Discrete-time signals

Implementing and analysing an IIR system

Andre' Vella

November 21, 2021

Preliminaries

Given a first-order infinite-duration impulse response (IIR) filter described by the difference equation:

$$y(n) = 0.6y(n-1) + 0.2x(n) + 0.2x(n-1)$$

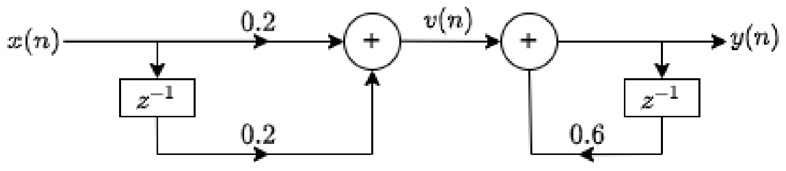


Figure 1: A first-order system: direct form I

This system can be divided into 2 systems:

- $v(n) = 0.2x(n) + 0.2x(n-1)$ which is not recursive i.e. no feedback is shown in the diagram above.
- $v(n) = v(n) + 0.6y(n-1)$ which is recursive i.e. feedback is shown in the diagram above.

Considering properties of linearity, commutative law etc, we can achieve a direct form I with swapped parts as shown below:

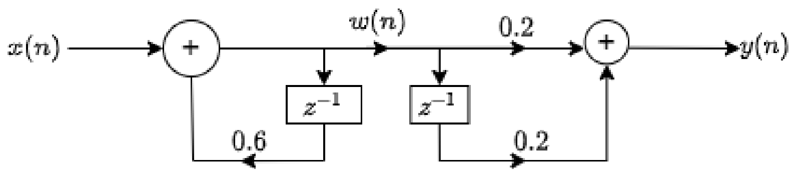


Figure 2: A first-order system: direct form I with swapped parts

This system can be divided into 2 new systems:

- $w(n) = 0.6w(n-1) + x(n)$ which is recursive i.e. feedback is shown in the diagram above.
- $y(n) = 0.2w(n) + 0.2w(n-1)$ which is not recursive i.e. no feedback is shown in the diagram above.

Which can be reduced to the direct form II.

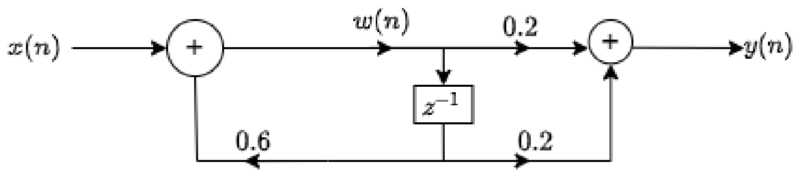


Figure 3: Direct form ii

Tasks

Task 1. Modelling the IIR filter in Python, using a python function that takes a NumPy array as input and outputs the response of the system shown in the Preliminaries using the direct form ii representation as shown in Figure 3.

```python
import numpy as np
def filter(x):
    # supposing that our n starts from 0 onwards we can
    # set w(n-1)=0 when n=0
    w_prev = 0
    y = np.zeros_like(x)
    for n, x_n in enumerate(x):
        # update w_n
        w_n = 0.6*w_prev+x_n
        # set output
        y[n] = 0.2*w_n+0.2*w_prev
        # set w_prev for next iteration
        w_prev = w_n
    # return list
    return y
```

Task 2. (a) Generating an input sequence where the input is the impulse sequence, $x(n) = \delta(n)$ for a length of 10 samples.

```
x = np.zeros(10)
x[0]=1
```

(b) Producing an output sequence $y(n)$ by passing the input sequence to the filter from Task 1,

```
y=filter(x);
```

(c) Plotting the output.

```
import matplotlib.pyplot as plt
fig, ax= plt.subplots()
n = np.arange(len(y))
ax.stem(n, y)
ax.set_xlabel('$n$')
ax.set_ylabel('$y(n)$')
plt.show()
```

