

Digital Filtering:

TD 03: IIR filters

Antonin NOVAK

Laboratoire d'Acoustique de l'Université du Maine, Le Mans, France



Goal:

IIR (infinite impulse response) filters

- z-transform (from time to frequency domain and back)
- design in Python
- implementation in Sigma Studio (build-in and general)

feedback in Sigma Studio

- learn how a feedback works
- implement a 1st order IIR filter with add, gain, delay, and feedback



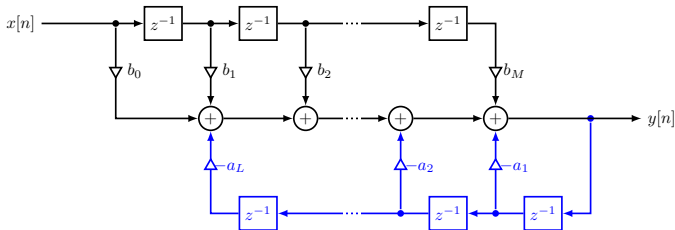
1 IIR filters refresh

Feedback in Sigma Studio

IIR filters refresh

IIR (infinite impulse response) filter

- has **delayed feedback(s)**
- has poles, usually have zeros
- $y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + \dots + b_Mx[n-M]$
 $-a_1y[n-1] - a_2y[n-2] - \dots - a_Ly[n-L]$



z-transform refresh

$$\text{delay by one sample} \iff z^{-1}$$

$$y[n] \iff Y(z)$$

$$y[n-1] \iff Y(z) z^{-1}$$

z-transform refresh

$$\text{delay by one sample} \iff z^{-1}$$

$$y[n] \iff Y(z)$$

$$y[n-1] \iff Y(z) z^{-1}$$

FIR filter (z-transform)

$$y[n] = 0.5 x[n] + 0.5 x[n-5]$$

$$Y(z) = 0.5 X(z) + 0.5 X(z) z^{-5}$$

$$H(z) = \frac{Y(z)}{X(z)} = 0.5 + 0.5 z^{-5}$$

z-transform refresh

IIR filter (z-transform)

$$y[n] - 0.5 y[n - 1] = 0.3 x[n] + 0.1 x[n - 1]$$

$$Y(z) - 0.5 Y(z) z^{-1} = 0.3 X(z) + 0.1 X(z) z^{-1}$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.3 + 0.1 z^{-1}}{1 - 0.5 z^{-1}}$$

$$H(z) = \frac{0.3 z + 0.1}{z - 0.5}$$

$$1 \text{ zero at } z = -\frac{1}{3}, 1 \text{ pole at } z = \frac{1}{2}$$

IIR filters refresh

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + \dots + b_Mx[n-M] \\ - a_1y[n-1] - a_2y[n-2] - \dots - a_Ly[n-L]$$

$$H(z) = \frac{b_0 + b_1z^{-1} + b_2z^{-2} + \dots + b_Mz^{-M}}{1 + a_1z^{-1} + a_2z^{-2} + \dots + a_Lz^{-L}}$$

IIR filter (Butterworth) example

normalized cut-off frequency
 $W_n \in (0, 1)$, corresponding to $(0, f_s/2)$

filter order

`[b, a] = signal.butter(N=2, Wn=0.1, btype='low')`

filter type
(low pass, high-pass, ...)

IIR filter (Butterworth) example

```
# PYTHON code
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt

# filter design
[b, a] = signal.butter(2, 0.1, btype='low')

# calculate the FRF
[F, H] = signal.freqz(b, a, 512, fs=48e3)

# plot the FRF
fig, ax = plt.subplots(2)

ax[0].semilogx(F, 20*np.log10(np.abs(H)))
ax[0].set(ylim=[-50, 10])
ax[0].set(xlabel='Frequency [Hz]')
ax[0].set(ylabel='Magnitude [dB]')
ax[0].grid()

ax[1].semilogx(F, np.angle(H))
ax[1].set(ylim=[-np.pi, 0])
ax[1].set(xlabel='Frequency [Hz]')
ax[1].set(ylabel='Phase [rad]')
ax[1].grid()

plt.show()
```

IIR filters refresh

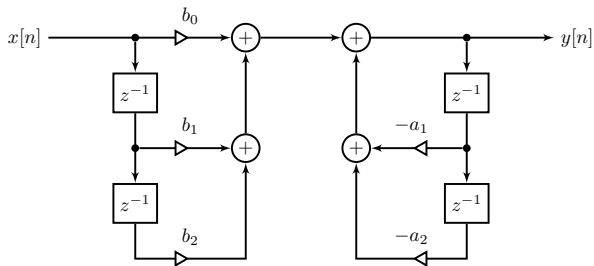
4 basic IIR biquad (2nd order) forms

- Direct Form I
- Direct Form II
- Transposed Direct Form I
- Transposed Direct Form II

