos\left(2\pi\frac{64-16.43}{64}n\right)$, with periods 3.89 and 1.34 steps respectively, produce amplitude responses of about 25% maximal amplitude.

c)

Create three unit-amplitude 64-sample sinusoidal signals at the three frequencies (low, mid, high) that you found in part (b). Convolve the filter with each, and verify that the amplitude of the response is approximately consistent with the answers you gave in part (b). (Hint: to estimate amplitude, you'll either need to project the response onto sine and cosine of the appropriate frequency, or compute the DFT of the response and measure the amplitude at the appropriate frequency.)

In [13]:
```
def convolved_amplitude(f):
    signal = np.cos(2 * np.pi * f * n)
    return np.max(convolve(signal, gabor))
```

In [14]:
```
f1 = 10 / 64
f2 = 16.43 / 64
f3 = (64 - 16.43) / 64
convolved_amplitude(f1), convolved_amplitude(f2), convolved_amplitude(f3)
```

Out[14]:
```
(4.386364050882267, 1.0967347691723444, 1.096734769172324)
```

These responses are **25% of the maximum response 4.386.**

In [15]:
```
1.0967 / 4.386
```

Out[15]:
```
0.2500455996352029
```

Homework 3 - Question 3 - Luke Arend

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.fft import fft, ifft, fftshift
from scipy.io import loadmat
from scipy.signal import convolve, resample
```

Neuronal activity causes local changes in deoxyhemoglobin concentration in the blood, which can be measured using functional magnetic resonance imaging (fMRI). One drawback of fMRI is that the haemodynamic response (blood flow in response to neural activity) is much slower than the underlying neural responses. We can model the delay and spread of the measurements relative to the neural signals using a linear shift-invariant system

$$r(n) = \sum_k x(n-k)h(k),$$

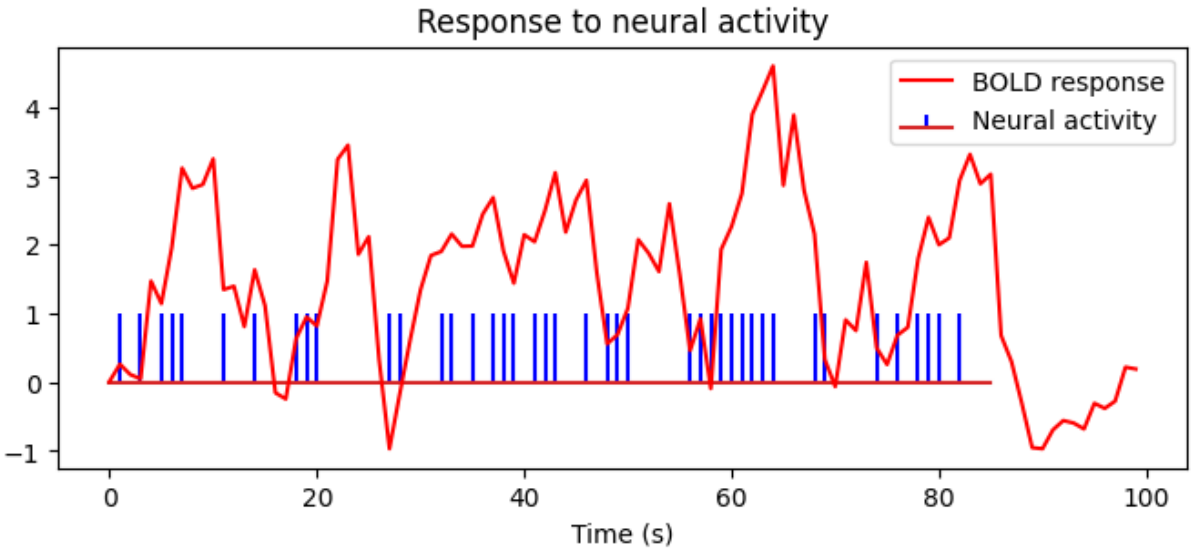
where $x(n)$ is an input signal delivered over time (for example, a sequence of light intensities), $h(k)$ is the haemodynamic response to a single light flash at time $k = 0$ (i.e., the impulse response of the MRI measurement), and $r(n)$ is the MRI response to the full input signal.

In the file `hrfDeconv.mat`, you will find a response vector r and an input vector x containing a sequence of impulses (indicating flashes of light). Your goal is to estimate the HRF, h, from the data. Each of these signals are sampled at 1 Hz.

In [2]:
```
obj = loadmat('hw3/hrfDeconv.mat')
x = np.squeeze(obj['x'])
r = np.squeeze(obj['r'])
```

Plot vectors r and x versus time to get a sense for the data. Use the `stem` command (or `plt.stem` in Python) for x, and label the x-axis.

In [3]:
```
plt.subplots(figsize=(8, 3))
plt.title('Response to neural activity')
plt.stem(x, 'b', label='Neural activity', markerfmt='')
plt.plot(r, 'r', label='BOLD response')
plt.xlabel('Time (s)')
plt.legend()
plt.show()
```



a)

Convolution is linear, and thus we can re-write the equation above as a matrix multiplication, $r = Xh$, where h is a vector of length M, N is the length of the input x, and X is an $[N + M - 1] \times M$ matrix. Write a function `createConvMat`, that takes as arguments an input vector x and M (the dimensionality of h) and generates a matrix X such that the response $r = Xh$ is as defined in Eq. (1) for any h.

In [4]:
```
def create_conv_mat(x, M):
    N = len(x)
    X = np.zeros((N + M - 1, M))
    padding = np.zeros(M - 1)
    x_padded = np.concatenate([padding, x, padding])
    for i in range(N + M - 1):
        X[i, :] = x_padded[i:i + M][::-1]
    return X
```

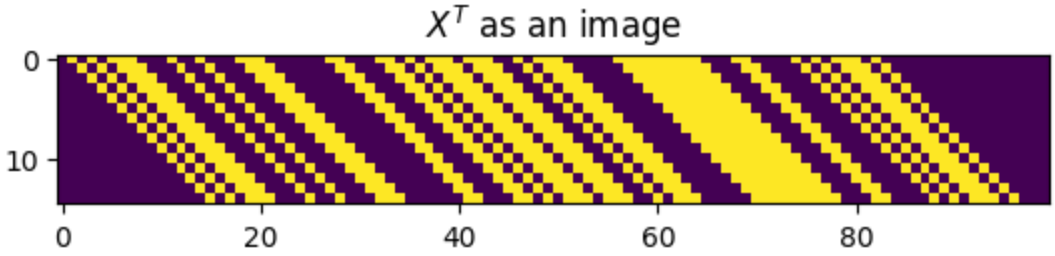
Verify that the matrix generated by your function produces the same response as Matlab's `conv` function when applied to a few random h vectors of length $M = 15$.

In [5]:
```
X = create_conv_mat(x, 15)
for i in range(3):
    h = np.random.randn(15)
    ok = np.allclose(X @ h, convolve(h, x))
    print(ok)
```