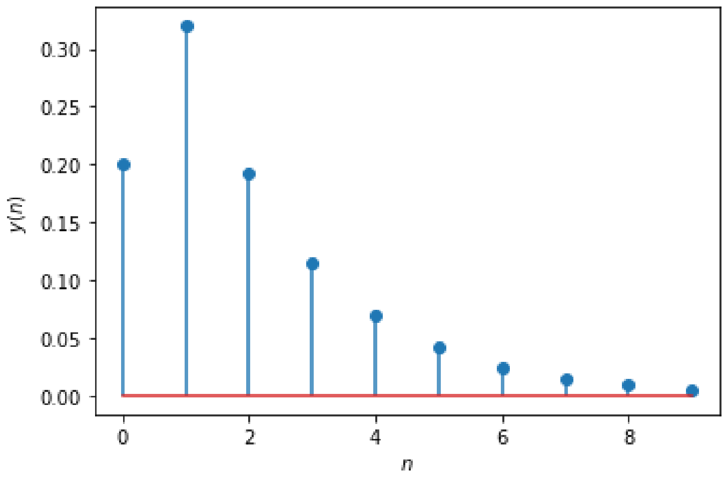


Figure 4: Plot of output y when $x[n] = \delta[n]$.

Task 3. Let the input be the unit step signal, $x[n] = U[n]$ for a length of 20 samples.

```
x = np.ones(20);
```

Repeating task *2b-2c* for this input sequences, we get the following plot:

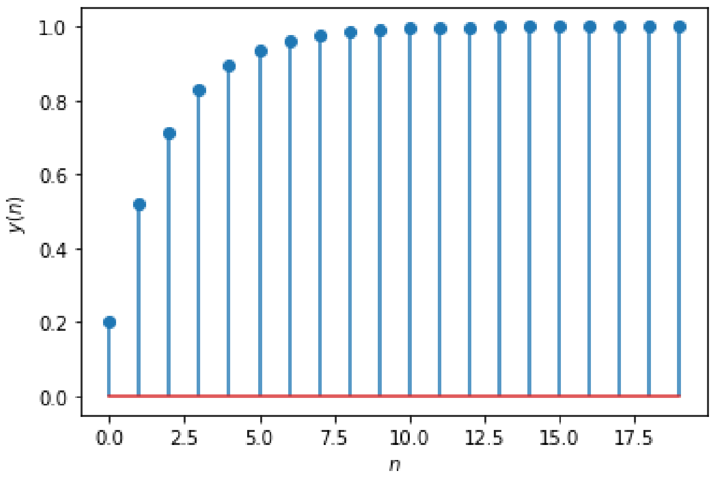


Figure 5: Plot of output y when $x[n] = U[n]$.

Task 4. Let the input be the unit step signal, $x[n] = U[n] - U[n-10]$ for a length of 20 samples.

```
x=np.ones(20)
x[10:20]=0
```

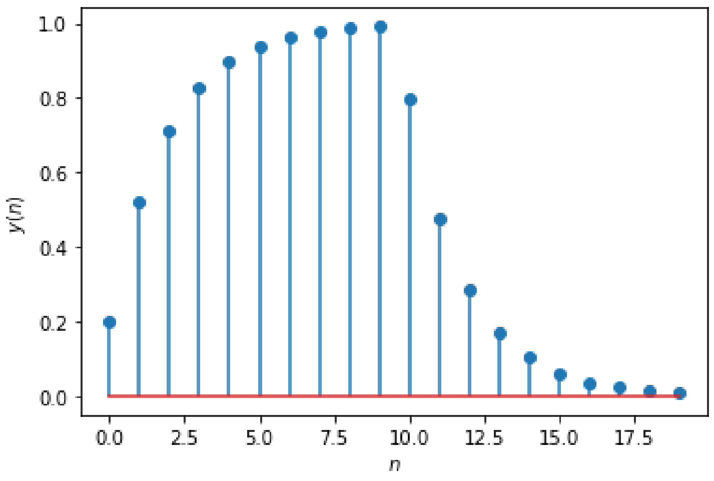


Figure 6: Plot of output y when $x[n] = U[n] - U[n-10]$.

