Digital Signal Processing Lab

Demo 54 - Exercise 1 (filter with live spectrum)

Saad Zubairi
shz2020

October 15, 2025

Solution

To solve this problem, we started with the demo programs from the course materials that handle real-time audio plotting and playback (e.g., `demo 12.py` and `prog_B4.py`) and combined them with the recursive difference-equation filter implementation used previously for wave-file processing (e.g., `prog_A5.py`).

This integration allows live microphone input to be filtered, played back in real time, and simultaneously visualized in both time and frequency domains.

Overview of Modifications

- **Microphone Input and Playback Setup**: The PyAudio stream is opened in full-duplex mode to allow real-time capture and playback

```
p = pyaudio.PyAudio()
PA_FORMAT = p.get_format_from_width(WIDTH)
stream = p.open(
    format = PA_FORMAT,
    channels = CHANNELS,
    rate = RATE,
    input = True,
    output = True,
    frames_per_buffer = BLOCKLEN
)

```

- **High-Pass Filter Implementation**: The high-pass filter is implemented using the recursive difference equation:

$$y[n] = \alpha \left(y[n-1] + x[n] - x[n-1]\right)$$

The cutoff frequency was chosen to match the previous Butterworth setup (fc = 0.1 * RATE ≈ 800 Hz).
Initialization of state variables is as follows

```
fc_hz = 0.1 * RATE
RC = 1.0 / (2 * np.pi * fc_hz)
T = 1.0 / RATE
alpha = RC / (RC + T)

x_prev = 0.0
y_prev = 0.0

```

Snippet 1: Filter Initialization

- **Block-Level Processing**: For each block of samples read from the microphone, the filter is applied sample-by-sample before playback.
The implementation uses the same variable naming and conventions as previous assignments:

```
for n in range(BLOCKLEN):
xn = x_block[n]
yn = alpha * (y_prev + xn - x_prev)
y_block[n] = yn
x_prev = xn
y_prev = yn
y_play = np.clip(y_block, -32768, 32767).astype('int16')
```

```
    stream.write(y_play.tobytes(), BLOCKLEN)

```

Snippet 2: Recursive Block-Level Filter Application

- **Block-Level Processing**: The Matplotlib FuncAnimation framework is used to plot:
  - Input Signal (time domain)
  - Spectrum of Input (with frequency response $\times 100$)
  - Output Signal
  - Spectrum of Output

  These are updated per animation frame to display the filtering effect in real time, matching the provided example layout:

```
    my_anima = animation.FuncAnimation(
        fig1,
        my_update,
        init_func = my_init,
        interval = 10,
        blit = True,
        cache_frame_data = False,
        repeat = False
    )
    pyplot.show()