Direct Form I

Direct implementation of the difference equation

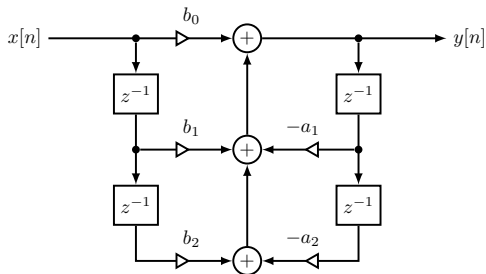
$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]$$



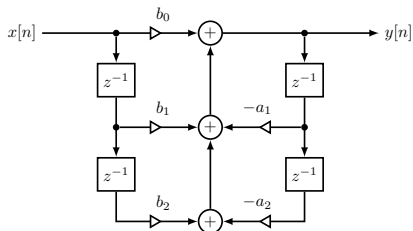
Direct Form I

Direct implementation of the difference equation

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]$$



Direct Form I



PYTHON code

```
# b = [b0, b1, b2]
# a = [1, a1, a2]
```

```
d = [0, 0, 0, 0] # memory buffer
```

```
for n in range(N):
    y[n] = b[0]*x[n] + b[1]*d[0] + b[2]*d[1] + \
          - a[1]*d[2] - a[2]*d[3];
```

```
# memory update
```

```
d[1] = d[0]
d[0] = x[n]
d[3] = d[2]
d[2] = y[n]
```

Advantages

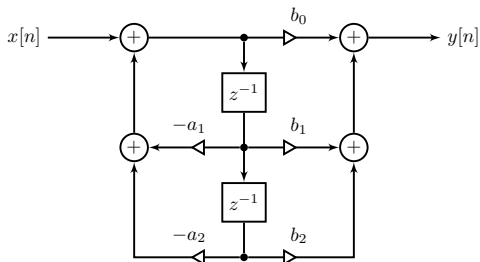
- great for *fixed-point arithmetics*
fixed-point cannot overflow
- optimal for varying coefficients

Disadvantages

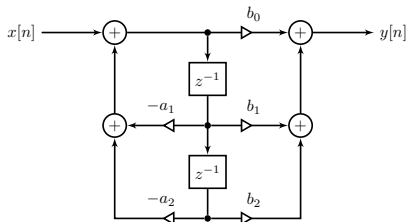
- twice as many delays as are necessary

Direct Form II

Take DF-I exchange the order of poles and zeros to get DF-II



Direct Form II



```
# PYTHON code
# b = [b0, b1, b2]
# a = [1, a1, a2]

d = [0, 0] # memory buffer

for n in range(N):
    v = x[n] - a[1]*d[0] - a[2]*d[1]
    y[n] = b[0]*v + b[1]*d[0] + b[2]*d[1]

    # memory update
    d[1] = d[0]
    d[0] = v
```

Advantages

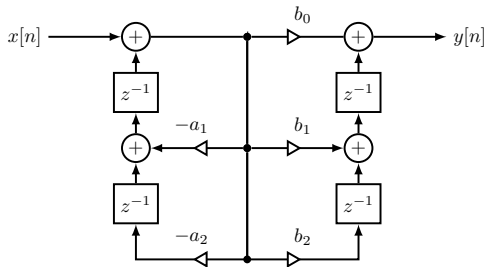
- canonical with respect to delay (memory is shared)

Disadvantages

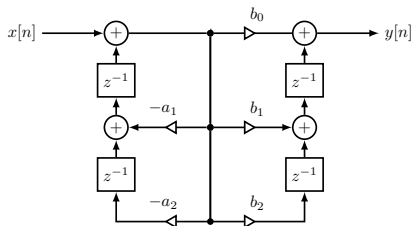
- in *fixed-point* overflow can happen
- not optimal for varying coefficients

Note: Feedback part (poles) usually provide a large signal boost, followed by attenuation in the feedforward part (zeros). Since the poles come first in the DF-II realization, the overflow can easily happen.

Transposed Direct Form I



Transposed Direct Form I



```
# PYTHON code
```

```
# b = [b0, b1, b2]
```

```
# a = [1, a1, a2]
```

```
d = [0, 0, 0, 0] # memory buffer
```

```
for n in range(N):
```

$$v = x[n] + d[0]$$
$$d[0] = -a[1]*v + d[1]$$
$$d[1] = -a[2]*v$$
$$y[n] = b[0]*v + d[2]$$

```
d[2] = b[1]*v + d[3]
```

$$d[3] = b[2]*v$$

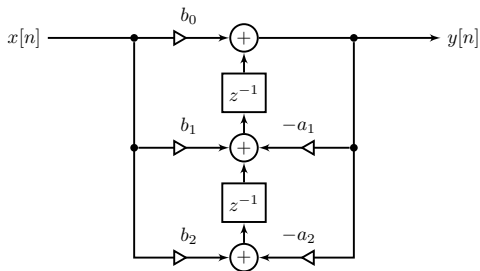
Advantages

- ???