```
True
True
True
```

Visualize the matrix X as an image (evaluate `imagesc(X)` in MATLAB or `plt.imshow` in Python), and describe its structure.

In [6]:
```
plt.title('$X^T$ as an image')
plt.imshow(X.T)
plt.show()
```



I plotted X^T since it fits on the page better. The columns in the image above are the rows of X.

The rows of X are views of x through a sliding window of size 15. The first window overlaps with just the first value of x and sees 14 zeros to the left of it. The window takes steps of size 1 along x from left to right. The last window overlaps with just the last value of x and sees 14 zeros to the right of it.

b)

Now, given the X generated by your function for $M = 15$, solve for h by formulating a least-squares regression problem:

$h_{opt} = \text{argmin}_h \|r - Xh\|^2.$

As we saw in class, if X has the singular value decomposition $X = USV^T$, then the β_{opt} which minimizes $\|y - X\beta\|^2$ is $VS^\sharp U^T y$, where $S^\sharp$ is the pseudoinverse of S. So in our case,

$h_{opt} = VS^\sharp U^T r.$

In [7]:
```
U, s, Vt = np.linalg.svd(X)
s
```

Out[7]:
```
array([17.63933449,  6.43899633,  5.77725291,  5.57980237,  5.29411603,
        5.28811517,  5.08114521,  5.07735029,  5.00764389,  4.34175081,
        4.16739293,  3.84902884,  3.68032107,  2.81980577,  2.77262653])
```

In [8]:
```
Vt.shape, U.shape
```

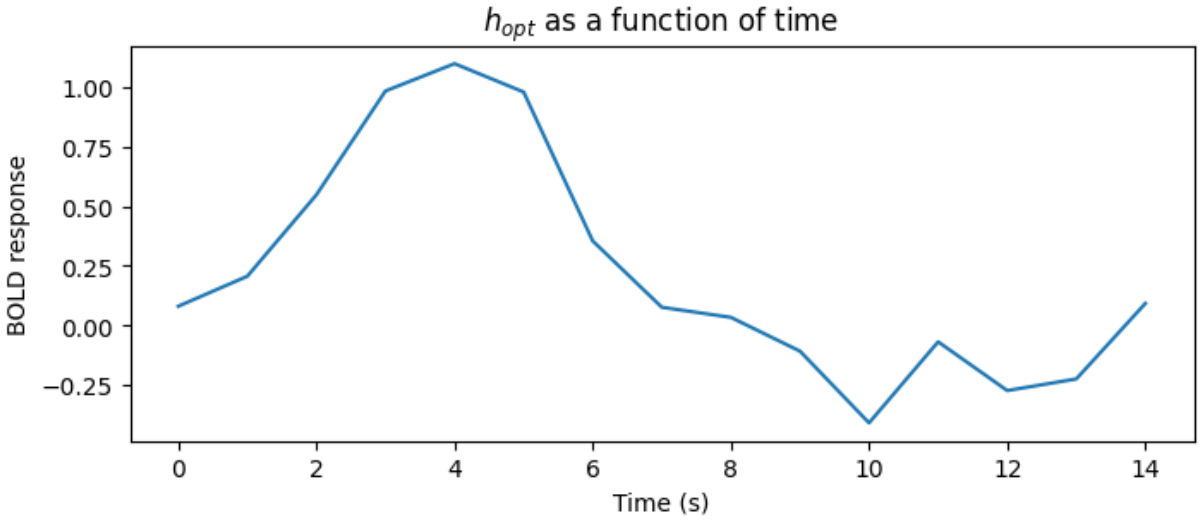
Out[8]:
```
((15, 15), (100, 100))
```

In [9]:
```
padding = np.zeros((len(Vt), len(U) - len(s)))
s_sharp = np.concatenate([np.diag(1 / s), padding], axis=1)
h_opt = Vt.T @ s_sharp @ U.T @ r
h_opt
```

Out[9]:
```
array([ 0.08038615,  0.20651242,  0.54851395,  0.98458329,  1.09930568,
        0.98008733,  0.35467779,  0.07621158,  0.03380402, -0.10810732,
       -0.40934496, -0.0687829 , -0.27365116, -0.22476736,  0.09256217])
```

Plot h_{opt} as a function of time (label your x-axis, including units). How would you describe it? How long does it last?

In [10]:
```
plt.subplots(figsize=(8, 3))
plt.title('$h_{opt}$ as a function of time')
plt.plot(h_opt)
plt.xlabel('Time (s)')
plt.ylabel('BOLD response')
plt.show()
```



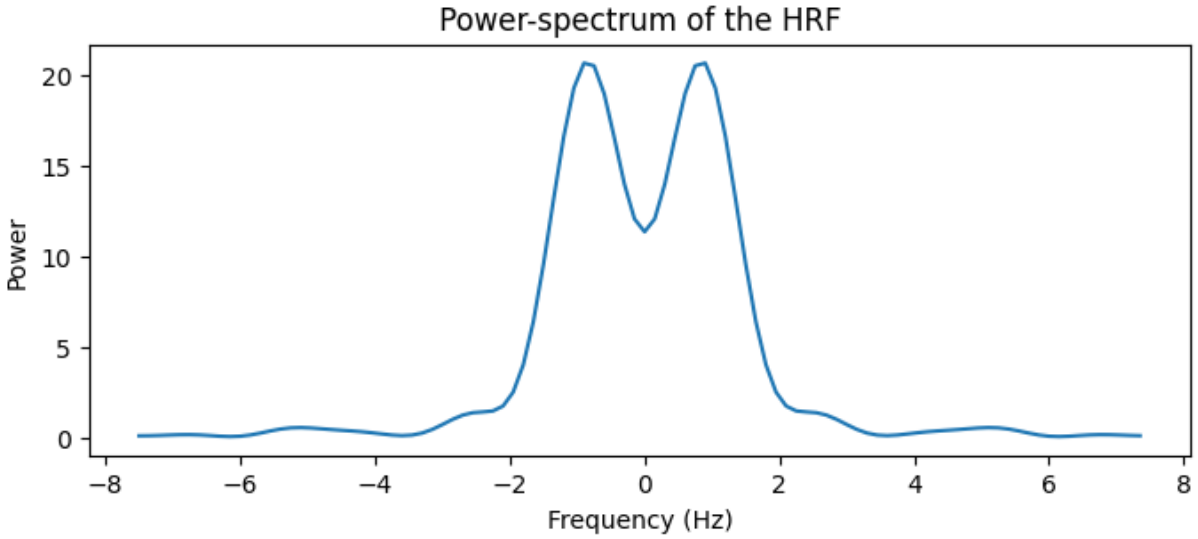
The hemodynamic response ramps up over the first few seconds, reaching a maximum around 1.0 at 4 seconds after stimulus onset, and then ramps back down over the next 6 seconds, reaching a minimum around -0.25 at t = 10. The total response **lasts 15 seconds**.

c)

It's often easier to understand an LSI system by viewing it in the frequency domain. Plot the power-spectrum of the HRF (i.e. $|F(h)|^2$, where $F(h)$ is the Fourier transform of the HRF). Plot this with the zero frequency (DC) in the middle (in Matlab you can use a built-in function called `fftshift`), and label the x axis, in Hz.

In [11]:
```
nbins = 100
n = np.arange(nbins) - 50
freqs = len(h_opt) / 100 * n
h_fft = fft(h_opt, n=nbins)
power = np.abs(h_fft) ** 2
plt.subplots(figsize=(8, 3))
plt.title('Power-spectrum of the HRF')
plt.plot(freqs, fftshift(power))
```