Task 5. (a) Reading the input sequence $x(n)$ from the file:

```python
from scipy.io import wavfile
[fs, x]= wavfile.read('tones.wav')
```

(b) Filtering the audio using the filter from Task 1.

```python
y=filter(x)
```

(c) Saving the output.

```python
wavfile.write('output_tones.wav', fs, y)
```

Task 6. Listening and comparing the input audio file and the output file, I assume that the input signal goes to a higher pitch before the output signal. I think that the input signal gains a higher frequency earlier than the output signal. This may be due the fact that the output signal depends on previous value of time from the input signal.

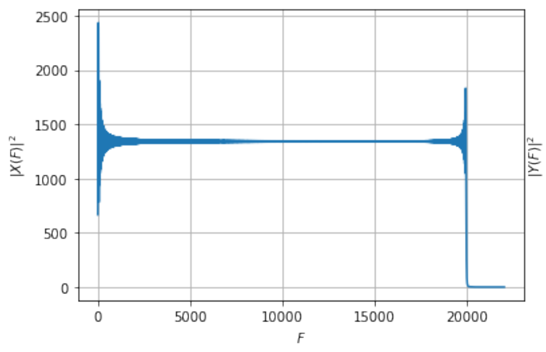
Note, this is only an assumption and an opinion. Everyone can perceive since not everyone's sense of listening is the same.

Task 7. Plotting the power spectrum of both the input signal and the output signal.

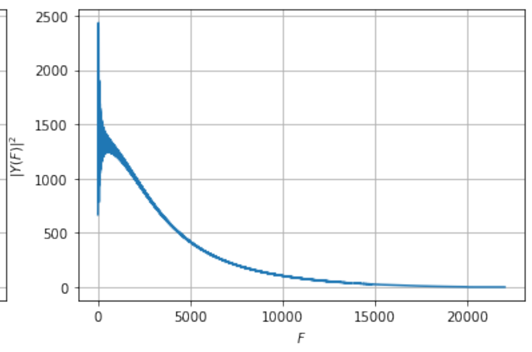
(a) To plot the spectrum of Y against F:

```python

Y = fft(y)/ len(y)
# spectrum is symmetric, so remove second half
Y = Y[:len(Y) // 2]
Yp = np.abs(Y)** 2
# frequencies corresponding to the output of fft
F = fftfreq(len(y),1 / fs)
F = F[:len(F) // 2]
fig, ax= plt.subplots()
ax.plot(F, Yp)
ax.set_xlabel('$F$')
ax.set_ylabel('$|Y(F)|^2$')
ax.grid(True)
plt.show()
```



(a) Power spectrum of input signal.



(b) Power spectrum of output signal.

Figure 7: Plot of the power spectrum of both the input signal and the output signal

In view the power spectrum shown in the figures above, I notice that the volume in the output signal starts dipping as frequency increases, which what could have created the impression of the effect that I taught of in Task 6.

Task 8. (a) Generating a sinusoid $x(n) = \cos \omega_0 n$ as the new input, where w_0 is the signal frequency in rad/sample. Take $w_0 = 0.1 rad/sample$. To generate a sinusoid of length L and generating the output by passing the input signal through the filter.

```
import mpld3
mpld3.enable_notebook()

import matplotlib.pyplot as plot
# setting signal frequency to 0.1 rad/sample
w_0 = 0.1
# setting length = 100
L = 100
n = np.arange(L)
x = np.cos(n*w_0)
y = filter(x)
```