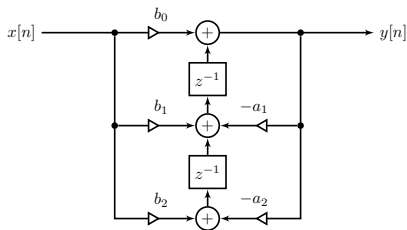
Disadvantages

- twice as many delays as are necessary
- poles goes first (possible overflow)
- not optimal for varying coefficients

Transposed Direct Form II



Transposed Direct Form II



PYTHON code

```
# b = [b0, b1, b2]
# a = [1, a1, a2]
```

```
d = [0, 0] # memory buffer
```

```
for n in range(N):
```

```
    y[n] = b[0] * x[n] + d[0]
```

```
    # memory update
```

```
    d[0] = b[1] * x[n] - a[1] * y[n] + d[1]
```

```
    d[1] = b[2] * x[n] - a[2] * y[n]
```

Advantages

- zeros go first (no feedback in the input)
- canonical with respect to delay
- (memory is shared)

Disadvantages

- not optimal for varying coefficients

other structures

- parallel structures
- serial structures
- lattice-ladder structures
- state-space realization

Work 1: IIR biquad in Python

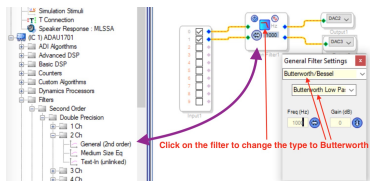
Implement one of the above biquad structures in Python

1. get coefficients `a` and `b` of a 2nd order Butterworth low-pass filter with `Wn=0.1`
2. prepare a Dirac-like signal (length `N=50`)
3. use `scipy.signal.lfilter` to filter the Dirac-like signal
4. implement one of the forms of biquad (see above) and compare the signals filtered using
 - a) `scipy.signal.lfilter`
 - b) biquad implementation

Work 2: IIR filter in Sigma Studio

Create a Sigma Studio project with an IIR 2nd order filter

1. Implement a 2nd order Butterworth filter (2 channel) in Sigma Studio.

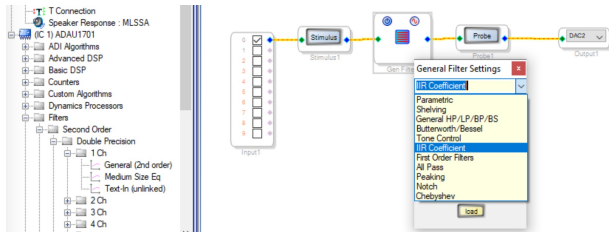


2. Link, compile, and download to the DSP ADAU 1701 and listen to it.
3. Modify the cut-off frequency to listen to the effect of the filter.
4. Change the type of the filter, use Stimuli and Probe to see their FRF and listen to it.

Work 3: From Python to Sigma Studio

Design a 2nd order filter in Python and use it in Sigma Studio

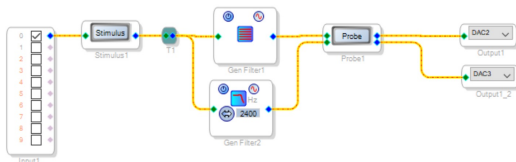
1. Design a 2nd order Butterworth filter in Python
`b, a = signal.butter(2, 0.05)`
2. Export the filter coefficients a and b to a text file in the following order b_0, b_1, b_2, a_1, a_2 (note that $a_0 = 1.0$ is not exported).
In Python use `np.savetxt('coeffs.txt', np.append(b, a[1:]))`.
3. implement a general IIR filter in Sigma Studio with the a and b coefficients loaded from the exported file, add the Probe and Stimuli block to verify it's FRF.



Work 4: Compare filters

In Sigma Studio, compare both previous filters

1. Add a 2nd order Butterworth low-pass filter (1 channel) in Sigma Studio and add it in parallel with the previous filter.
2. Add a T-connection after the Stimulus to split the signal to both filters.
3. Extend the Probe by adding 1 pin (right click on Probe) and connect both filters to Probe.



4. Check if both FRFs are equal using Probe. If not, change the cut-off frequency of the 2nd order Butterworth low-pass filter so that it matches the one designed from Python.



IIR filters refresh

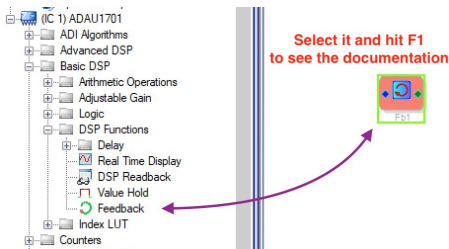
2

Feedback in Sigma Studio

Work 5: Implementation of IIR using feedback

Implement the following 1st order IIR filter: $H(z) = \frac{1 + \alpha}{1 + \alpha z^{-1}}$, for $\alpha \in (-1, 0)$

1. Express the coefficients a and b as a function of α .
2. In Python, plot the FRF of the filter using `scipy.signal.freqz` for different values of α
3. Implement this filter in Sigma Studio without the block IIR. Use `sum`, `gain`, `delay` and `feedback`.



Note, that `feedback` includes a delay by 1 sample !!!

Project

Start working on the project.

The project assignment is available on
<https://imdea.ant-novak.com/DF>.