```
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power')
plt.show()
```



Based on this plot, what kind of filter is the HRF? Specifically, which frequencies does it allow to pass, and which does it block?

The HRF is a kind of low-frequency bandpass filter that favors frequencies around 1 Hz and DC signals to some extent. It strongly attenuates 2 Hz and almost completely blocks frequencies 3 Hz and higher.

d)

Use the convolution theorem to now find h_{opt} by working in the Fourier domain. You will need to use the matlab functions `fft` and `ifft`. Remember to be careful about how many samples you choose to have in your `fft`.

The convolution theorem says that if $f = \mathcal{F}(x)$ and $g = \mathcal{F}(y)$ then $\mathcal{F}(x * y) = f \odot g$, where $*$ denotes convolution and $\odot$ denotes elementwise multiplication. Or in our case,

$$\mathcal{F}(x * h) = \mathcal{F}(r) = \mathcal{F}(x) \odot \mathcal{F}(h)$$

We can get $\mathcal{F}(h)$ by taking $\mathcal{F}(r)$ elementwise divided by $\mathcal{F}(x)$, and get h by taking the inverse Fourier transform of $\mathcal{F}(h)$.

In [12]:
```
def fourier_solve(x, r, nbins):
    r_fft = fft(r, n=nbins)
    x_fft = fft(x, n=nbins)
    window = np.abs(len(x) - len(r)) + 1
    return ifft(r_fft / x_fft, n=nbins)[:window]
```