(b) Plotting the input and output signal.

```
fig, ax = plt.subplots()
ax.plot(n, x, label='$x(n)$')
ax.plot(n, y, label='$y(n)$')
ax.set_xlabel('$n$')
ax.legend(loc='best', framealpha=1)
ax.grid(True)
plot.show()
```

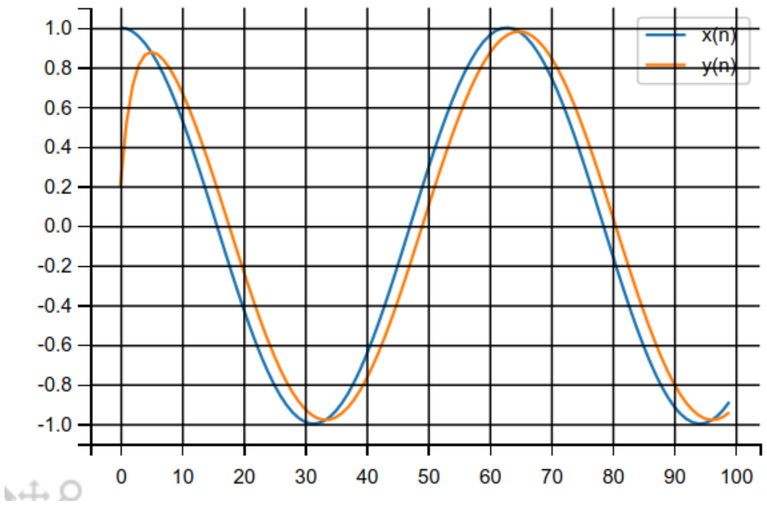


Figure 8: Input $x[n]$ and output $y[n]$

(c) Determining the delay: this can be meseaured at the zero crossing of the sinusoid.

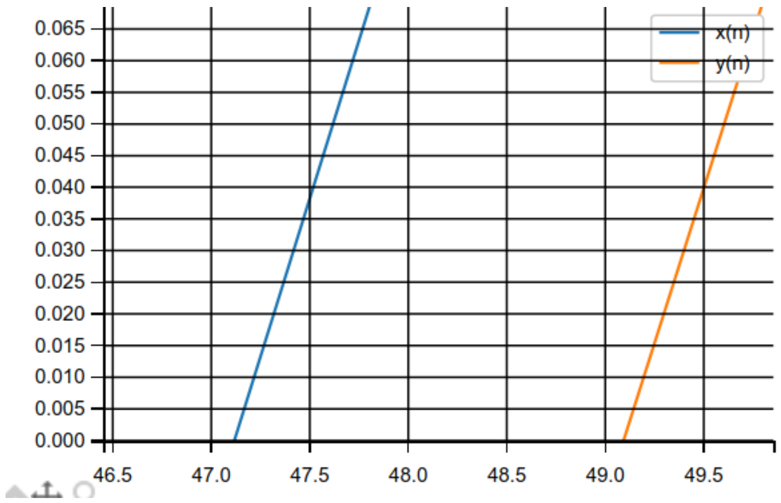


Figure 9: Zooming into the graph.

Hence the delay is given by:

$$80.515 - 78.54 \approx 1.975$$

.

Note that this delay is measured in samples. Converting into radians we get

$\approx 1.975(0.1) = 0.1975$

NB:

- we have $0.1 rad/samples$
- we have 100 samples
- total radians in sample space 10 .. [0,10]
- Gain is defined as the magnitude of the output sinusoid divided by the magnitude of the input sinusoid.

> **NB:** The gain is given by magnitude of y divided by the magnitude of x, but the magnitude of x is 1. So the function `np.max(y[20:100])` is used to find the gain while ignoring the transient part.

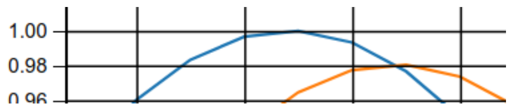


Figure 10: Measuring the gain.