```

Snippet 3: Recursive Block-Level Filter Application

Addendum: Full implementation

```
import pyaudio
import matplotlib
from matplotlib import pyplot
from matplotlib import animation
import numpy as np

matplotlib.use('TkAgg')
print('The matplotlib backend is %s' % pyplot.get_backend())
WIDTH = 2               # bytes per sample
CHANNELS = 1            # mono
RATE = 8000             # frames per second
BLOCKLEN = 512          # block length in samples
# BLOCKLEN = 256
print('Block length: %d' % BLOCKLEN)
print('Duration of block in milliseconds: %.1f' % (1000.0 * BLOCKLEN / RATE))

p = pyaudio.PyAudio()
print("Default input device:", p.get_default_input_device_info()["name"])
PA_FORMAT = p.get_format_from_width(WIDTH)
stream = p.open(
    format=PA_FORMAT,
    channels=CHANNELS,
    rate=RATE,
    input=True,
    output=True,
    input_device_index=None,
    output_device_index=None,
    frames_per_buffer=BLOCKLEN
)

# high pass filter diff equation
fc_hz = 0.1 * RATE
RC = 1.0 / (2.0 * np.pi * fc_hz)
T = 1.0 / RATE
alpha = RC / (RC + T)

x_prev = 0.0
y_prev = 0.0

# figure prep
fig1 = pyplot.figure(1)
fig1.set_size_inches((12, 7))

ax_x = fig1.add_subplot(2, 2, 1)
ax_X = fig1.add_subplot(2, 2, 2)
ax_y = fig1.add_subplot(2, 2, 3)
ax_Y = fig1.add_subplot(2, 2, 4)

t = np.arange(BLOCKLEN) * (1000.0 / RATE)
x = np.zeros(BLOCKLEN)
X = np.fft.rfft(x)
f_X = np.arange(X.size) * RATE / BLOCKLEN

# Precompute HPF frequency response curve for plotting
# H(e^jw) = alpha * (1 - e^{-jw}) / (1 - alpha * e^{-jw})
w = 2.0 * np.pi * (np.linspace(0, RATE/2, num=X.size) / RATE)  # rad/sample
```

```python
ejw = np.exp(-1j * w)
H = alpha * (1.0 - ejw) / (1.0 - alpha * ejw)
f_H = np.linspace(0, RATE/2, num=X.size)

# input signal plot
[g_x] = ax_x.plot([], [])
ax_x.set_ylim(-10000, 10000)
ax_x.set_xlim(0, 1000.0 * BLOCKLEN / RATE)
ax_x.set_xlabel('Time (milliseconds)')
ax_x.set_title('Input signal')

# input spectrum plot (+ HPF response x100)
[g_X] = ax_X.plot([], [])
[g_H] = ax_X.plot(f_H, 100.0 * np.abs(H), label='Frequency response (x100)',
    color='green')
ax_X.set_xlim(0, RATE/2)
ax_X.set_ylim(0, 300)  # matches the visual scale in your screenshot
ax_X.set_title('Spectrum of input signal')
ax_X.set_xlabel('Frequency (Hz)')
ax_X.legend()

# output signal plot
[g_y] = ax_y.plot([], [])
ax_y.set_ylim(-10000, 10000)
ax_y.set_xlim(0, 1000.0 * BLOCKLEN / RATE)
ax_y.set_xlabel('Time (milliseconds)')
ax_y.set_title('Output signal')

# output spectrum plot
[g_Y] = ax_Y.plot([], [])
ax_Y.set_xlim(0, RATE/2)
ax_Y.set_ylim(0, 500)   # matches the visual scale in your screenshot
ax_Y.set_title('Spectrum of output signal')
ax_Y.set_xlabel('Frequency (Hz)')

fig1.tight_layout()

def my_init():
    g_x.set_xdata(t)
    g_x.set_ydata(x)
    g_y.set_xdata(t)
    g_y.set_ydata(x)
    g_X.set_xdata(f_X)
    g_X.set_ydata(np.abs(X))
    g_Y.set_xdata(f_X)
    g_Y.set_ydata(np.abs(X))
    return (g_x, g_y, g_X, g_Y)

def my_update(i):
    global x_prev, y_prev

    # read audio input stream (mic)
    signal_bytes = stream.read(BLOCKLEN, exception_on_overflow=False)

    # convert binary data to numpy int16
    x_block = np.frombuffer(signal_bytes, dtype='int16').astype(np.float64)

    # recursive HPF per-sample
    y_block = np.empty_like(x_block)
```

```
    xp = x_prev
    yp = y_prev
    for n in range(BLOCKLEN):
        xn = x_block[n]
        yn = alpha * (yp + xn - xp)
        y_block[n] = yn
        xp = xn
        yp = yn
    x_prev = xp
    y_prev = yp

    y_play = np.clip(y_block, -32768, 32767).astype('int16')

    Xk = np.fft.rfft(x_block) / BLOCKLEN
    Yk = np.fft.rfft(y_block) / BLOCKLEN

    # update
    g_x.set_ydata(x_block)
    g_y.set_ydata(y_block)
    g_X.set_ydata(np.abs(Xk))
    g_Y.set_ydata(np.abs(Yk))

    #play
    stream.write(y_play.tobytes(), BLOCKLEN)

    return (g_x, g_y, g_X, g_Y)

my_anima = animation.FuncAnimation(
    fig1,
    my_update,
    init_func=my_init,
    interval=10,          # milliseconds
    blit=True,
    cache_frame_data=False,
    repeat=False
)
pyplot.show()

stream.stop_stream()
stream.close()
p.terminate()
print('* Finished')
```

Snippet 4: example code