```
h_opt2 = np.real(fourier_solve(x, r, nbins=86 * 100))
h_opt2
```

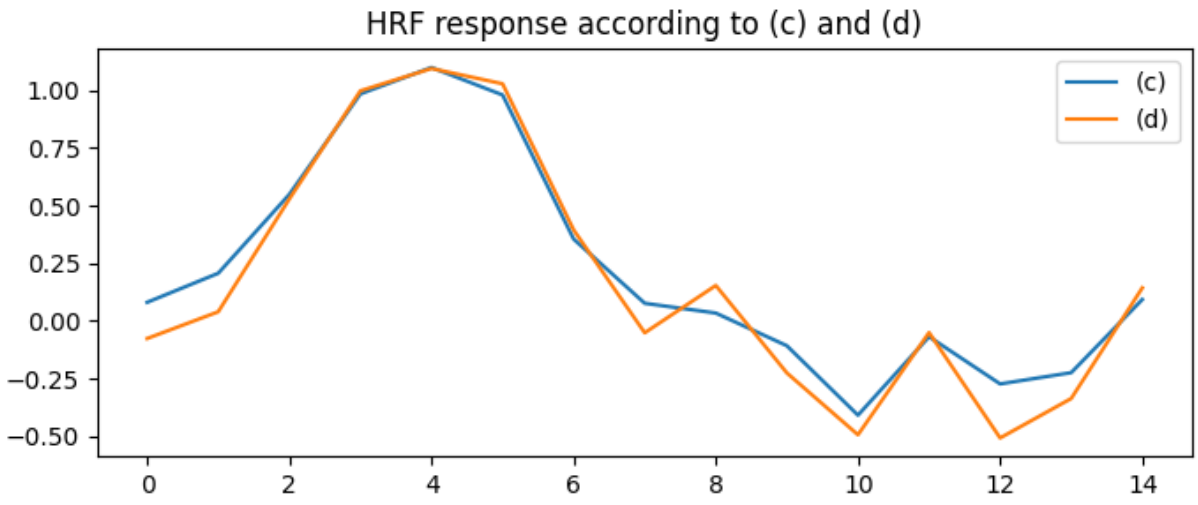
Out[12]:
```
array([-0.07660524,  0.03939486,  0.52864874,  0.99800785,  1.0947841 ,
        1.02850338,  0.39444263, -0.05186693,  0.15348495, -0.22611325,
       -0.49428447, -0.05086755, -0.50811995, -0.33690616,  0.14311   ])
```

Based on the operations you have done, what can you say about when this method will fail?

This method reconstructs h_{opt} with better precision as the Fourier `nbins` grows. Also, due to Fourier resampling, the best reconstructions occur when `nbins` is a multiple of the lengths of `x` and `r`. The reconstruction degrades as `nbins` decreases and a full signal **cannot be constructed if `nbins` is less than 15**.

On the same graph, plot the HRF impulse response you recovered from parts (c) and (d).

In [13]:
```
plt.subplots(figsize=(8, 3))
plt.title('HRF response according to (c) and (d)')
plt.plot(h_opt, label='(c)')
plt.plot(h_opt2, label='(d)')
plt.legend()
plt.show()
```



In [ ]:
```

```

Homework 3 - Question 4 - Luke Arend

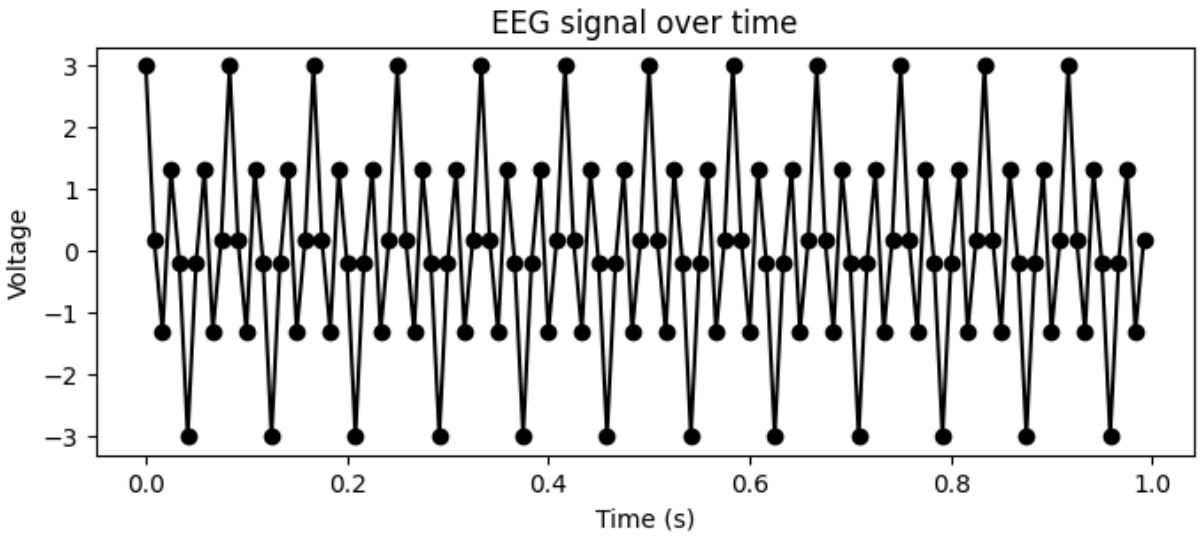
In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.fft import fft, ifft, fftshift
from scipy.io import loadmat
```

