Digital Signal Processing Lab

Demo 59 - Exercise 2 (multi-note keyboard)

Saad Zubairi
shz2020

October 29, 2025

Solution

The original program used a single second-order difference equation (a resonator filter) to play a single note, triggered by a keyboard impulse. To enable the ability to play multiple notes/chords simultaneously, the system was extended to manage **12 independent filters**, one for each note in a musical octave.

Overview of Code Modifications

The changes primarily involve replacing single, note-specific variables with lists and iterating through these lists in the main processing loop to simulate 12 parallel synthesizers.

- **Frequency and Filter Setup**:
  - **Frequencies**: A list `frequencies` is pre-calculated for $k = 0$ to 11.
  - **Coefficients**: The coefficients (a, b) for each frequency are stored in `a_list` and `b_list`. These lists contain the unique parameters for all 12 filters.
  - **Filter States**: The state memory for each filter is critical for its independent operation. This is stored in a list of NumPy arrays: `states_list = [np.zeros(ORDER) for _in range(NUM_NOTES)]`.
  - **Inputs**: A list of input vectors `x_list` is used, where each vector x_k receives a discrete impulse ($x_k[0] = 10000.0$) when its corresponding key is pressed.
- **Key Mapping and Tracking**:
  - A dictionary `KEY_MAP` links keyboard keys to the note index $k \in [0, 11]$.
  - The single boolean `KEYPRESS` is replaced by a dictionary `KEYPRESS_NOTES` that tracks which specific note index k requires a sound-triggering impulse in the current processing block.
- **The Main Audio Loop**: The single filtering operation is replaced by a `for` loop that iterates from $k = 0$ to 11.
  1. An empty array, `y_total = np.zeros(BLOCKLEN)`, is initialized to accumulate the output.
  2. Inside the loop, the k-th note's input x_k is impulsed if its key was pressed.
  3. The function `signal.lfilter` is called using the k-th filter's coefficients (b_k, a_k) and its unique state vector (`states_list[k]`).
  4. The output y_k is added to the total: `y_total = y_total + y_k`.
  5. After the loop, `y_total` is clipped and written to the audio stream.

Addendum: Full implementation

```python
import pyaudio
import numpy as np
from scipy import signal
from math import sin, cos, pi
import tkinter as Tk

BLOCKLEN = 64    # Number of frames per block
WIDTH = 2        # Bytes per sample
CHANNELS = 1     # Mono
RATE = 8000      # Frames per second

MAXVALUE = 2**15 - 1# Maximum allowed output signal value (because WIDTH = 2)

NUM_NOTES = 12  # Full octave
f0 = 440.0      # Base frequency (Middle A)
Ta = 1.0        # Decay time (seconds)
ORDER = 2       # Filter order (second-order IIR)

# Calculate frequencies for the full octave (f_k = 2^(k/12) * f0)
frequencies = [f0 * (2**(k/12.0)) for k in range(NUM_NOTES)]

# Pole radius, kept same for all notes
r = 0.01**(1.0/(Ta*RATE))       # 0.01 for 1 percent amplitude

# Initialize lists for filter coefficients, states, and input signal for each
    note
a_list = []
b_list = []
states_list = [np.zeros(ORDER) for _ in range(NUM_NOTES)]  # Independent states
    for each filter
x_list = [np.zeros(BLOCKLEN) for _ in range(NUM_NOTES)]    # Independent input
    signal for each filter

# Calculate the filter coefficients (a_k, b_k) for each note
for f in frequencies:
    om = 2.0 * pi * f / RATE
    # Filter coefficients (second-order IIR)
    a_k = [1, -2*r*cos(om), r**2]
    b_k = [r*sin(om)]
    a_list.append(a_k)
    b_list.append(b_k)

# Map keyboard keys to notes (k=0 to k=11)
KEY_MAP = {
    'a': 0, 'w': 1, 's': 2, 'e': 3,
    'd': 4, 'f': 5, 't': 6, 'g': 7,
    'y': 8, 'h': 9, "u": 10, 'j': 11
}
# Open the audio output stream
p = pyaudio.PyAudio()
PA_FORMAT = pyaudio.paInt16
stream = p.open(
        format      = PA_FORMAT,
        channels    = CHANNELS,
        rate        = RATE,
        input       = False,
        output       = True,
        frames_per_buffer = BLOCKLEN)
# specify low frames_per_buffer to reduce latency

CONTINUE = True
# KEYPRESS is no longer a single boolean; we track which note was pressed.
KEYPRESS_NOTES = {} # Stores {k: True} for notes that need an impulse

def my_function(event):
    global CONTINUE
    global KEYPRESS_NOTES
    print('You pressed ' + event.char)
    if event.char == 'q':
      print('Good bye')
      CONTINUE = False

    if event.char in KEY_MAP:
        # Mark the note index k to be played in the next block
        note_index = KEY_MAP[event.char]
        KEYPRESS_NOTES[note_index] = True
        print(f'Playing note k={note_index} at {frequencies[note_index]:.2f} Hz')


root = Tk.Tk()
root.bind("<Key>", my_function)

print('Press "q" to quit')
print('Press keys for sound. (Keys: ' + ', '.join(KEY_MAP.keys()) + ')')

while CONTINUE:
    root.update()

    # Initialize the total output signal for the current block
    y_total = np.zeros(BLOCKLEN)

    # Loop through all 12 notes (filters)
    for k in range(NUM_NOTES):

        # 1. Check if an impulse is needed for this note's filter
        if k in KEYPRESS_NOTES and KEYPRESS_NOTES[k]:
            # Apply impulse to the input of the k-th filter at the start of the block
            x_list[k][0] = 10000.0    # Use index 0 for the impulse (as in original demo)
            KEYPRESS_NOTES[k] = False  # Reset the keypress flag

        # 2. Filter the input signal x_k using the k-th filter's coefficients (a_k, b_k) and states
        [y_k, states_list[k]] = signal.lfilter(b_list[k], a_list[k], x_list[k], zi = states_list[k])

        # 3. Sum the output of the k-th filter to the total output
        y_total = y_total + y_k

        # 4. Reset the impulse in the input for the next block
        x_list[k][0] = 0.0         # Reset the input for the next block

    # The original logic for a single note (commented out):
    # if KEYPRESS and CONTINUE:
```

```
    #     # Some key (not 'q') was pressed
    #     x[0] = 10000.0
    # [y, states] = signal.lfilter(b, a, x, zi = states)
    # x[0] = 0.0
    # KEYPRESS = False
    # y = np.clip(y, -MAXVALUE, MAXVALUE)     # Clip
    # y_16bit = y.astype('int16')
    # y_bytes = y_16bit.tobytes()
    # stream.write(y_bytes, BLOCKLEN)

    # Use the summed output signal y_total
    y_total = np.clip(y_total, -MAXVALUE, MAXVALUE)

    y_16bit = y_total.astype('int16')
    y_bytes = y_16bit.tobytes()
    stream.write(y_bytes, BLOCKLEN)

print('* Done.')

stream.stop_stream()
stream.close()
p.terminate()
```

Snippet 1: Polyphonic Difference Equation Synthesis Code