In this example we have that the gain is $\approx \frac{0.9803}{1} \approx 0.98$

Task 9. (a) Generating 10 logarithmically spaced values for w_0 in the range 0.1 to π

```python
w_0 = np.geomspace(0.1, np.pi, 10)
```

(b) Repeating task 8 for all values of w_0 we get the following values, (worked in the same way as task 8):

(i) For $w_0[0] = 0.1$
- Gain and Phase shift already worked out.

(ii) For $w_0[1] = 0.14667293612499196$
- Gain: 0.959
- Phase shift: 0.286

(iii) For $w_0[2] = 0.21512950191525967$
- Gain: 0.918
- Phase shift:0.408

(iv) For $w_0[3] = 0.31553675693018207$

- Gain:0.840
- Phase shift:0.567

(v) For $w_0[4] = 0.46280702594307715$
- Gain:0.727
- Phase shift:0.758

(vi) For $w_0[5] = 0.6788126535434645$
- Gain:0.578
- Phase shift:0.959

(vii) For $w_0[6] = 0.9956344497401685$
- Gain:0.418
- Phase shift:1.145

(viii) For $w_0[7] = 1.4603262805058121$
- Gain: 0.269
- Phase shift:1.258

(ix) For $w_0[8] = 2.141903432622761$
- Gain:0.135
- Phase shift:1.699

(x) For $w_0[8] = 3.141592653589793$
- Gain: 0 (assumed that it converged to zero since np.max returned a negative exponential)
- Phase shift: 0

(c) Plotting the magnitude and phase response of the filter using my measurements:

(i) Generating NumPy arrays for magnitude gain and phase gain:

```
amplitude_gain = np.array
    ([0.98,0.959,0.918,0.840,0.727,0.578,0.418,0.269,0.135])
phase = np.array
    ([0.1975,0.286,0.408,0.567,0.758,0.959,1.145,1.258,1.699])
```

(ii) Generating the power gain in dB by using the equation $20log_{10}G$. Where G is the amplitude gain.

```
power_gain = 20*np.log10(amplitude_gain)
```

(iii) Plotting with `semilogx` method.

```python
plt.subplot(2, 2, 1)
plt.semilogx(w_0[0:9], power_gain, marker = ".",
    markersize = 15,
    color = "green")
# set the title
plt.title("Magnitude response")
plt.show

plt.subplot(2, 2, 2)
plt.semilogx(w_0[0:9], phase, marker = ".",
    markersize = 15,
    color = "blue")
# set the title
plt.title("Phase response")
plt.show
```

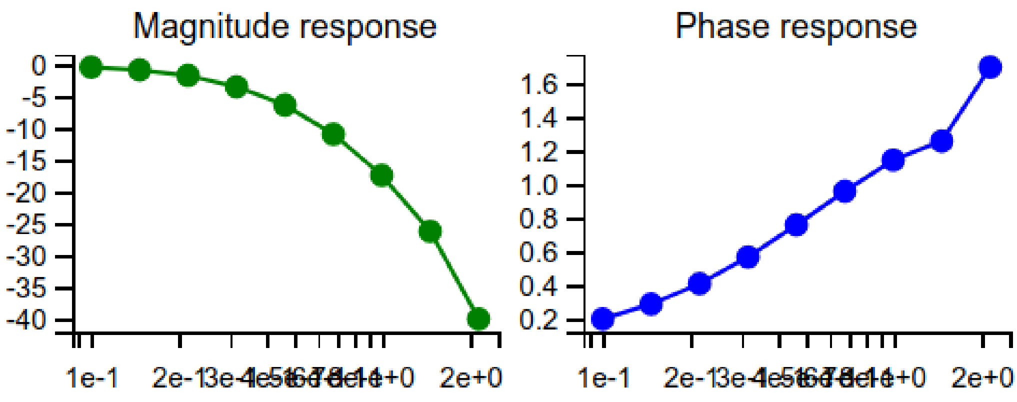


Figure 11: Magnitude and phase response plots.

End of practical.