Load the file `myMeasurements.mat` into Matlab. It contains a vector, `sig`, containing voltage values measured from an EEG electrode, sampled at 120 Hz.

In [2]:
```
obj = loadmat('hw3/myMeasurements.mat')
sig = np.squeeze(obj['sig'])
t = np.squeeze(obj['time'] / 120)
```

Plot `sig` as a function of vector time (time, in seconds), using the flag `'ko-'` in matlab's plot command so you can see the samples.

In [3]:
```
plt.subplots(figsize=(8, 3))
plt.title('EEG signal over time')
plt.plot(t, sig, 'ko-')
plt.xlabel('Time (s)')
plt.ylabel('Voltage')
plt.show()
```



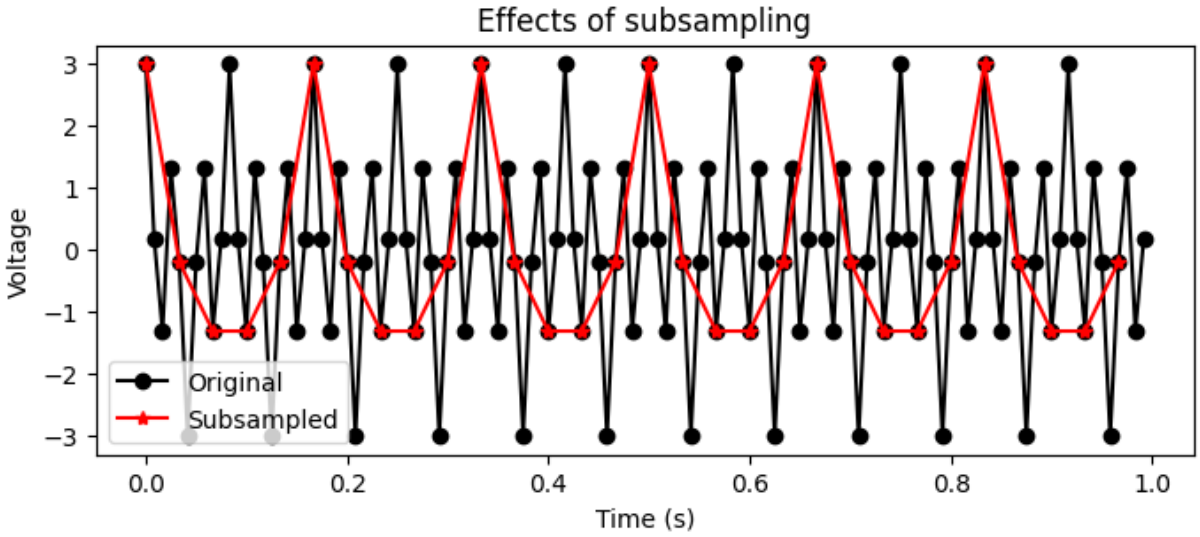
a)

The voltage signal is densely sampled (120 Hz), and thus expensive to store. Create a subsampled version of the signal, which contains every fourth value. Plot this, against

the corresponding entries of the time vector, on top of the original data (use matlab's `hold` function, and plot with flag `'r*-'`).

In [4]:
```
def subsample(x, factor=4, offset=0):
    return x[offset::factor]

plt.subplots(figsize=(8, 3))
plt.title('Effects of subsampling')
plt.plot(t, sig, 'ko-', label='Original')
plt.plot(subsample(t), subsample(sig), 'r*-', label='Subsampled')
plt.xlabel('Time (s)')
plt.ylabel('Voltage')
plt.legend()
plt.show()
```



Is the subsampling operation linear? Shift-invariant?

It's easy enough to see that subsampled(A + B) = subsampled(A) + subsampled(B), so subsampling **is linear**. However, subsampling starting from index 0 gives a different result than subsampling starting from index 1. So subsampling **is not shift-invariant**.

How does this reduced version of the data look, compared to the original? Does it provide a good summary of the original measurements? Explain.

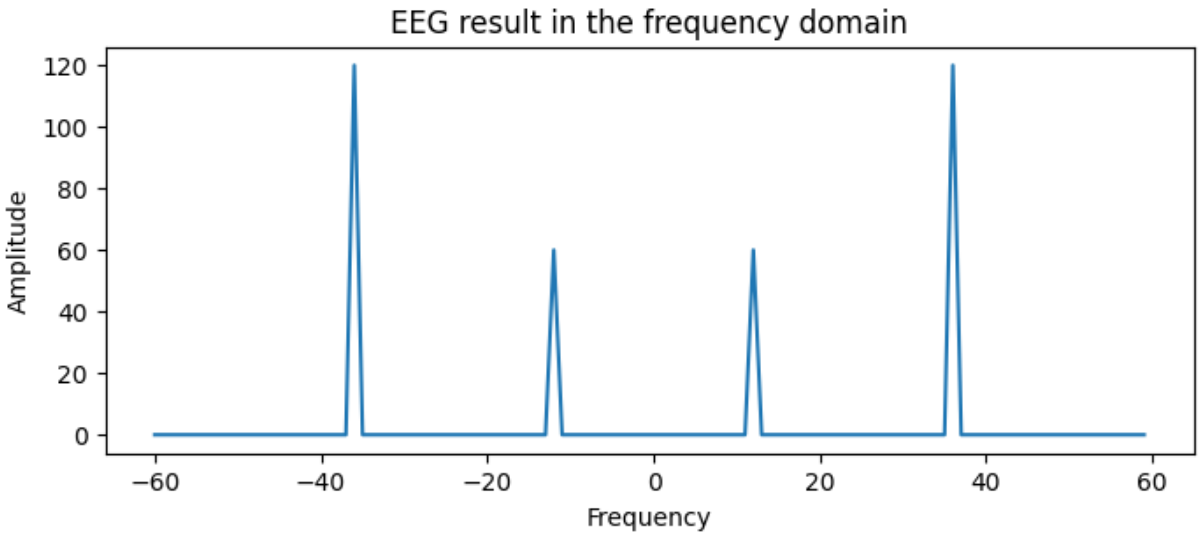
The subsampled signal is aliased, showing a low frequency not present in the original data. In this case subsampling **does not** provide a good estimate of the signal since the frequencies in the original signal exceed the sampling rate.

b)

Examine your EEG result in the frequency domain. First plot the magnitude (amplitude) of the Fourier transform of the original signal, over the range $[-N/2, (N/2) - 1]$ (use

`fftshift`).

In [5]:
```
amplitude = np.abs(fft(sig))
nfreqs = len(amplitude)
freqs = np.arange(nfreqs) - nfreqs // 2
plt.subplots(figsize=(8, 3))
plt.plot(freqs, fftshift(amplitude))
plt.title('EEG result in the frequency domain')
plt.xlabel('Frequency')
plt.ylabel('Amplitude')
plt.show()
```



By convention, the “Delta” band corresponds to frequencies less than 4 Hz, “Theta” band is 4-7 Hz, “Alpha” band 8-15 Hz, and “Beta” is 16-31 Hz. For these data, which band shows the strongest signal? Is there any power in frequencies outside of these known bands, and if so can you explain the origin of this part of the signal?

In [6]:
```
peaks = amplitude[:nfreqs // 2] > 0.01
np.nonzero(peaks)[0]
```

Out[6]:
```
array([12, 36])
```

There is power at the frequencies 12 Hz and 36 Hz. The 12 Hz frequency is within the alpha band. 36 Hz is a bit above (and outside) the beta band. In the original signal, the high-frequency component has a cycle length around 3 1/3 line segments. Dividing the length of the whole signal (120 samples/sec), by this period (3.33 samples/cycle), gives 36 cycles/second, explaining the 36 Hz component present in the Fourier transform.

c)

Write a function `signalPart = bandWiseReconstruct(bandName)` that reconstructs the signal (and plots the reconstruction) using only sinusoids from the

band corresponding to the string `bandName` (i.e. for `bandName = 'Delta'` the reconstruction should be a sum of sinusoids with frequencies from 0-4 Hz).

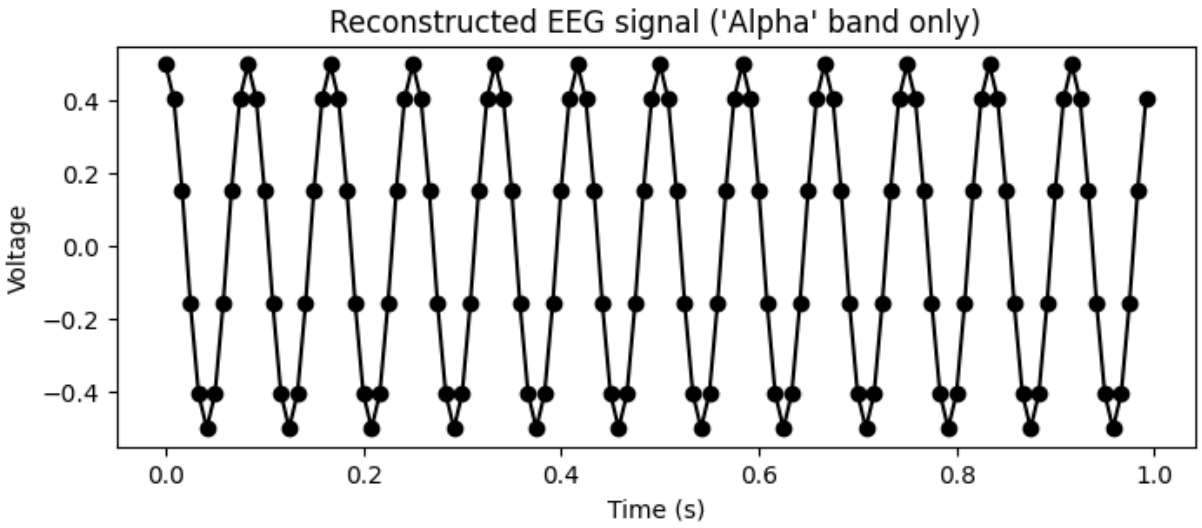
In [7]:
```
freq_bands = {
    'Delta': [0, 4],
    'Theta': [4, 7],
    'Alpha': [8, 15],
    'Beta': [16, 31]
}

def bandwise_reconstruct(band_name):
    start, end = freq_bands[band_name]
    middle = nfreqs // 2
    mask = np.zeros(nfreqs, dtype='bool')
    mask[start:end] = 1
    mask[middle + start:middle + end] = 1

    part_fft = np.zeros(nfreqs, dtype='complex')
    part_fft[mask] = fft(sig)[mask]
    signal_part = ifft(part_fft)

    plt.subplots(figsize=(8, 3))
    plt.title(f"Reconstructed EEG signal ('{band_name}' band only)")
    plt.plot(t, np.real(signal_part), 'ko-')
    plt.xlabel('Time (s)')
    plt.ylabel('Voltage')
    return signal_part
```

In [8]:
```
reconstructed = bandwise_reconstruct('Alpha')
```

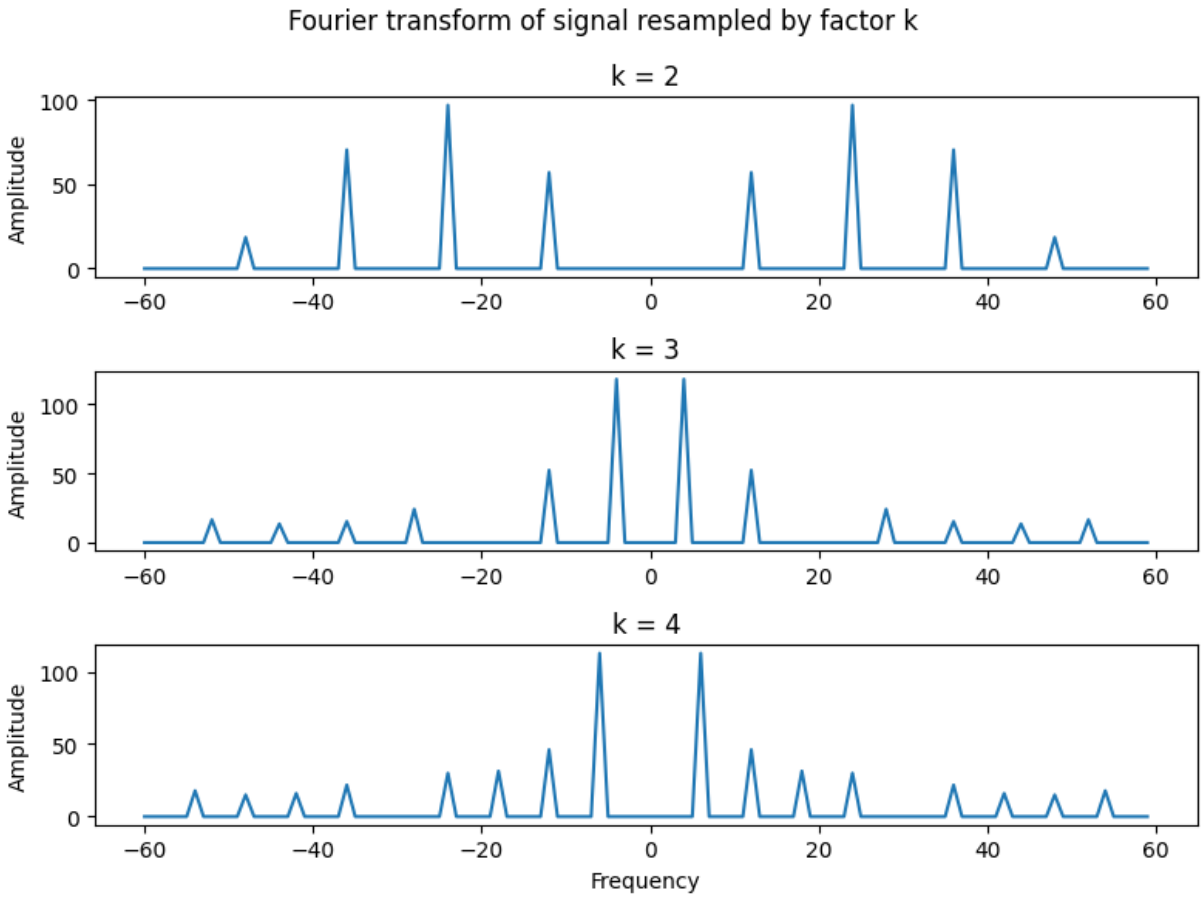


d)

Plot the Fourier magnitude for signals downsampled by factors of 2, 3, and 4, after upsampling them back to full size (i.e., make a full-size signal filled with zeros, and set every kth sample equal to one of the subsampled values, for subsampling by factor k).

In [9]:
```
signals = []
for k in [2, 3, 4]:
    subsampled = subsample(sig, factor=k)
    resampled = np.zeros(nfreqs)
    for offset in range(k):
        resampled[offset::k] = subsampled
    signals.append(resampled)

fig, axs = plt.subplots(3, 1, figsize=(8, 6))
plt.suptitle('Fourier transform of signal resampled by factor k')
freqs = np.arange(nfreqs) - nfreqs // 2
for i in range(3):
    plt.sca(axs[i])
    plt.title(f'k = {i + 2}')
    plt.plot(freqs, fftshift(np.abs(fft(signals[i]))))
    plt.ylabel('Amplitude')
plt.xlabel('Frequency')
plt.tight_layout()
plt.show()
```



What is the relationship between these plots and the original frequency plot?

These plots contain peaks at the same frequencies as the original signal, but also have **new peaks at frequencies not present in the original**. These are artifacts due to resampling.

What has happened to the frequency components of the original signal?

The frequency components of the original signal remain **present but they are rescaled**. In general they are reduced as some power from their bands has migrated to the new frequency bands that were not present in the original signal.

Does the strongest signal band change?

Yes. The strongest band in the original signal was at 36 Hz, but the strongest band in each resampled version of the signal differs from all the others.

In [ ]: