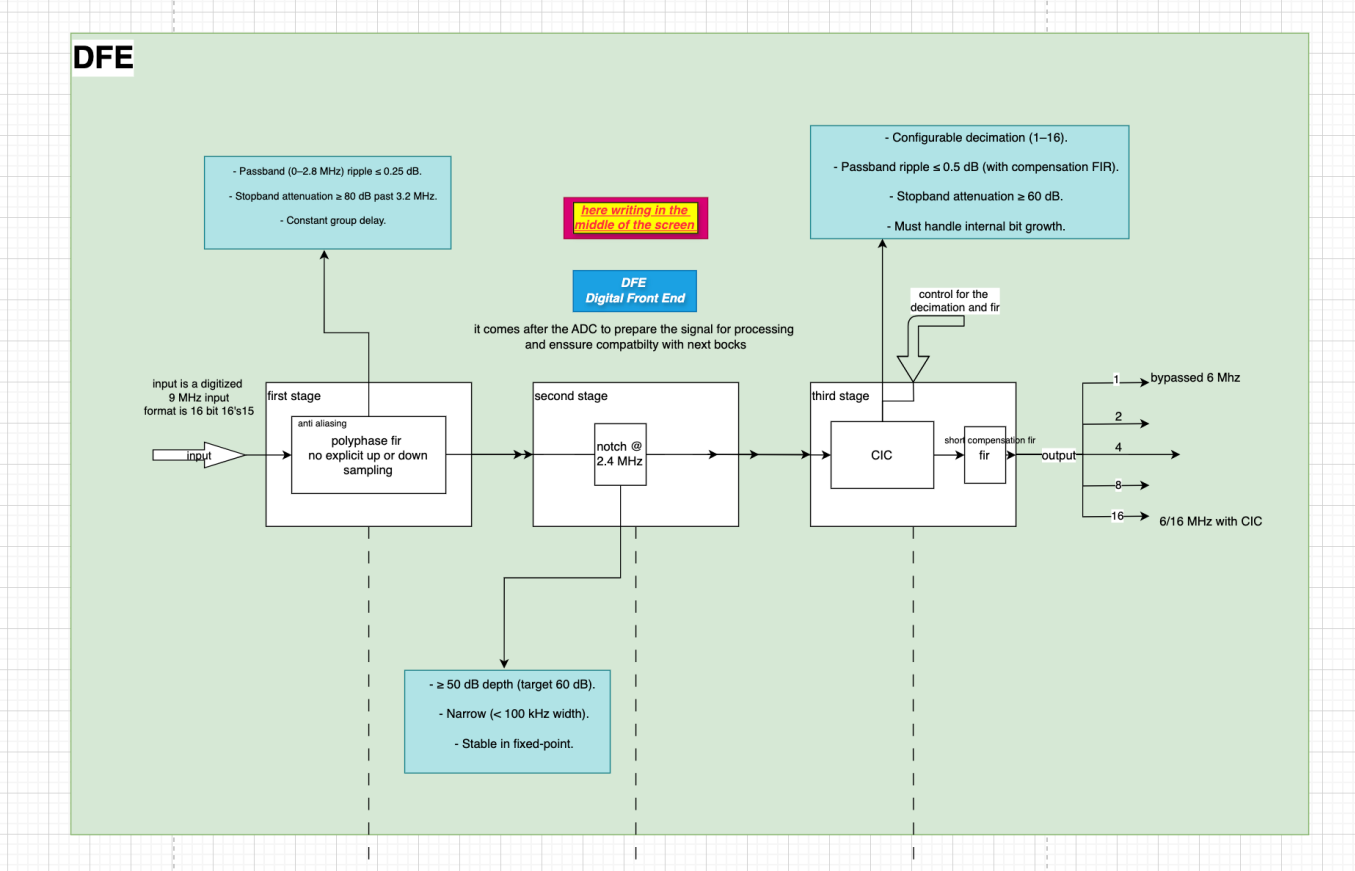
Golden Model report

Hazem Yasser mahmoud

Mohamed Ahmed Mohamed

Muhammad Naim



the golden model is split into two Jupyter notebooks filterdesign.ipynb & goldenmodel.ipynb

general notes

> **problem_1** : document state an interference at 5 MHz while the sampling frequency is 9 MHz. so max frequency component is 4.5 and if present it will alias to 4 MHz.
> and with further downsampling to 6 MHz it will fold even around 3 MHz.

> **problem_2** : for rational resampling the prototype filter cuttof frequncy $w_c = \frac{1}{\max(L,M)} \cdot \frac{F_s}{2}$, so filter specs are wrong

> **resolution** : change specs from 3 to 1.5 so same specs but frequency is divided by two when implementing this prototype at polyphase structure it gives effective filtering at 3.

the golden model is split into two Jupyter notebooks filterdesign.ipynb and goldenmodel.ipynb

steps

1. first use python for design of the FIR IIR and CIC and export coefficients
2. use python to analyze stability of fixed point approximation of the coeff and use s1.23 instead of s1.15 if needed and for IIR coefficients you need to use s2.14 as coefficients are bigger than 1
3. design polyphase filter for rational resampling realizing a filter at a lower sampling freq is easier and less resource intensive .
4. use biquad IIR filter to realize the notch filter and DF II Transposed also use multiple cascaded IIR if needed we will filter only at 2.4 MHz.
5. design and visualize the proper CIC and short compensation filter
6. design a general abstract high level model
7. import coefficients to python and build the functions that do the same functions in low level without abstracting any details
8. test the model in Jupyter notebook and Matlab by resampling a signal at 9 MHz and injecting noise at 6 MHz.

fractional decimator

we will use polyphase fir filter to sample at only the needed ouput phase for efficent computaion

1. Naive way

The straightforward way is:

1. Upsample by 2 → insert zeros → sampling rate goes 9 → 18 MHz.
2. Low-pass filter at 1.5 MHz (to remove images).
3. Downsample by 3 → pick every 3rd sample → final rate is 6 MHz.

- Issue:
  - The filter runs at the high intermediate rate (18 MHz), so a lot of wasted multiply-accumulate (MAC) operations on zeros.
  - High computational cost, especially if the filter has many taps.

2. Polyphase way

With polyphase decomposition, you restructure the filter so you:

- Never actually insert zeros.
- Only compute the output samples that survive after downsampling.
- Exploit symmetry between the up/down steps.
  For L/M resampling (L=2, M=3):
- The polyphase filter has M = 3 branches (because of downsampling factor).
- Each branch only processes the needed input samples for one output stream.
- So instead of filtering at 18 MHz, the effective computation happens closer to the final 6 MHz rate.

notes

in implemneting the up and down sampler we use mux and demux

we are free to discard the zero-valued replacement samples. In reality we do not insert the zero-valued samples since they are immediately discarded.

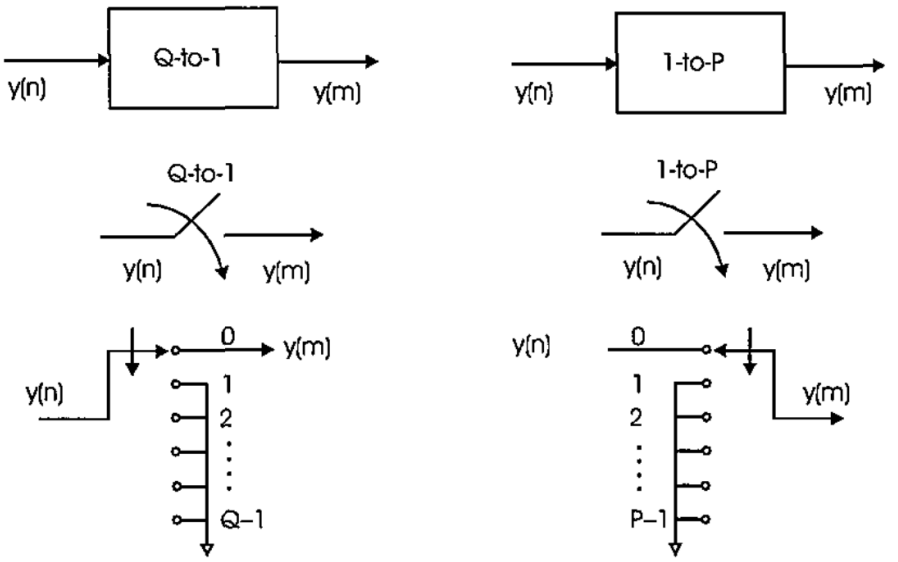
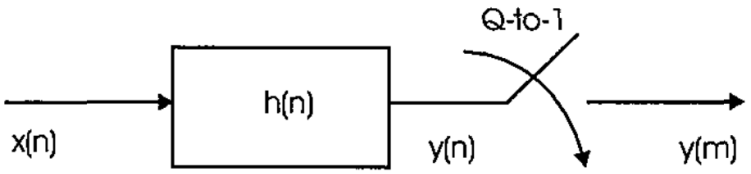


Figure 2.8 Symbols Representing Down Sampling and Up Sampling Elements



- identities used to compute polyphase filter efficently

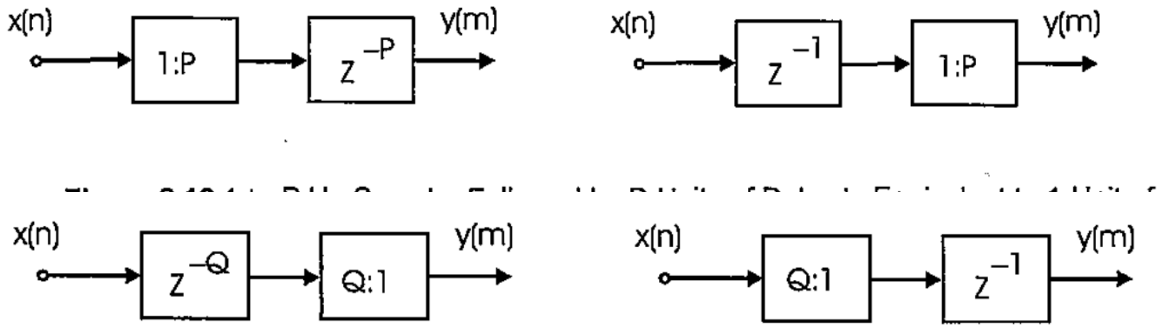


Figure 2.15 Q-units of Delay and Q-to-1 Down Sampler Is Equivalent to Q-to-1 Down Sampler and 1-unit of Delay.

Design

python scipy

we will use Python to design a FIR filter at 9 MHz sampling rate and we will optimise the taps to get the best fir that meets

- -80 dBs stopband at 1.6 MHz.
- < .25 dBs passband at 1.4 MHz.
- a Gain of L = 2 to normalize after upsampling.

for best optmized LPF FIR with steep freq response we will use kaiser window

- Kaiser order estimate:

$N \approx \left\lceil \frac{A-8}{2.285\ \Delta\omega} \right\rceil = \left\lceil \frac{80-8}{2.285 \times 0.1396} \right\rceil \approx 226$

```
Number of taps = N+1 = 227
```

Problems

a prototype filter at 3 MHz passpand has effective filtering at 6 MHz when implemented in polyphase resampler

resolution: it was a specs error as all materials state to filter up to
$w_c = \frac{1}{\max(L,M)} \cdot \frac{F_s}{2}$ when resampling

- so filter should be at 1.5 MHz so the prototype filter was designed as such
- at 1.4 ripple less than .25dB and at 1.6 less than -80dB

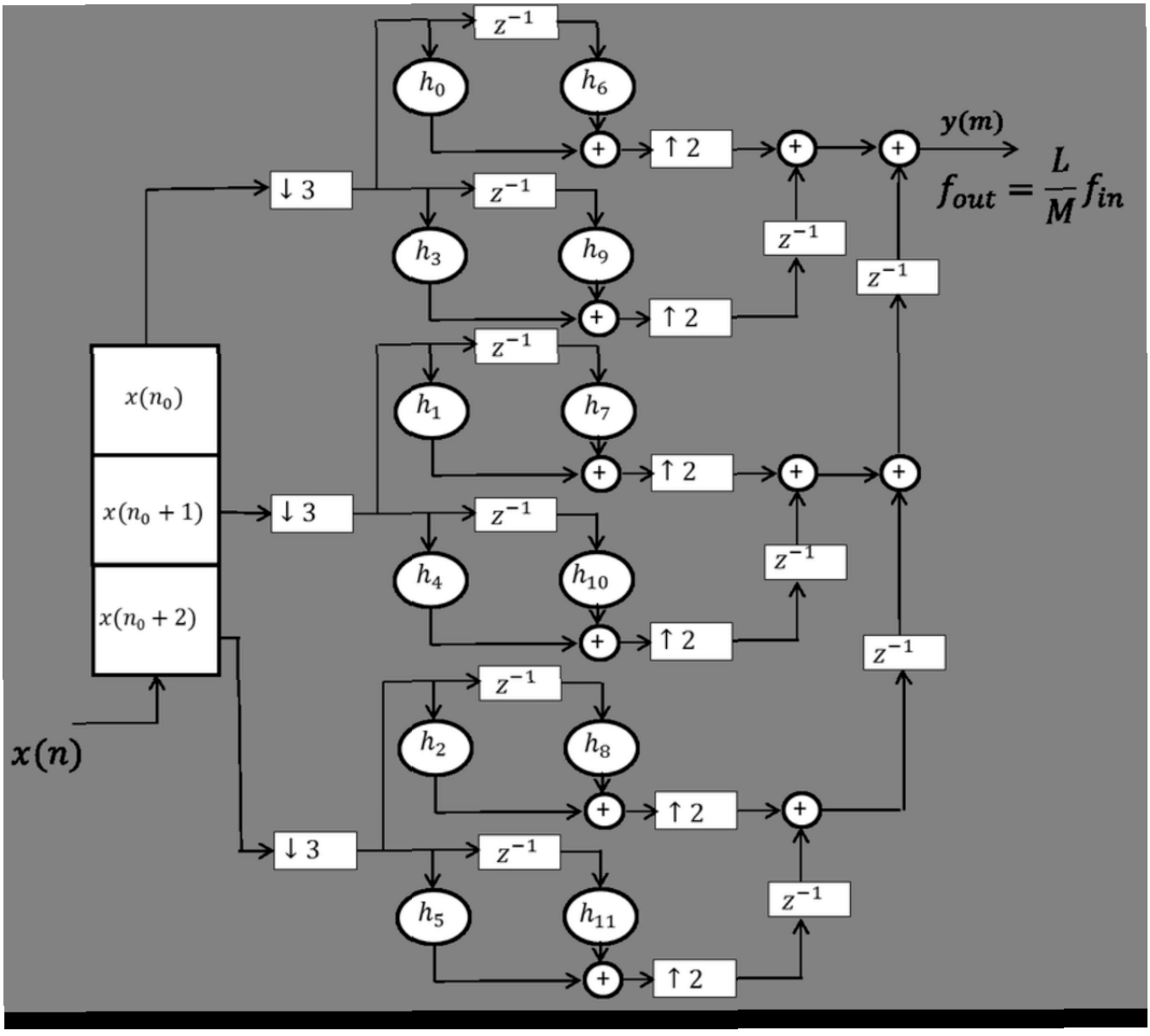
> observation : after implementing the fir in polyphase structure
> it is filtering upto 3 MHz it seems that the structure has a frequency doubling eeffect

implementaions

First approach

then we will implement it in polyphase structure to achieve resampling effects
we will use pipelined delay lines after multiplier or pipelined filter phases to achieve better timing.

this polyphase structure was proposed in a paper
https://www.researchgate.net/profile/Abhishek-Kumar-202/publication/319763113



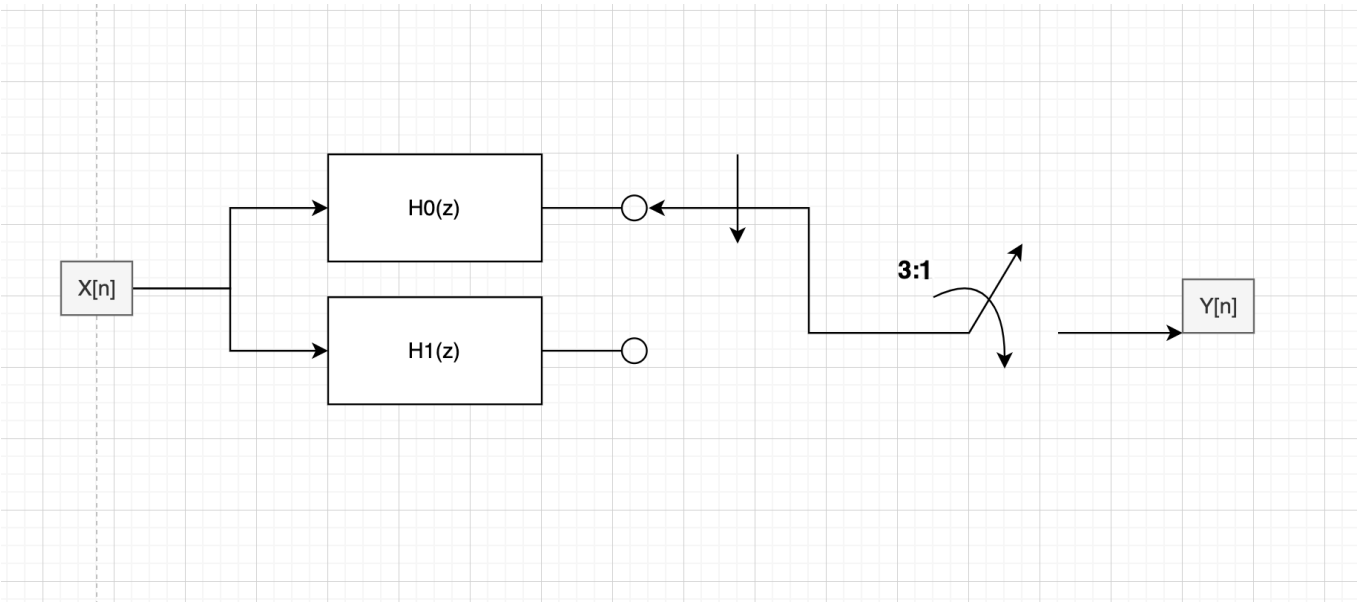
- three main branches for downsampling
- 2 secondary branches for upsampling
- can replace all the delays lines by muxes and demux

> note that each branch has more coeff than in the image each sybsequent two have 6 in difference so h0,h6,h12,h18....
> h1,h7,h13,h19....

while complex it uses more effiecnt computations and thus less delay considerably less
also much better when L > M than the traiditional polyphase structure such as the one used in matlab

second approach

using the traditional structure will be easier to convert to HDL with hdlcoder in matlab
not as optimized as the paper proposed but easier to implement



IIR Notch Filter Design at 6 MHz Sampling

> After resampling, the maximum frequency component is 3 MHz, so we only need to design a notch filter centered at 2.4 MHz to remove unwanted interference.

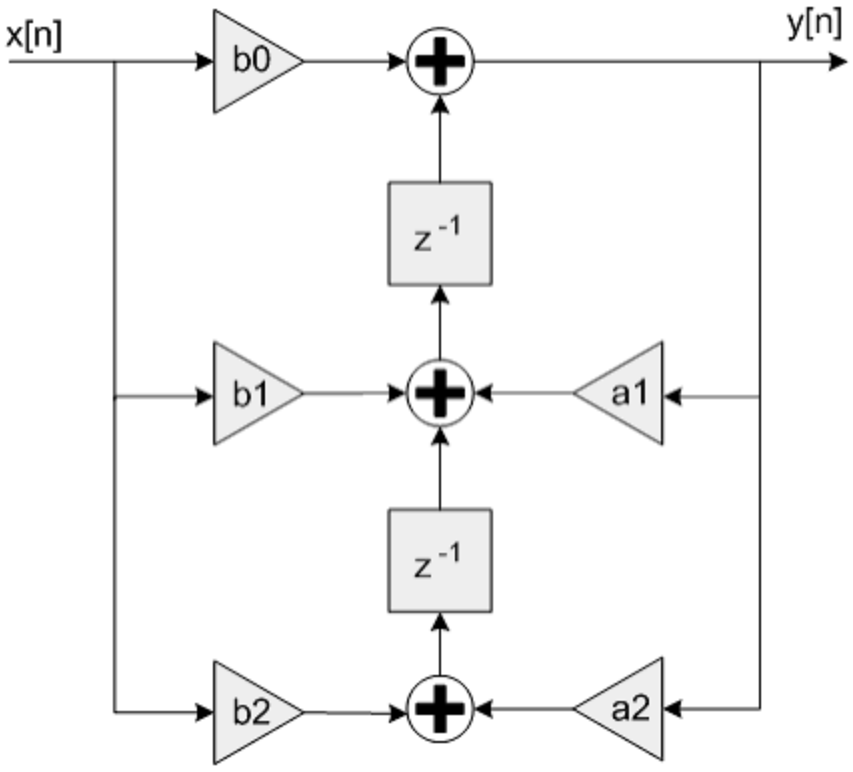
Design Steps

- We design a second-order IIR notch filter with $\Delta f = 100\,\text{kHz}, F_s = 6\,\text{MHz}, and F_0 = 2.4\,\text{MHz}.$
- The quality factor is chosen as Q = 30 to achieve a deep attenuation of approximately 60 dB, while maintaining stability after fixed-point quantization.

- The pole radius is given by $r = e^{-\pi \frac{\Delta f}{F_s}}$ ensuring a narrow bandwidth around F_0.
- The transfer function of the biquad notch filter is: $H(z) = \frac{1 - 2\cos(\omega_0) z^{-1} + z^{-2}}{1 - 2r\cos(\omega_0) z^{-1} + r^2 z^{-2}}$ where $\omega_0 = 2\pi \frac{2 * F_0}{F_s}$.
- The filter coefficients b_i and a_i are then quantized to Q2.14 format for hardware implementation the quantization is not in Q1.15 because values exceed one.

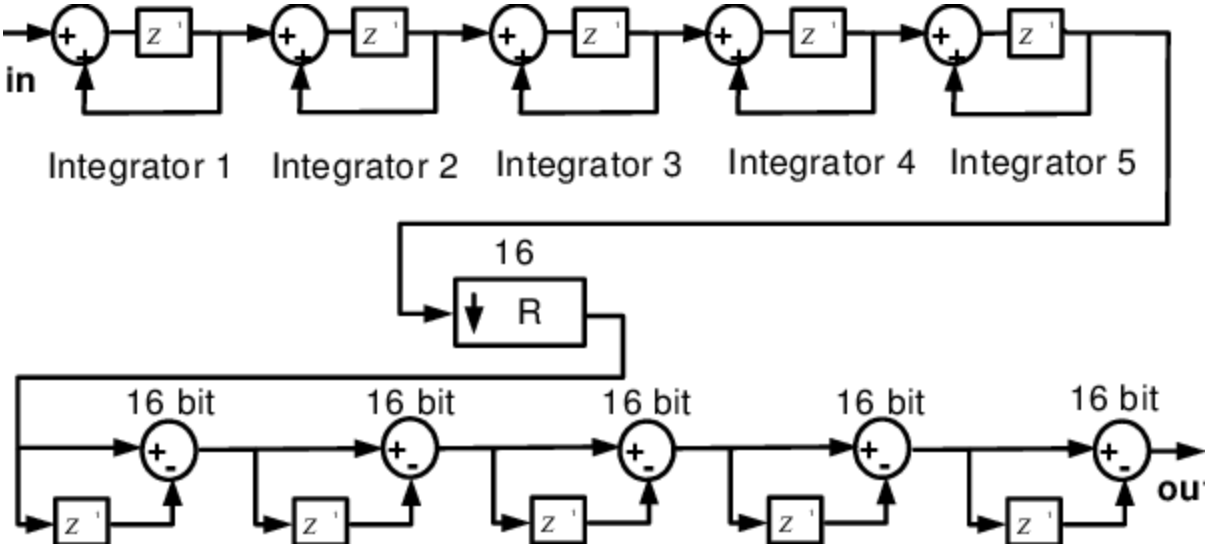
```
later in implementation  in HDL we need to account for change in format
between first and second stage and design multipy and add accordingly
```

- After quantization, we evaluate:
  - The frequency response, to confirm the −60 dB notch depth near 2.4 MHz.
  - The impulse response, to verify correct transient behavior.
  - The pole-zero diagram, to ensure all poles remain inside the unit circle (stability).
- The filter is implemented using the Direct Form II Transposed (DFII-T) structure, chosen for numerical efficiency and good fixed-point behavior.



.....

CIC Decimator

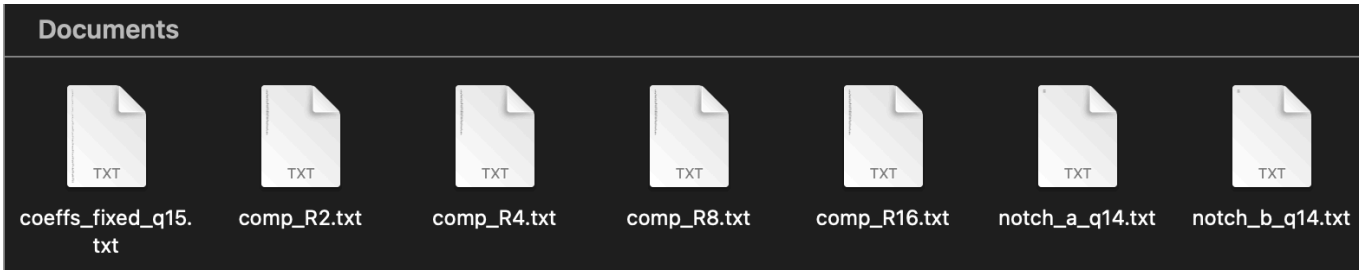


steps

- Design N-stage CIC filter with decimation factor R and differential delay M.
- Use only adders and delays (no multipliers) for efficiency.
- Compensate passband droop with a short FIR filter loaded from comp_R{R}.txt
- Scale output by theoretical gain G = (R*M)^N to normalize amplitude
- Implement in Python first, then test with single-tone and multi-tone signals

1. N and M chosen to be 5 and 1
2. R is configurable from 1,2,4,8,16
3. when R is 1 the signal bypass the CIC altogether
4. the CIC is followed by an fir for the droop

all filter coefficients are quantized and stored in txt files



Filter Design (FIR ,IIR ,CIC)

by :

Hazem Yasser Mahmoud , Group : 7

Mohamed Ahmed Elsayed

mohamed Naem

imports

In [ ]:
```
import numpy as np
from scipy.signal import firwin, remez, freqz, kaiserord, iirnotch, lfilter,
import matplotlib.pyplot as plt
```

Polyphase filter for rational resampling

Promplems and Solutions

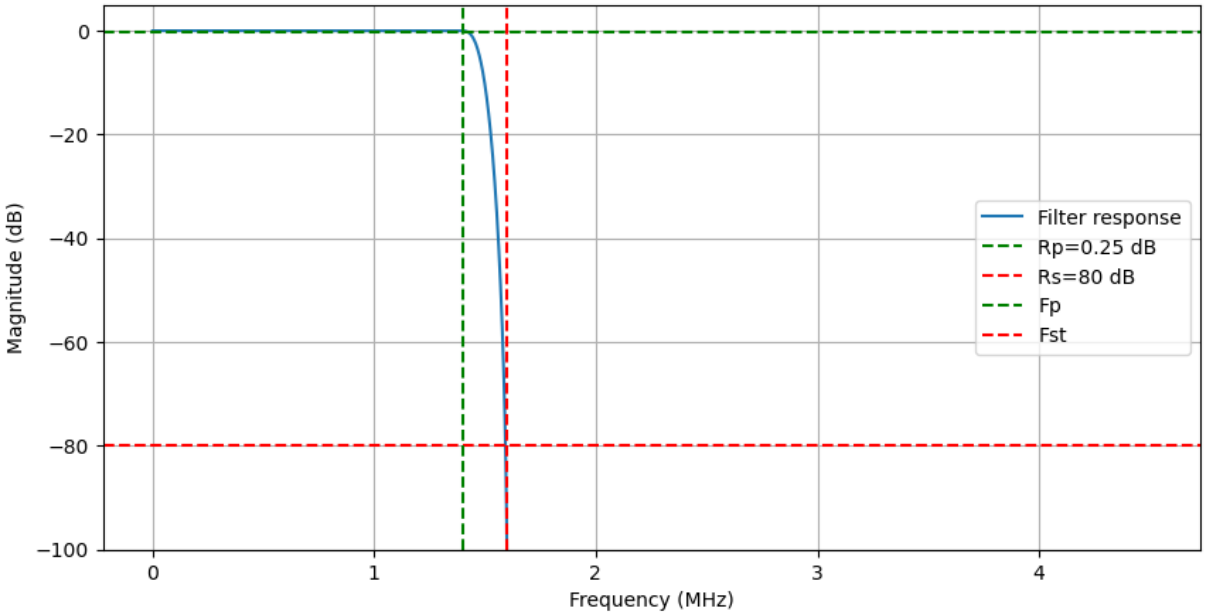
1. For prpoer resampling the fir filter need to be $w_c = \frac{1}{\max(L, M)} \cdot \frac{F_s}{2}$
2. so we Design a Prototype FIR at 1.5 wher 1.4 lesss than .25dB and after 1.6 less than -80dB
3. implementing this prototype FIR in polyphase structure give effective filtering at 3 MHz

DESIGN

In [ ]:
```
# Specs
Fs  = 9e6
Fp  = 1.4e6
Fst = 1.6e6

# Desired ripple/attenuation
Rp = 0.25   # passband ripple in dB
Rs = 80     # stopband attenuation in dB

# Convert to linear ripple/attenuation
delta_p = (10**(Rp/20) - 1) / (10**(Rp/20) + 1)   # passband ripple (linear)
delta_s = 10**(-Rs/20)                            # stopband ripple (linear)
```

```
# Relative weights: higher weight on passband or stopband as needed
# (choose ratio based on importance)
weights = [1/delta_p, 1/delta_s]

# Bands in Hz
bands   = [0, Fp, Fst, Fs/2]   # passband 0–Fp, stopband Fst–Fs/2
desired = [1, 0]               # gain in each band

# Estimate number of taps (Kaiser formula as starting point)

N, beta = kaiserord(Rs, (Fst-Fp)/(Fs/2))
numtaps = N if N % 2 else N+1  # odd length

print(f"Estimated filter order: {numtaps}")

# Design equiripple filter
h = remez(numtaps, bands, desired, weight=weights, fs=Fs)

# Frequency response
w, H = freqz(h, worN=8192, fs=Fs)
H_db = 20*np.log10(np.maximum(np.abs(H), 1e-12))

# Plot
plt.figure(figsize=(10,5))
plt.plot(w/1e6, H_db, label="Filter response")
plt.axhline(-Rp, color="g", linestyle="--", label="Rp=0.25 dB")
plt.axhline(-Rs, color="r", linestyle="--", label="Rs=80 dB")
plt.axvline(Fp/1e6, color="g", linestyle="--", label="Fp")
plt.axvline(Fst/1e6, color="r", linestyle="--", label="Fst")
plt.xlabel("Frequency (MHz)")
plt.ylabel("Magnitude (dB)")
plt.ylim([-100, 5])
plt.grid()
plt.legend()
plt.show()
```

```
Estimated filter order: 227
```



export to quantized

In [5]:
```
nbits = 16
scale = 2**(nbits - 1)

h_fixed = np.round(h * scale)
h_fixed = np.clip(h_fixed, -scale, scale - 1).astype(np.int16)

np.savetxt("coeffs_fixed_q15.txt", h_fixed, fmt="%d")
```

biquad IIR Notch at 2.4 MHz

Promplems and Solutions

1. original sampling frequency is 9 MHz so Max freq component Present is 4.5 MHz
2. any frequency higher would have been either filtered by the previous filter

or aliased around the 4.5 so an interfernce at 5 would be at 4 if not filtered 3. after the resampling filter the original interfernce at 4 (previously 5) is already filtered 4. so we design only for the IIR at 2.4

numerical stabilty

- hor numerical stabilty we design biquad iir and implement DFII

Design

In [8]:
```
# ============================================================
# 1. Design notch filter (floating point)
# ============================================================
```

```
fs = 6e6        # Sampling frequency
f0 = 2.4e6      # Notch frequency
Q = 30          # Quality factor

# Design IIR notch (2nd order)
b, a = iirnotch(f0/(fs/2), Q)

print("Floating-point coefficients:")
print("b =", b)
print("a =", a)
```

```
Floating-point coefficients:
b = [0.95977357 1.55294626 0.95977357]
a = [1.         1.55294626 0.91954714]
```

In [9]:

```
# ================================================================
# 2. Quantize to Q1.14 fixed-point (16-bit signed)
# ================================================================

frac_bits = 14
scale = 2**frac_bits

def float_to_q14(x):
    return np.round(x * scale).astype(int)

def q14_to_float(x):
    return x / scale

b_q14 = float_to_q14(b)
a_q14 = float_to_q14(a)

print("\nQ1.14 integer coefficients:")
print("b_q14 =", b_q14)
print("a_q14 =", a_q14)

# Convert back to float for simulation
b_fixed = q14_to_float(b_q14)
a_fixed = q14_to_float(a_q14)

print("\nQ1.14 quantized coefficients (float equivalent):")
print("b_fixed =", b_fixed)
print("a_fixed =", a_fixed)
```

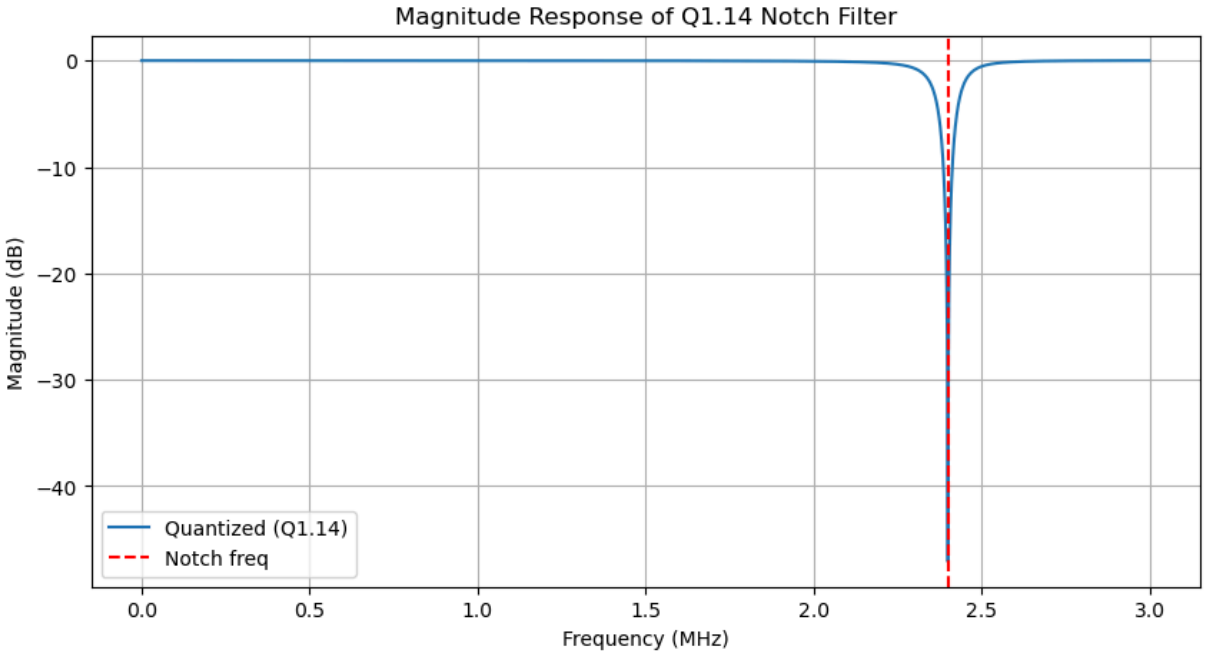
```
Q1.14 integer coefficients:
b_q14 = [15725 25443 15725]
a_q14 = [16384 25443 15066]

Q1.14 quantized coefficients (float equivalent):
b_fixed = [0.95977783 1.55291748 0.95977783]
a_fixed = [1.         1.55291748 0.91955566]
```

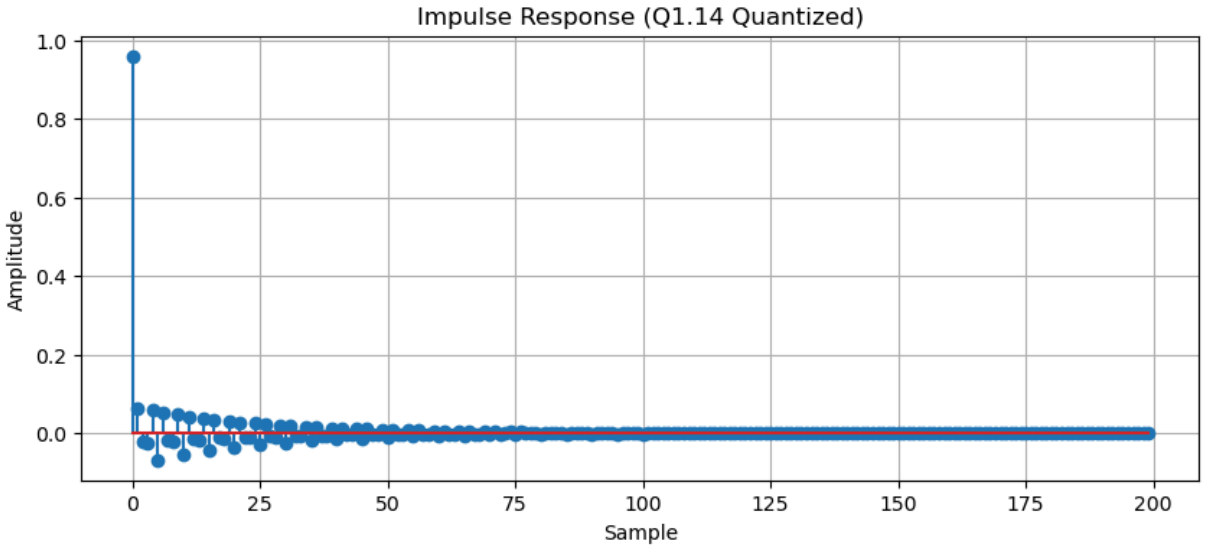
In [10]:

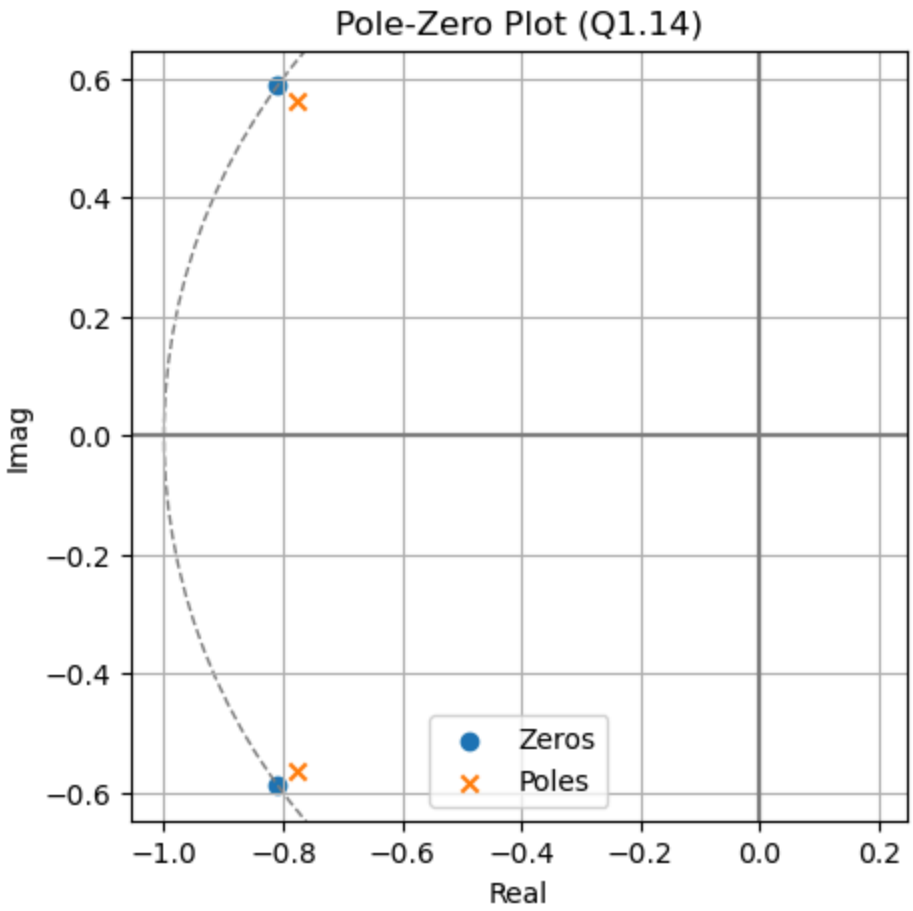
```
# ================================================================
# 3. Frequency response (quantized)
# ================================================================

w, h = freqz(b_fixed, a_fixed, worN=4096, fs=fs)
```

```
plt.figure(figsize=(10,5))
plt.plot(w/1e6, 20*np.log10(np.abs(h)), label='Quantized (Q1.14)')
plt.title('Magnitude Response of Q1.14 Notch Filter')
plt.xlabel('Frequency (MHz)')
plt.ylabel('Magnitude (dB)')
plt.grid(True)
plt.axvline(f0/1e6, color='r', linestyle='--', label='Notch freq')
plt.legend()
plt.show()

# ================================================================
# 4. Impulse response (quantized)
# ================================================================

impulse = np.zeros(200)
impulse[0] = 1.0
resp = lfilter(b_fixed, a_fixed, impulse)

plt.figure(figsize=(10, 4))
plt.stem(resp)
plt.title('Impulse Response (Q1.14 Quantized)')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.grid(True)
plt.show()
# ================================================================
# 5. Pole–zero plot (quantized)
# ================================================================

z, p, k = tf2zpk(b_fixed, a_fixed)

plt.figure(figsize=(5,5))
plt.axhline(0, color='gray')
plt.axvline(0, color='gray')
plt.scatter(np.real(z), np.imag(z), marker='o', label='Zeros')
plt.scatter(np.real(p), np.imag(p), marker='x', label='Poles')
circle = plt.Circle((0,0), 1, color='gray', fill=False, linestyle='--')
plt.gca().add_artist(circle)
plt.title('Pole-Zero Plot (Q1.14)')
plt.xlabel('Real')
plt.ylabel('Imag')
plt.grid(True)
plt.axis('equal')
plt.legend()
plt.show()

# ================================================================
# 6. Check stability
# ================================================================

stable = np.all(np.abs(p) < 1)
print("\nFilter stability after quantization:", "Stable ✅" if stable else "

# ================================================================
# 7. Save fixed-point coefficients (for HDL)
```

```
# ============================================================

np.savetxt("notch_b_q14.txt", b_q14, fmt="%d")
np.savetxt("notch_a_q14.txt", a_q14, fmt="%d")

print("\nSaved coefficients to notch_b_q14.txt and notch_a_q14.txt")
```








```
Filter stability after quantization: Stable ✅

Saved coefficients to notch_b_q14.txt and notch_a_q14.txt
```

CIC decimation filter

CIC Filter Specs:

- Configurable decimation (1–16)

- Passband ripple ≤ 0.5 dB (with compensation FIR)

- Stopband attenuation ≥ 60 dB

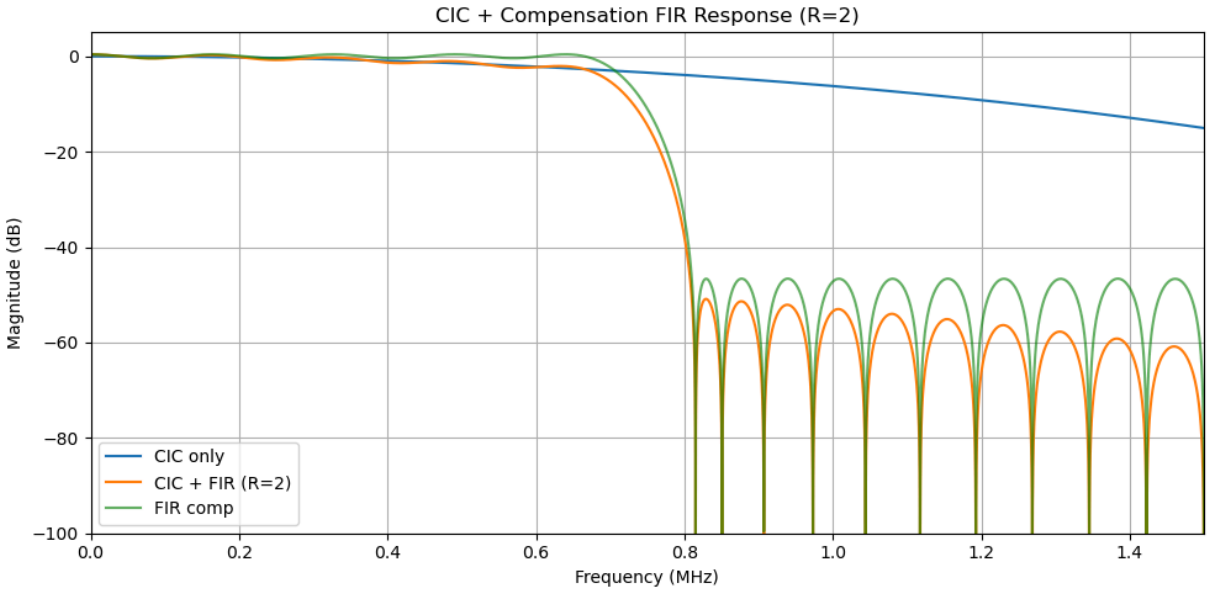
- Must handle internal bit growth

In [15]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import remez, freqz

# ----------------------------
# Parameters
# ----------------------------
fs_in = 6e6         # input sampling rate (Hz)
N = 5               # number of CIC stages
M = 1               # differential delay
numtaps = 40
```

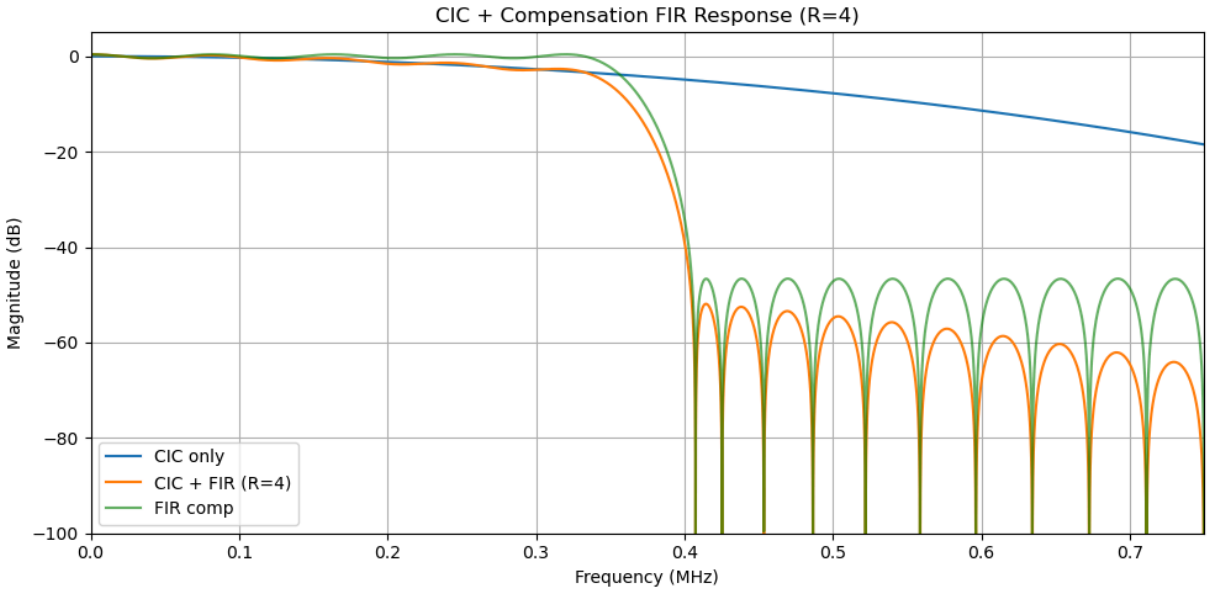
```
nfft = 16384
eps = 1e-12

# Quantization parameters (s.1.15)
N_FRAC = 15
SCALE = 2 ** N_FRAC
MAX_Q = (2 ** (N_FRAC + 1)) - 1

# ----------------------------
# Function to quantize coefficients to Q1.15
# ----------------------------
def quantize_q15(x):
    x = np.clip(x, -1, 1 - 1/SCALE)
    q = np.round(x * SCALE).astype(int)
    return q, q / SCALE

# ----------------------------
# Main Loop for different decimation factors
# ----------------------------
R_values = [2, 4, 8, 16]

for R in R_values:
    print("\n==============================")
    print(f"Designing for R = {R}")
    print("==============================")

    fs_out = fs_in / R
    w_out = np.linspace(0, np.pi, nfft)
    f_out = w_out * fs_out / (2 * np.pi)

    # ----------------------------
    # CIC Response (output-domain)
    # ----------------------------
    num = np.sin(M * w_out / 2.0)
    den = np.sin(w_out / (2.0 * R))
    den[np.abs(den) < eps] = eps
    H_cic_out = (num / (R * M * den)) ** N


    print("CIC diagnostics:")
    print("  CIC DC gain =", ((R * M) ** N))
    print("  finite check: any NaN/Inf? ", np.any(~np.isfinite(H_cic_out)))

    # ----------------------------
    # Compensation FIR Design
    # ----------------------------
    f_pass = 0.45 * (fs_out / 2.0)
    bands = [0, f_pass, f_pass * 1.2, fs_out / 2.0]
    desired = [1, 0]
    weights = [1, 10]

    comp = remez(numtaps, bands, desired, weight=weights, fs=fs_out)
    w_comp, H_comp = freqz(comp, 1, worN=w_out)
    H_total = H_cic_out * H_comp
```

```
    print("  finite check: any NaN/Inf? ", np.any(~np.isfinite(H_comp)))

    # ----------------------------
    # Quantize to Q1.15
    # ----------------------------
    q15_int, q15_float = quantize_q15(comp)
    print(f"  Quantized range: min={q15_int.min()} max={q15_int.max()}")

    # ----------------------------
    # Export coefficients
    # ----------------------------
    filename = f"comp_R{R}.txt"
    np.savetxt(filename, q15_int, fmt="%d")
    print(f"  Saved quantized coefficients to {filename}")

    # ----------------------------
    # Optional Plot
    # ----------------------------
    plt.figure(figsize=(10, 5))
    plt.plot(f_out / 1e6, 20*np.log10(np.abs(H_cic_out)+1e-18), label="CIC o
    plt.plot(f_out / 1e6, 20*np.log10(np.abs(H_total)+1e-18), label=f"CIC +
    plt.plot(f_out / 1e6, 20*np.log10(np.abs(H_comp)+1e-18), color="green",
    plt.title(f"CIC + Compensation FIR Response (R={R})")
    plt.xlabel("Frequency (MHz)")
    plt.ylabel("Magnitude (dB)")
    plt.grid(True)
    plt.legend()
    plt.xlim(0, fs_out/2/1e6)
    plt.ylim(-100, 5)
    plt.tight_layout()
    plt.show()
```

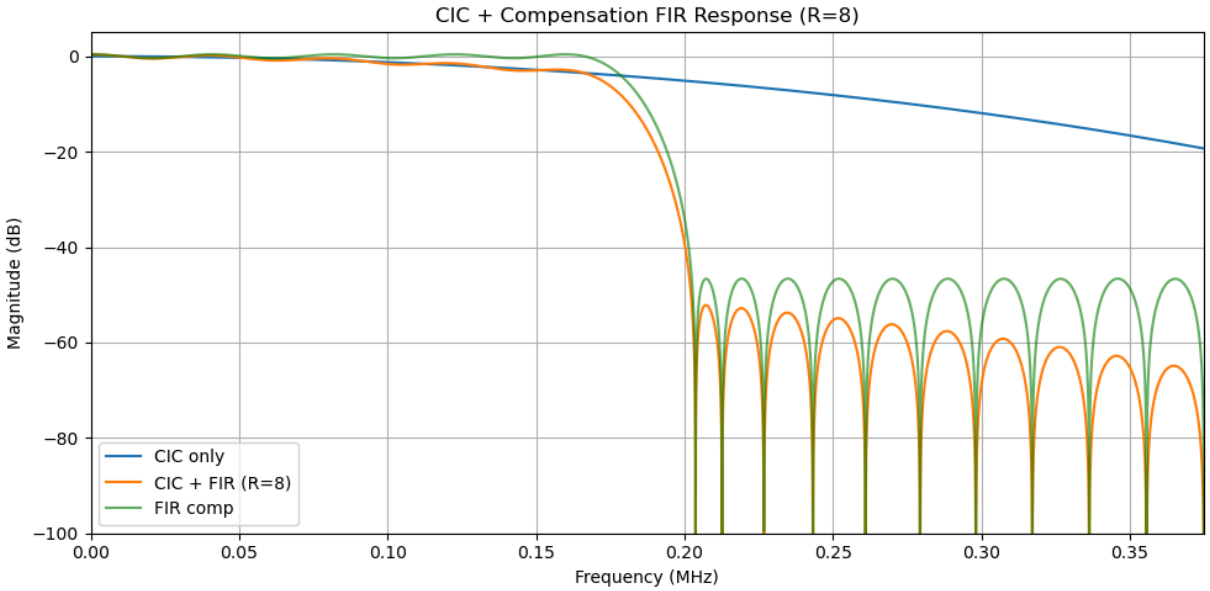
```
==============================
Designing for R = 2
==============================
CIC diagnostics:
  CIC DC gain = 32
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R2.txt
```



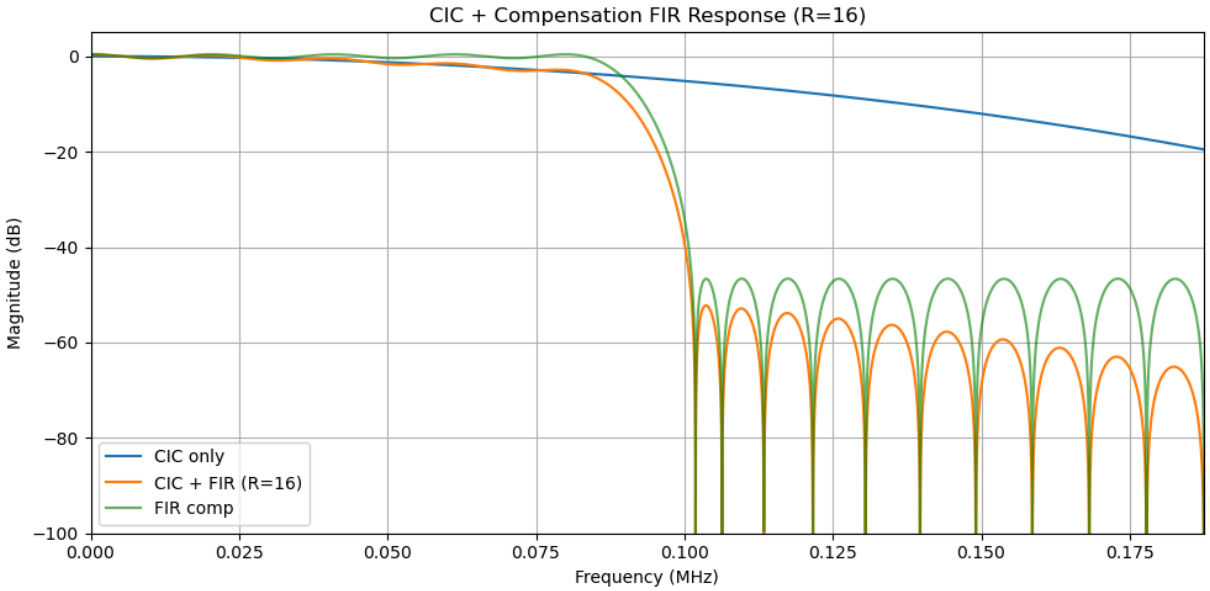

```
==============================
Designing for R = 4
==============================
CIC diagnostics:
  CIC DC gain = 1024
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R4.txt
```




```
==============================
Designing for R = 8
==============================
CIC diagnostics:
  CIC DC gain = 32768
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R8.txt
```




```
==============================
Designing for R = 16
==============================
CIC diagnostics:
  CIC DC gain = 1048576
  finite check: any NaN/Inf?  False
  finite check: any NaN/Inf?  False
  Quantized range: min=-2576 max=14420
  Saved quantized coefficients to comp_R16.txt
```



In [ ]:

Golden Model For DFE

by : Group 7

Hazem Yasser Mahmoud

Mohamed Ahmed Elsayed

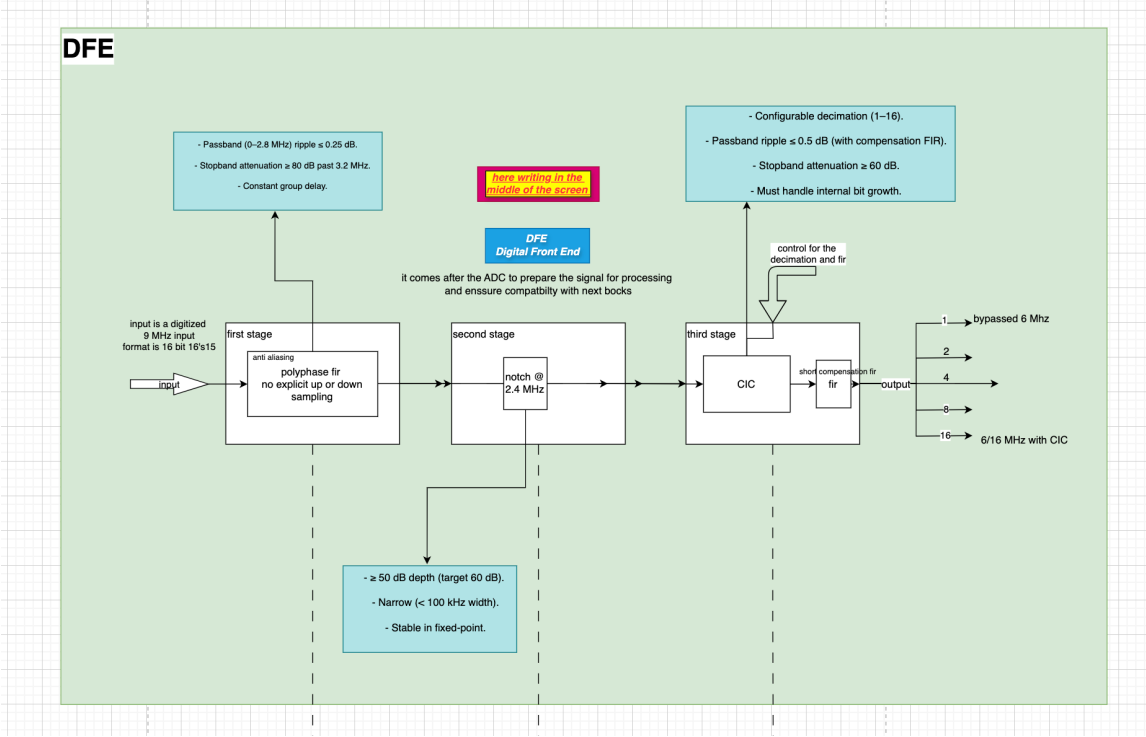
mohamed Naem

imports

In [108…

```
import numpy as np
from scipy.signal import firwin , freqz , resample_poly
import matplotlib.pyplot as plt
```

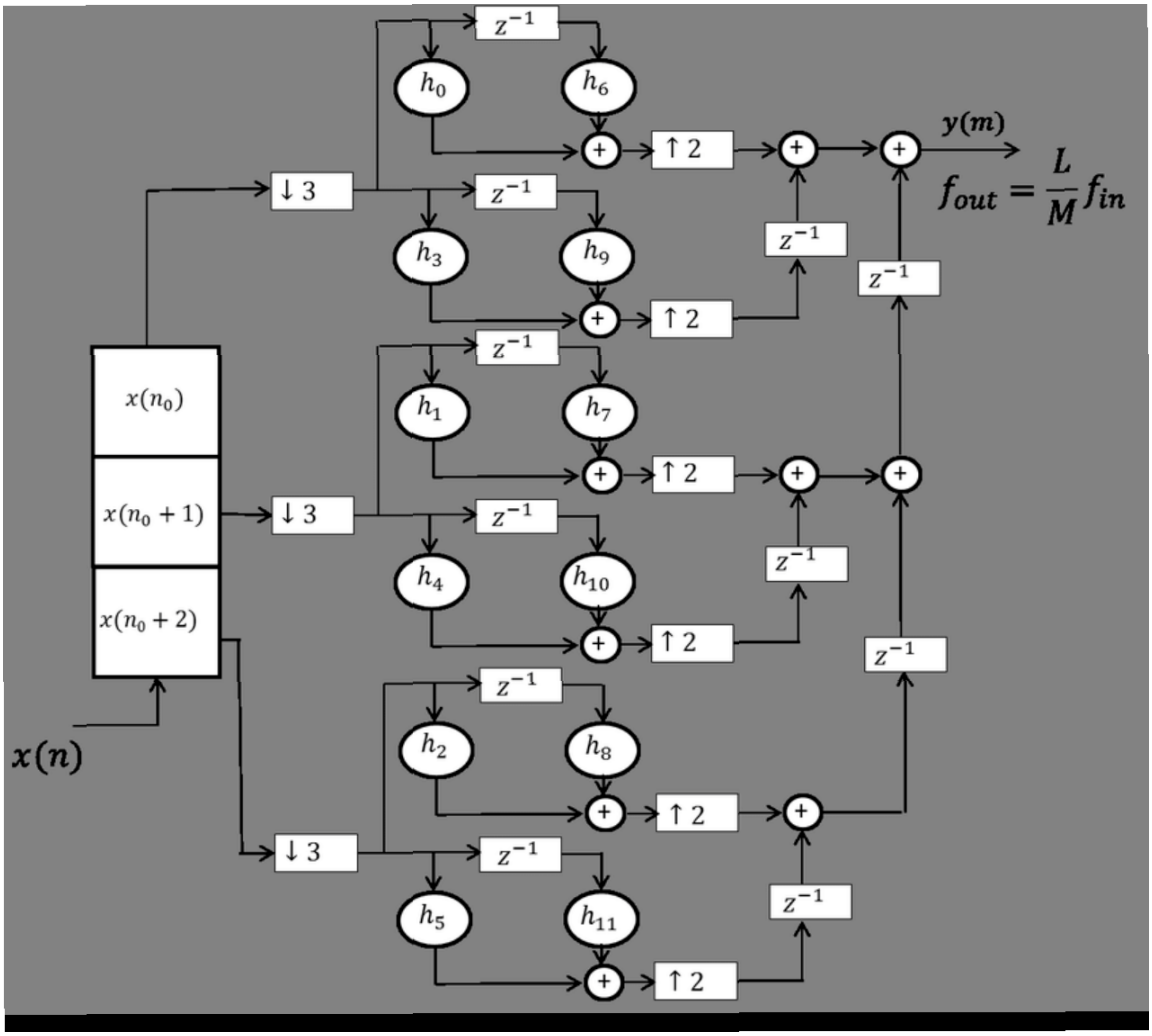
DFE stages



Paper Proposed rational Polyphase resampler structure

note that each branch has more coeff than in the image each sybsequent two have 6 in differnce so h0,h6,h12....

https://www.researchgate.net/profile/Abhishek-Kumar-202/publication/319763113

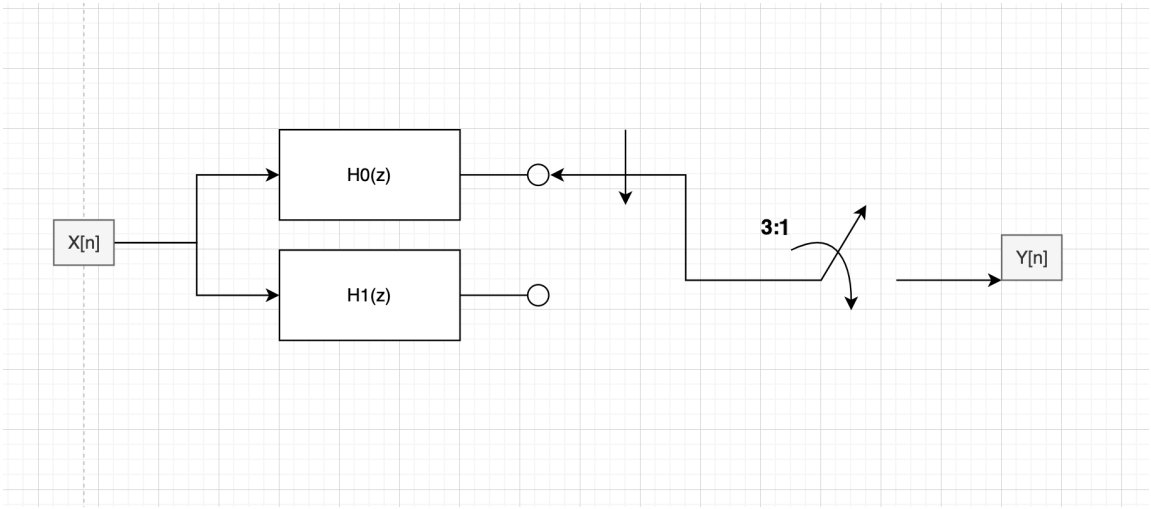


In [109…

```python
def polyphase_resample(x, L, M,h):
    """
    Rational resampler using explicit 3D polyphase structure with
    parallel main phases and interleaved subphases.
    """

    # 2. Polyphase decomposition into M x L branches
    phases = []
    for m in range(M):   # main phases
        row = []
        for l in range(L):  # sub-phases
            row.append(h[m + l*M :: L*M])
        phases.append(row)

    # 3. Split input into M decimated streams (one per main phase)
    x_p = [x[m::M] for m in range(M)]

    # 4. Process each main phase separately
    parallel_outputs = [[] for _ in range(M)]

    for m in range(M):       # main phase
        x_m = x_p[m]
```

```
        h_m = phases[m]

        parallel_outputs[m].extend([0.0] * m)

        for n in range(len(x_m)):
            subphase_samples = []
            for l in range(L):   # subphases
                acc = 0.0
                h_ml = h_m[l]
                for k in range(len(h_ml)):
                    if n - k >= 0:
                        acc += h_ml[k] * x_m[n - k]
                subphase_samples.append(acc)
            # interleave subphase results into one stream per main phase
            parallel_outputs[m].extend(subphase_samples)

    # 5. Sum the parallel streams (sample-wise)
    min_len = min(len(p) for p in parallel_outputs)
    summed_output = np.zeros(min_len)
    for m in range(M):
        summed_output += np.array(parallel_outputs[m][:min_len])

    return np.asarray(summed_output) #, parallel_outputs
```

traditional polyphase resampler structure structure

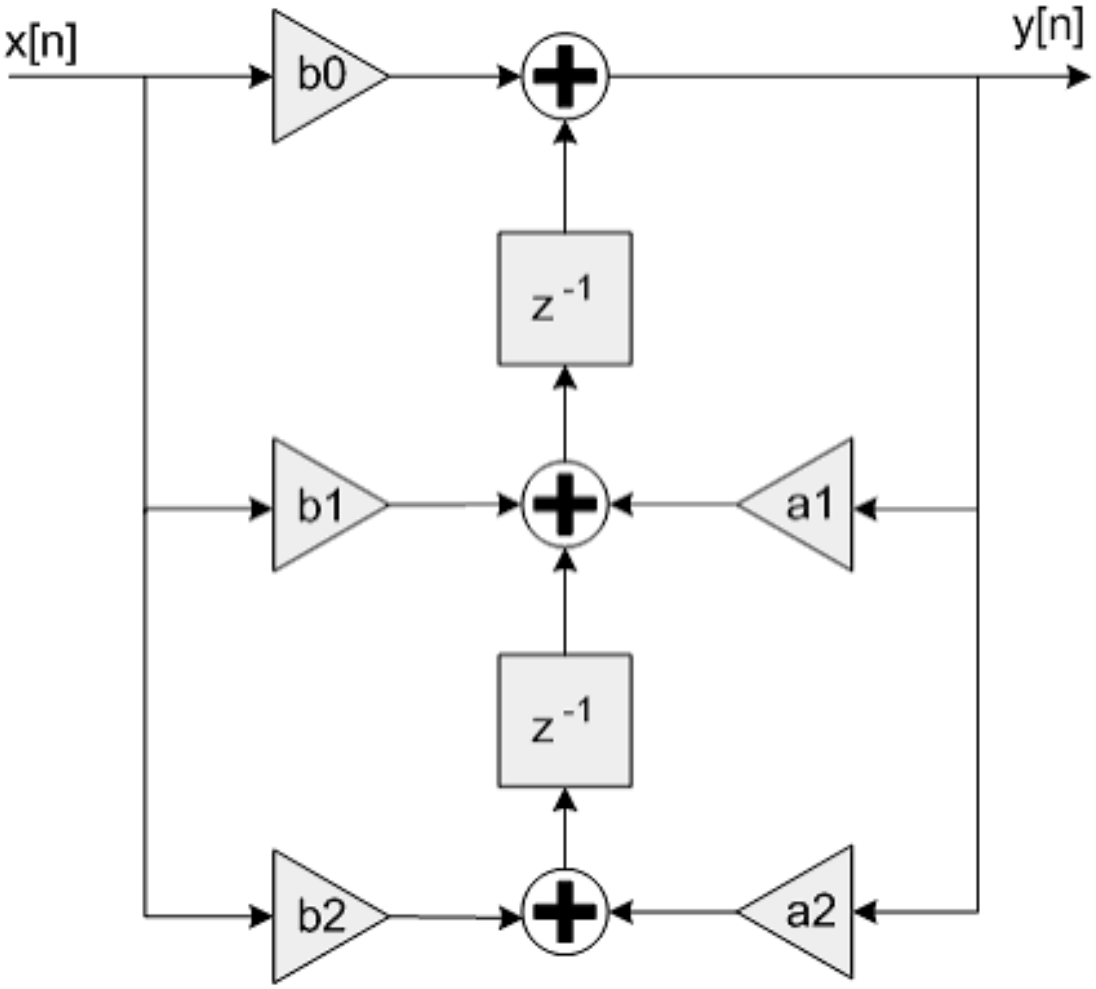


In [110…

```
def polyphase_traditional(x, L, M, h):

    phases = [h[l::L] for l in range(L)]
    y = []
    t = 0
    for n in range(len(x)):
        for l in range(L):
            acc = 0.0
            for k in range(len(phases[l])):
                if n - k >= 0:
                    acc += phases[l][k] * x[n - k]
            # output sample if timing hits a downsample point
            if t % M == 0:
```

```
                y.append(acc)
            t += 1
    return np.array(y)
```

biquad_IIR_DFII_T

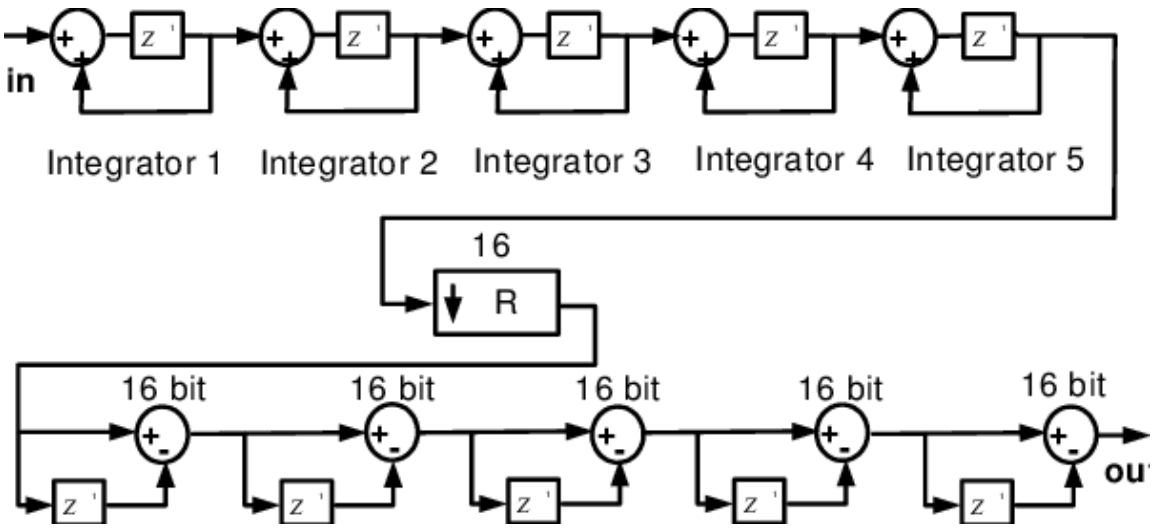


In [111…

```
def biquad_df2t(x, b, a):
    """
    Floating-point Direct Form II Transposed biquad filter
    x : input signal (float array)
    b : [b0, b1, b2]
    a : [1, a1, a2]  (a[0] must be 1)
    returns y : filtered output (float array)
    """
    N = len(x)
    y = np.zeros(N)
    s1, s2 = 0.0, 0.0  # delay elements

    b0, b1, b2 = b
    _, a1, a2 = a

    for n in range(N):
        # Direct Form II Transposed structure
        y_n = b0 * x[n] + s1
```

```
        s1 = b1 * x[n] - a1 * y_n + s2
        s2 = b2 * x[n] - a2 * y_n
        y[n] = y_n

    return y
```

CIC + comp FIR



CIC no compensation

using fixed point integers

more prone to overflow at high freq but accurate HW modelling. overflow happen for big frequency near nyqist

In [112…

```
def cic_decimator(x, R=2, M=1, N=5, B_in=16, B_out=16):
    """
    CIC decimator model with FPGA-style bit growth and truncation.

    Parameters:
        x : np.ndarray      - Input signal in range [-1, 1)
        R : int             - Decimation factor
        M : int             - Differential delay
        N : int             - Number of stages
        B_in : int          - Input bit width
        B_out : int         - Final output width

    Returns:
        y_out : np.ndarray  - Output after decimation and truncation
    """
    x = np.asarray(x, dtype=np.float64)

    # Convert to fixed-point integer
    x_int = np.round(x * (2**(B_in - 1))).astype(np.int64)

    n_in = len(x_int)
    integ = np.zeros((N, n_in), dtype=np.int64)
```

```
    # --- Integrator section ---
    integ[0, 0] = x_int[0]
    for i in range(1, n_in):
        integ[0, i] = integ[0, i-1] + x_int[i]
    for stage in range(1, N):
        integ[stage, 0] = integ[stage-1, 0]
        for i in range(1, n_in):
            integ[stage, i] = integ[stage, i-1] + integ[stage-1, i]
    y_int = integ[-1, :]

    # --- Decimate ---
    y_dec = y_int[::R]

    # --- Comb section ---
    n_out = len(y_dec)
    comb = np.zeros((N, n_out), dtype=np.int64)
    for stage in range(N):
        delay = np.zeros(M, dtype=np.int64)
        for i in range(n_out):
            cur_in = y_dec[i] if stage == 0 else comb[stage-1, i]
            cur_out = cur_in - delay[0]
            comb[stage, i] = cur_out
            delay = np.roll(delay, -1)
            delay[-1] = cur_in

    y_out = comb[-1, :]

    # --- Bit growth and truncation ---
    shift_bits = int(N * np.ceil(np.log2(R * M)))
    y_trunc = np.right_shift(y_out, shift_bits)
    y_float = y_trunc / (2**(B_out - 1))

    # Clip to [-1, 1)
    y_float = np.clip(y_float, -1, 1 - 1/(2**(B_out-1)))
    return y_float
```

using quantized float

no overflow at all

In [113…

```
def cic_decimator_2(x, R=2, M=1, N=5, B_in=16, B_out=16):
    """
    CIC decimator model (safe version, no overflow).
    Keeps same parameters and scaling behavior,
    but uses float64 arithmetic internally.

    Parameters:
        x : np.ndarray      - Input signal in range [-1, 1)
        R : int             - Decimation factor
        M : int             - Differential delay
        N : int             - Number of stages
        B_in : int          - Input bit width
        B_out : int         - Final output width
```

```
    Returns:
        y_out : np.ndarray  - Output after decimation and truncation
    """
    x = np.asarray(x, dtype=np.float64)

    # --- Simulate quantization (just for realism) ---
    x_int = np.round(x * (2**(B_in - 1))) / (2**(B_in - 1))

    n_in = len(x_int)
    integ = np.zeros((N, n_in), dtype=np.float64)

    # --- Integrator section ---
    integ[0, 0] = x_int[0]
    for i in range(1, n_in):
        integ[0, i] = integ[0, i-1] + x_int[i]
    for stage in range(1, N):
        integ[stage, 0] = integ[stage-1, 0]
        for i in range(1, n_in):
            integ[stage, i] = integ[stage, i-1] + integ[stage-1, i]

    y_int = integ[-1, :]

    # --- Decimation ---
    y_dec = y_int[::R]

    # --- Comb section ---
    n_out = len(y_dec)
    comb = np.zeros((N, n_out), dtype=np.float64)
    for stage in range(N):
        delay = np.zeros(M, dtype=np.float64)
        for i in range(n_out):
            cur_in = y_dec[i] if stage == 0 else comb[stage-1, i]
            cur_out = cur_in - delay[0]
            comb[stage, i] = cur_out
            delay = np.roll(delay, -1)
            delay[-1] = cur_in

    y_out = comb[-1, :]

    # --- Bit growth and truncation emulation ---
    shift_bits = int(N * np.ceil(np.log2(R * M)))
    y_scaled = y_out / (2**shift_bits)

    # Clip to output range
    y_float = np.clip(y_scaled, -1, 1 - 1/(2**(B_out - 1)))
    return y_float
```

CIC compensated

In [114…

```
def cic_comp(x, R=2):
    """
    CIC + Compensation FIR cascade.
    - If R == 1: bypass (return x unchanged)
    - Else: call cic_decimator_fpga() and convolve with compensation filter
    """
```

```
    if R == 1:
        return np.asarray(x, dtype=np.float64)

    # --- 1. Run CIC decimator ---
    # y_cic = cic_decimator(x, R=R)
    y_cic = cic_decimator_2(x, R=R)

    # --- 2. Load compensation filter coefficients ---
    coeff_path = f"comp_R{R}.txt"
    try:
        h_comp = np.loadtxt(coeff_path)/(2**15)
    except FileNotFoundError:
        raise FileNotFoundError(f"Compensation file '{coeff_path}' not found

    # --- 3. Manual convolution (low-level style) ---
    n = len(y_cic)
    L = len(h_comp)
    y_out = np.zeros(n + L - 1, dtype=np.float64)

    for i in range(n):
        for k in range(L):
            y_out[i + k] += y_cic[i] * h_comp[k]

    return y_out
```

DFE (integrated stages)

In [115…

```
def dfe_integrated(x, R=2):
    """
    Full DFE processing chain:
        1. Polyphase Resampler (L=2, M=3)
        2. Biquad Notch Filter
        3. CIC Decimator + Compensation FIR

    Loads coefficients from:
        - coeffs_fixed_q15.txt  (polyphase filter)
        - notch_b_q14.txt, notch_a_q14.txt  (biquad filter)
        - comp_R{R}.txt  (CIC compensation FIR)
    """

    # --- Stage 1: Polyphase Resampler ---
    L , M = 2,3
    h_15 = np.loadtxt("coeffs_fixed_q15.txt",dtype=int)/(2**15) *L
    P = int(np.ceil(len(h_15)/M/L))
    h = np.concatenate([h_15, np.zeros(M * P * L - len(h_15))])
    y = polyphase_resample(x, L=2, M=3, h=h)

    # --- Stage 2: Biquad Notch Filter ---
    b = np.loadtxt("notch_b_q14.txt")
    a = np.loadtxt("notch_a_q14.txt")
    frac_bits = 14
    b = b / (2**frac_bits)
    a = a / (2**frac_bits)
    y = biquad_df2t(y, b, a)
```

```
    # --- Stage 3: CIC + Compensation ---

    y_out = cic_comp(y, R=R)
    return y_out
```

Helping Routines

Quantize

In [116…

```
def fixed_point_quantize(x, n_int=1, n_frac=15):
    """
    Quantize a value (or numpy array) x to signed fixed-point s.I.F format.

    Parameters:
        x: float or np.ndarray – input value(s)
        n_int: int – number of integer bits (excluding sign bit)
        n_frac: int – number of fractional bits

    Returns:
        qx: quantized value in float (after rounding)
        q_int: quantized integer representation (int)
    """
    import numpy as np

    # Compute limits
    scale = 2 ** n_frac
    max_val = (2 ** n_int) - (1 / scale)
    min_val = - (2 ** n_int)

    # Clamp input
    x_clamped = np.clip(x, min_val, max_val)

    # Quantize
    q_int = np.round(x_clamped * scale).astype(int)

    # Handle wrap-around if overflowed
    q_int = np.clip(q_int, -int(2 ** (n_int + n_frac)), int(2 ** (n_int + n_

    # Convert back to float
    qx = q_int / scale

    # return qx, q_int
    return qx
```

Filter inspection

In [117…

```
def inspect_filter(h, fs):
    # DC gain: Sum of filter coefficients (valid for FIR filters)
    dc_gain = np.sum(h)
```

```
    print(f"Filter length: {len(h)}, DC gain (sum of taps) = {dc_gain:.6f}")

    # Impulse response
    n = np.arange(len(h))  # Time indices for impulse response

    # Frequency response
    w, H = freqz(h, worN=4096)
    f = w * fs / (2 * np.pi)  # Convert rad/sample to Hz

    # Create side-by-side plots
    plt.figure(figsize=(12, 4))

    # Subplot 1: Impulse response
    plt.subplot(1, 2, 1)
    plt.stem(n, h, basefmt=" ")
    plt.xlabel('Sample index (n)')
    plt.ylabel('Amplitude')
    plt.title('Impulse Response')
    plt.grid(True)

    # Subplot 2: Frequency response
    plt.subplot(1, 2, 2)
    plt.plot(f / 1e6, 20 * np.log10(np.abs(H) + 1e-12))
    plt.axvline(3.0, color='r', linestyle='--', label='3 MHz (output Nyquist
    plt.xlabel('Frequency (MHz)')
    plt.ylabel('Magnitude (dB)')
    plt.title('Frequency Response')
    plt.grid(True)
    plt.legend()
    plt.xlim(0, fs / 2 / 1e6)

    plt.tight_layout()  # Adjust spacing between subplots
    plt.show()
```

plotting I/O

In [118…

```
def plot_time_freq(x, y, fs_in, fs_out, title,samples=200,L=1,M=1,step = 200
    fig, axes = plt.subplots(2,1, figsize=(12,6))

    # Trim to same duration for time plot
    min_samples = min(len(x), int(len(y)*fs_in/fs_out))
    x = x[:min_samples]
    y = y[:int(min_samples*fs_out/fs_in)]

    n_in = np.arange(len(x))/fs_in*1e6
    n_out = np.arange(len(y))/fs_out*1e6

    # ---- Time-domain ----
    axes[0].plot(n_in[samples-step:samples], x[samples-step:samples], label=
    axes[0].plot(n_out[int((samples-step)*(L/M)):int(samples*(L/M))], y[int(
    axes[0].set_title(f"{title} - Time domain ( {step} samples)")
    axes[0].set_xlabel("Time (µs)")
    axes[0].legend()

    # ---- Frequency spectrum ----
```

```
    Nfft = 16384*64  # power of 2 for clean FFT
    X = np.fft.fftshift(np.fft.fft(x, Nfft))
    Y = np.fft.fftshift(np.fft.fft(y, Nfft))
    f_in = np.fft.fftshift(np.fft.fftfreq(Nfft, 1/fs_in))/1e6
    f_out = np.fft.fftshift(np.fft.fftfreq(Nfft, 1/fs_out))/1e6

    axes[1].plot(f_in, 20*np.log10(np.abs(X)+1e-12),label="input",color="blu
    axes[1].plot(f_out, 20*np.log10(np.abs(Y)+1e-12),label="output", color="
    axes[1].set_xlim([-max(fs_out/(2*1e6),fs_in/(2*1e6)), max(fs_out/(2*1e6)
    axes[1].set_title("Output Spectrum")
    axes[1].set_xlabel("Frequency (MHz)")
    axes[1].set_ylabel("Magnitude (dB)")
    axes[1].legend()

    plt.tight_layout()
    plt.show()
```

Testing & Graphs

Polyphase Filter

Promplems and Solutions

1. For prpoer resampling the fir filter need to be $w_c = \frac{1}{\max(L, M)} \cdot \frac{F_s}{2}$
2. so we Design a Prototype FIR at 1.5 wher 1.4 lesss than .25dB and after 1.6 less than -80dB
3. implementing this prototype FIR in polyphase structure give effective filtering at 3 MHz

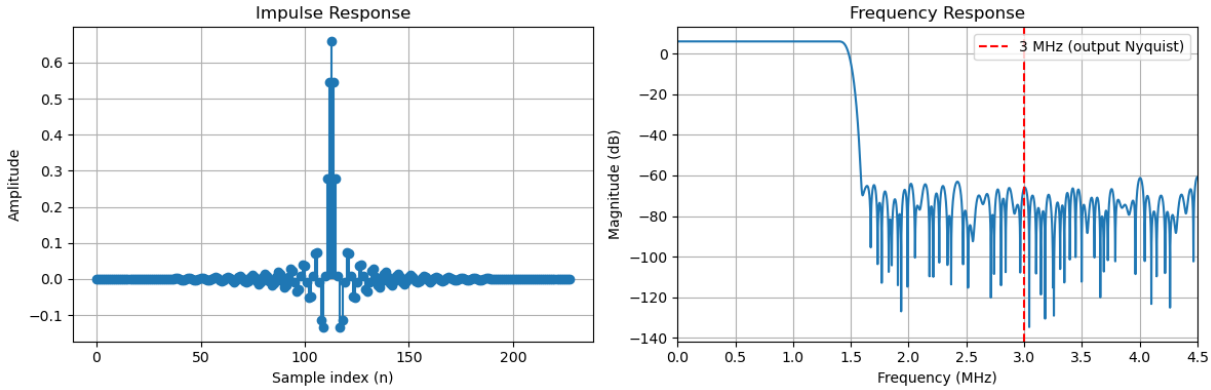
Paramaters definition & Filter Testing

In [119…

```
# Example usage
fs_in = 9e6
L, M = 2, 3
fs_out = fs_in * L/M
N = 8192   # number of samples
t = np.arange(N) / fs_in

#  # 1. Design prototype filter at upsampled rate

h_win = firwin(403, 1/ max(L,M) ) * L
h_15 = np.loadtxt("coeffs_fixed_q15.txt",dtype=int)/(2**15) *L
P = int(np.ceil(len(h_15)/M/L))
h = np.concatenate([h_15, np.zeros(M * P * L - len(h_15))])
```

```
inspect_filter(h,9e6)


# Example input: 1 MHz tone
t = np.arange(0, 9000) / fs_in
x = np.sin(2 * np.pi * 1e6 * t)

y = polyphase_resample(x, L, M,h)
print("Input length:", len(x))

print("Final output length:", len(y))
```

```
Filter length: 228, DC gain (sum of taps) = 1.998474
```

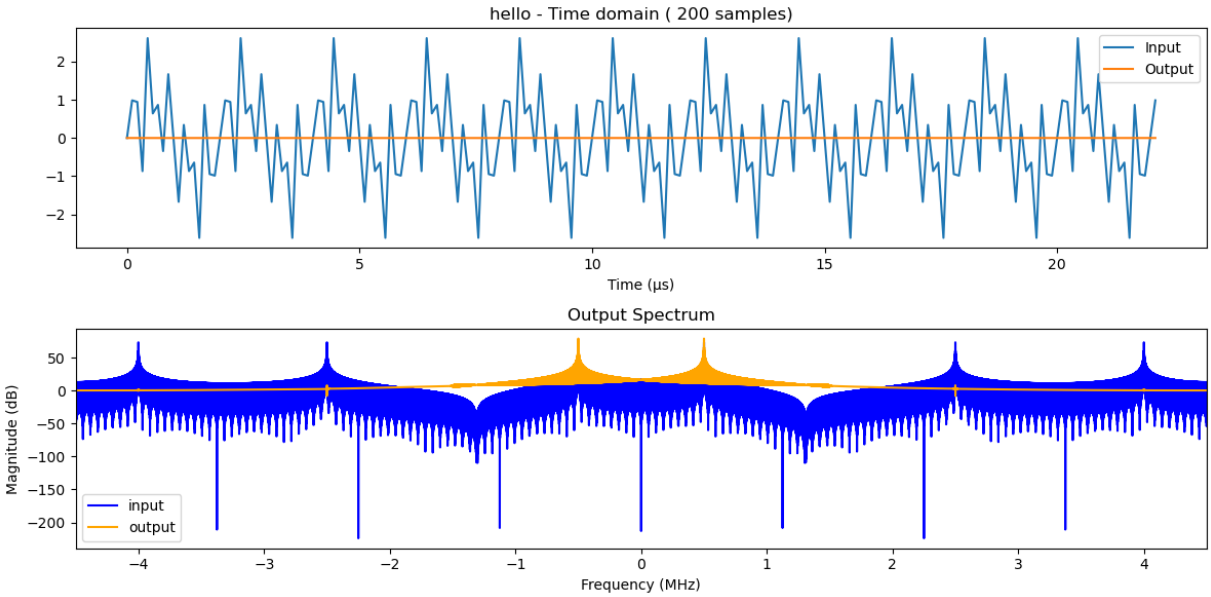



```
Input length: 9000
Final output length: 6000
```

prototype filter when not in polyphase strucure only filter at 1.5 MHz

In [120…

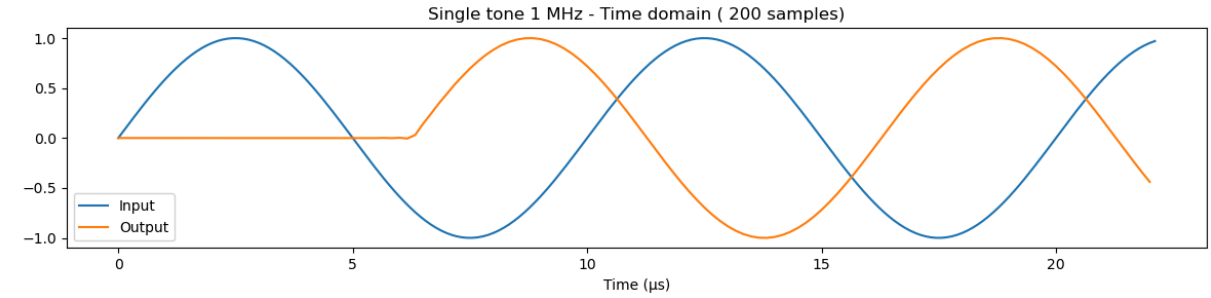
```
y = []
x = np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.sin(2*np.pi*5e6*t
h_ml = firwin(603, 1/ max(2,3) ) * 2
for n in range(len(x)):
        acc = 0.0
        for k in range(len(h_ml)):
            if n - k >= 0:
                acc += h_ml[k] * x[n - k]
        y.append(acc)
plot_time_freq(x, y, fs_in, fs_in,"hello")
```

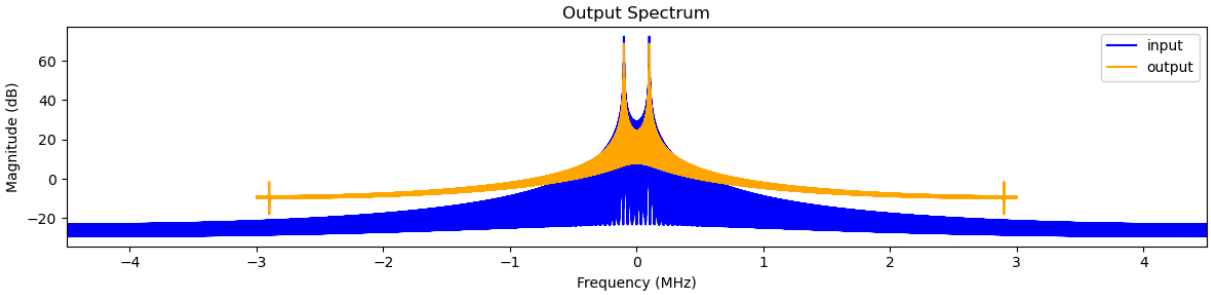


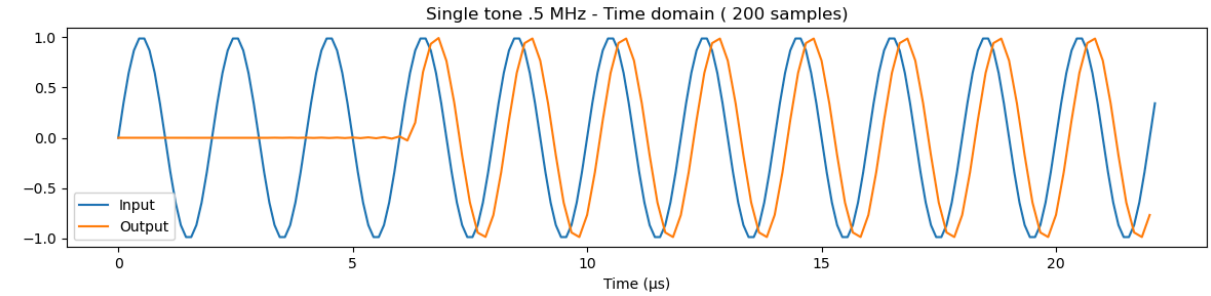
I/O Testing

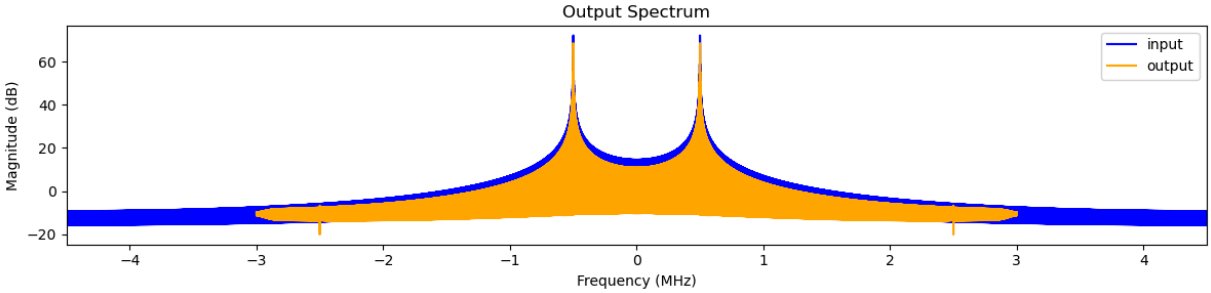
In [121…

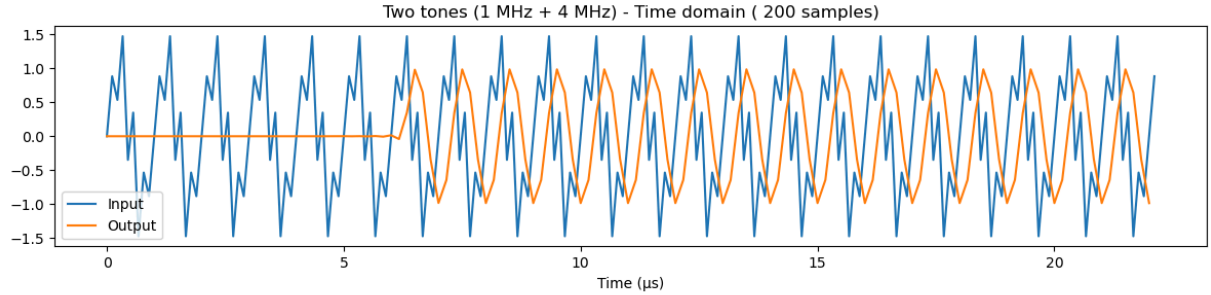
```
# ----------------- TESTING -----------------
fs_in = 9e6
L, M = 2, 3
fs_out = fs_in * L/M
N = 8192   # number of samples
t = np.arange(N) / fs_in

# Different input signals
signals = {
    "Single tone 1 MHz": np.sin(2*np.pi*1e5*t),
    "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
    "Two tones (1 MHz + 4 MHz)": np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.pi*
    "Wideband noise": np.random.randn(N),
    "Multi-tone": np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.sir
}

# Run tests
for name, sig in signals.items():
    y = polyphase_resample(sig, L, M,h)
    # y_2 = polyphase_traditional(sig, L, M,h)
    plot_time_freq(sig, y, fs_in, fs_out, name,200,L,M)
    # plot_time_freq(sig, y_2, fs_in, fs_out, name+" matlab",200,L,M)
```

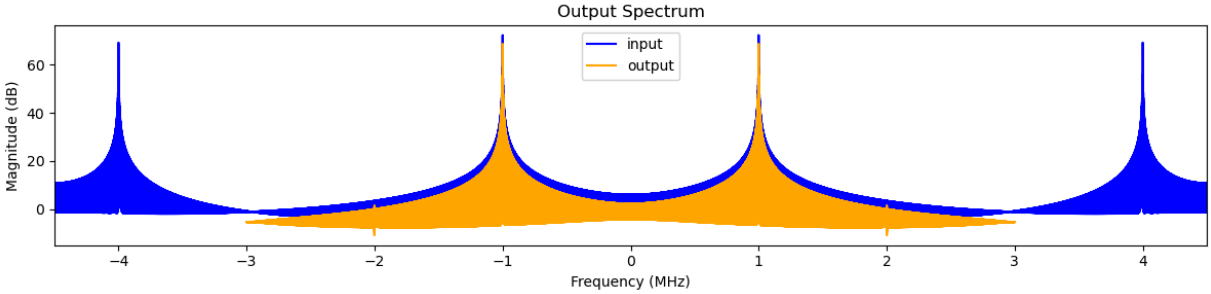

Single tone 1 MHz - Time domain (200 samples)
Input
Output
Time (µs)

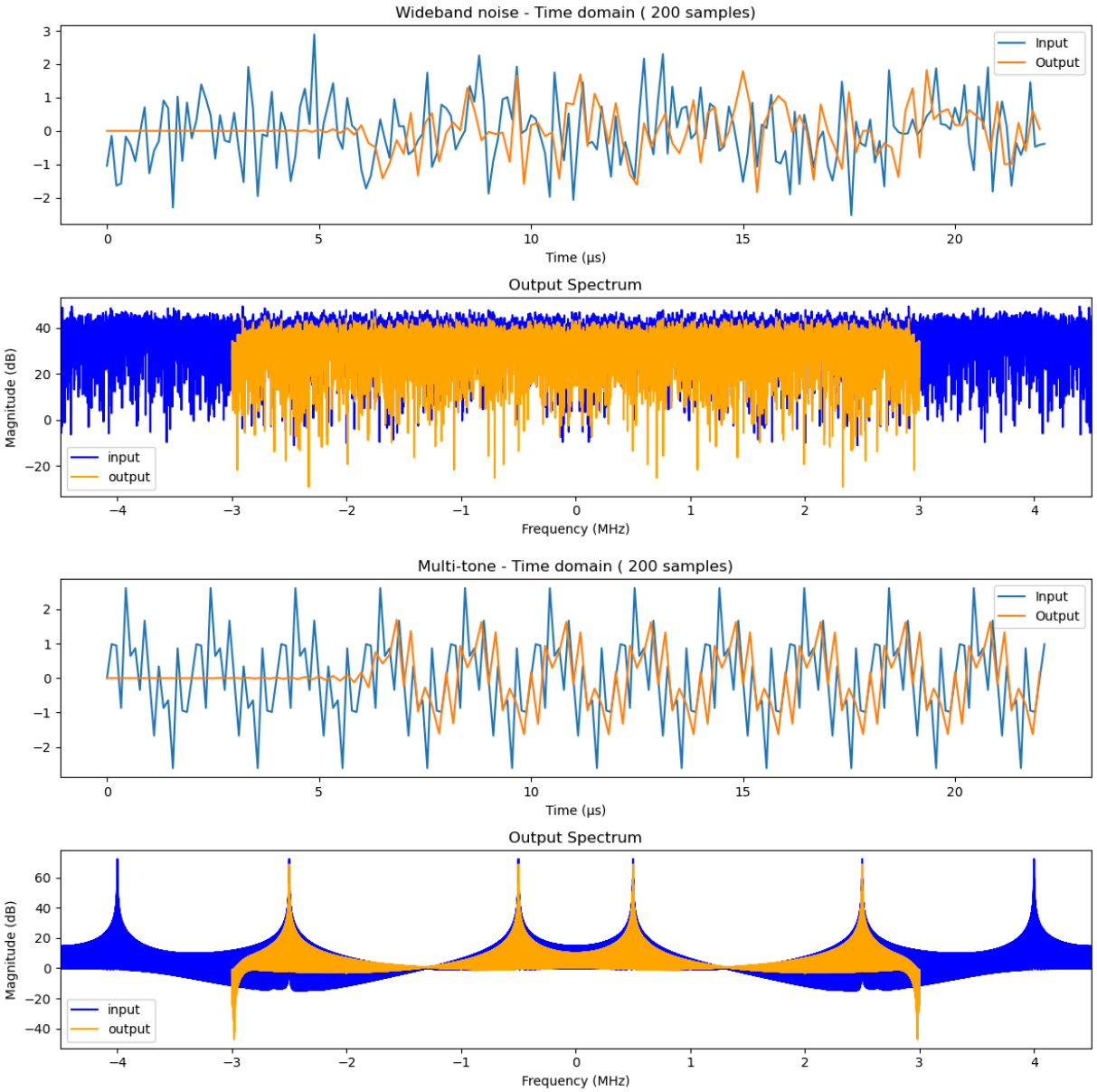

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)


Single tone .5 MHz - Time domain (200 samples)
Input
Output
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)


Two tones (1 MHz + 4 MHz) - Time domain (200 samples)
Input
Output
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)



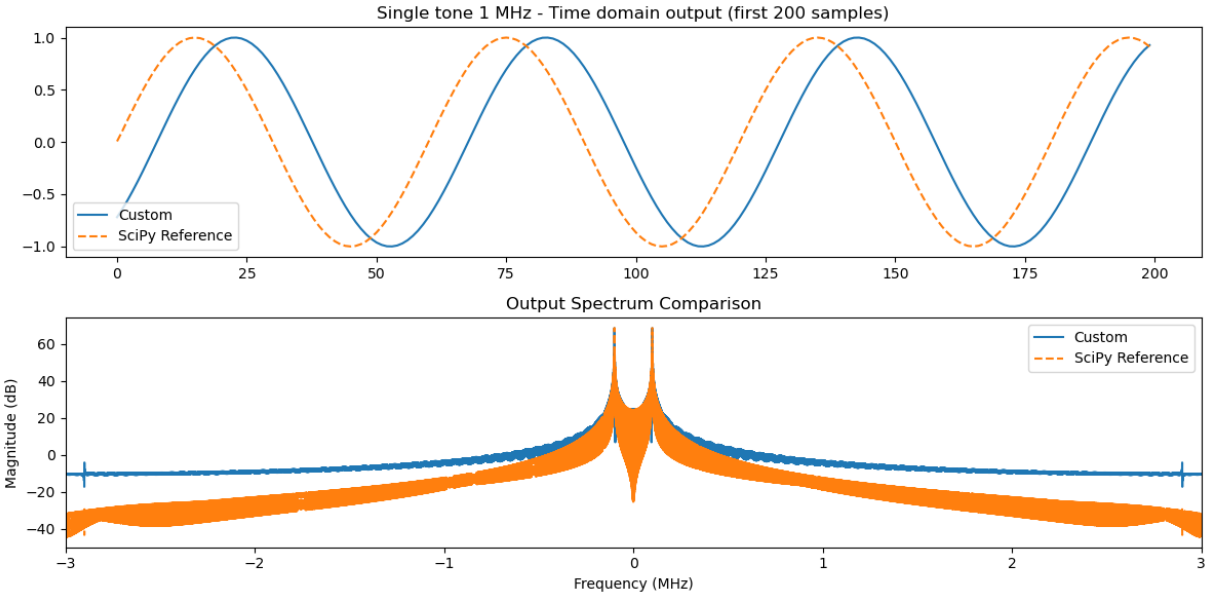
reference comparison

In [122…

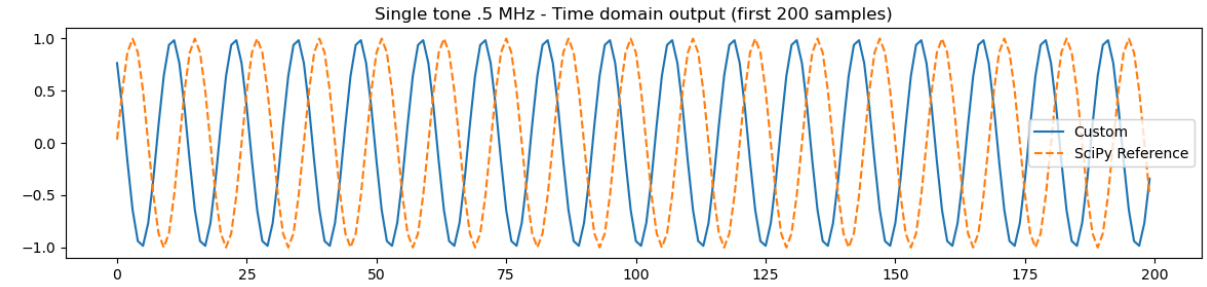
```
# ----------------- COMPARISON BLOCK -----------------
for name, sig in signals.items():
    # Your implementation
    y_custom = polyphase_resample(sig, L, M,h)

    # Reference implementation (SciPy)
    y_ref = resample_poly(sig, L, M, window=('kaiser', 8.6))  # default Kais

    # --- Match lengths for fair comparison ---
    min_len = min(len(y_custom), len(y_ref))
    y_custom = y_custom[:min_len]
    y_ref = y_ref[:min_len]

    # --- Time-domain difference ---
    mse = np.mean((y_custom - y_ref)**2)
    print(f"{name}: MSE vs. SciPy = {mse:.2e}")
```

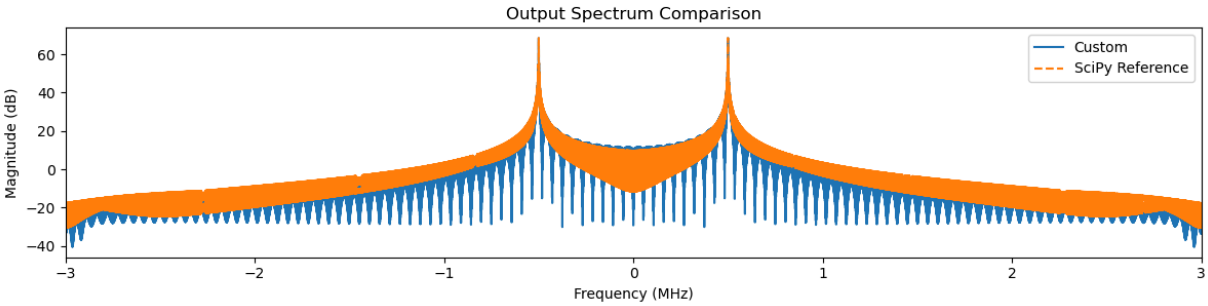
```
    # --- Plot comparison ---
    plt.figure(figsize=(12,6))

    # First 200 samples (time domain)
    plt.subplot(2,1,1)
    plt.plot(y_custom[150:350], label="Custom")
    plt.plot(y_ref[:200], '--', label="SciPy Reference")
    plt.title(f"{name} - Time domain output (first 200 samples)")
    plt.legend()

    # Spectrum comparison
    Nfft = 16384
    Yc = np.fft.fftshift(np.fft.fft(y_custom, Nfft))
    Yr = np.fft.fftshift(np.fft.fft(y_ref, Nfft))
    f = np.fft.fftshift(np.fft.fftfreq(Nfft, 1/fs_out))/1e6

    plt.subplot(2,1,2)
    plt.plot(f, 20*np.log10(np.abs(Yc)+1e-12), label="Custom")
    plt.plot(f, 20*np.log10(np.abs(Yr)+1e-12), '--', label="SciPy Reference"
    plt.xlim([-fs_out/(2*1e6), fs_out/(2*1e6)])
    plt.title("Output Spectrum Comparison")
    plt.xlabel("Frequency (MHz)")
    plt.ylabel("Magnitude (dB)")
    plt.legend()

    plt.tight_layout()
    plt.show()
```

```
Single tone 1 MHz: MSE vs. SciPy = 1.69e+00
```

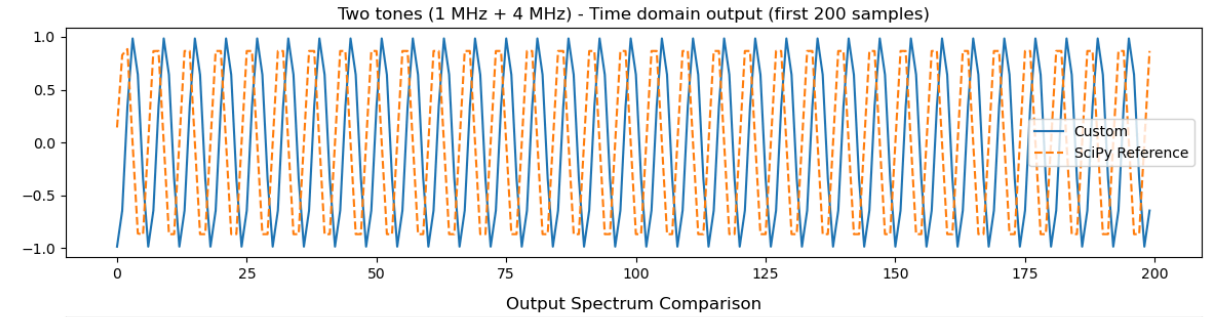


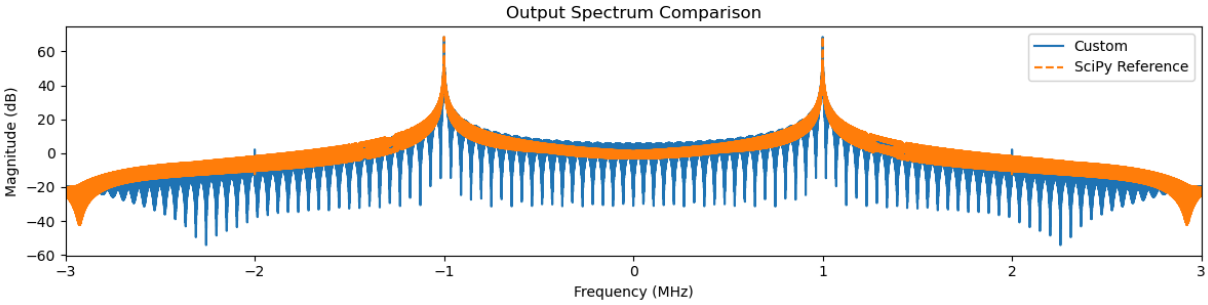

```
Single tone .5 MHz: MSE vs. SciPy = 3.58e-01
```



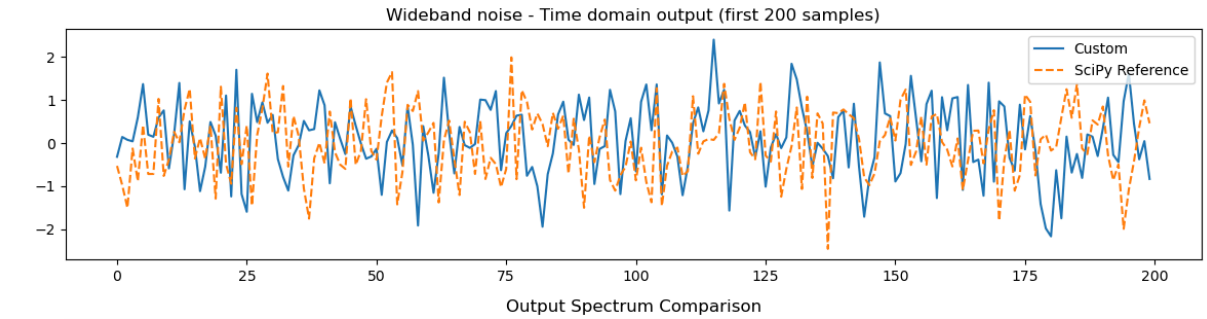


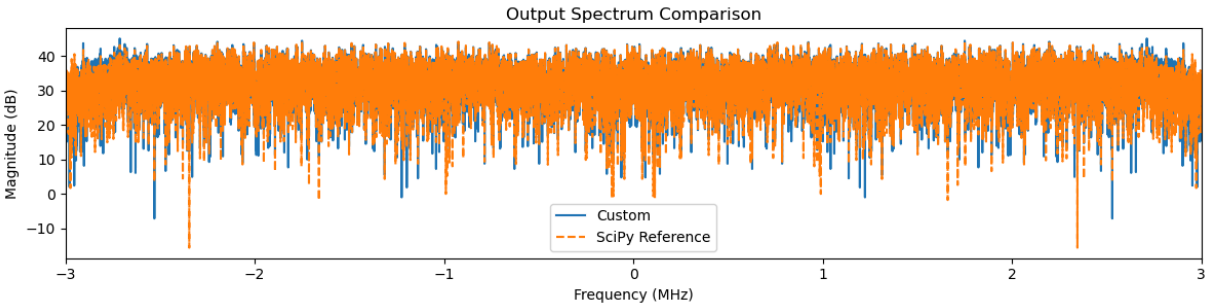
Two tones (1 MHz + 4 MHz): MSE vs. SciPy = 1.17e+00



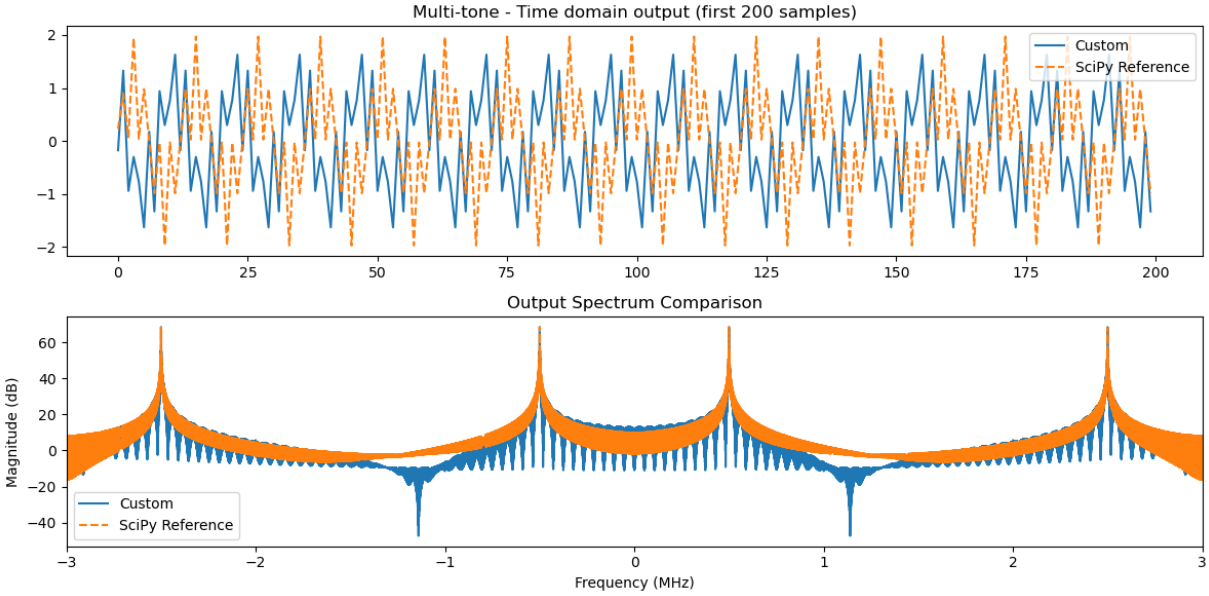


Wideband noise: MSE vs. SciPy = 1.25e+00





```
Multi-tone: MSE vs. SciPy = 1.65e+00
```



IIR Notch

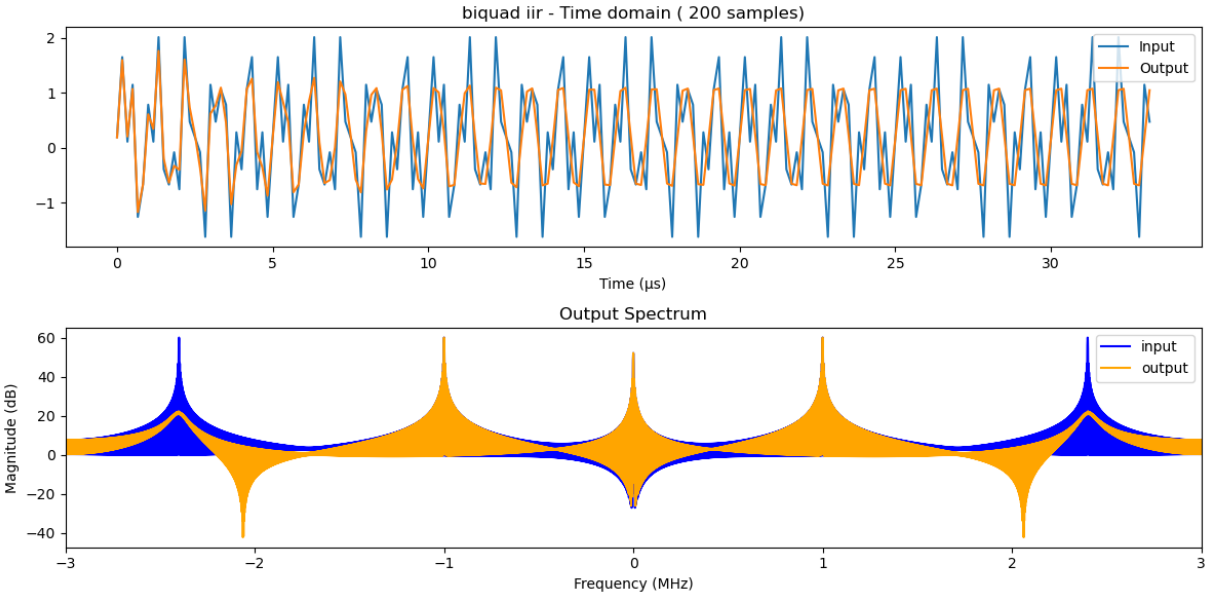
I/O + Filter Testing

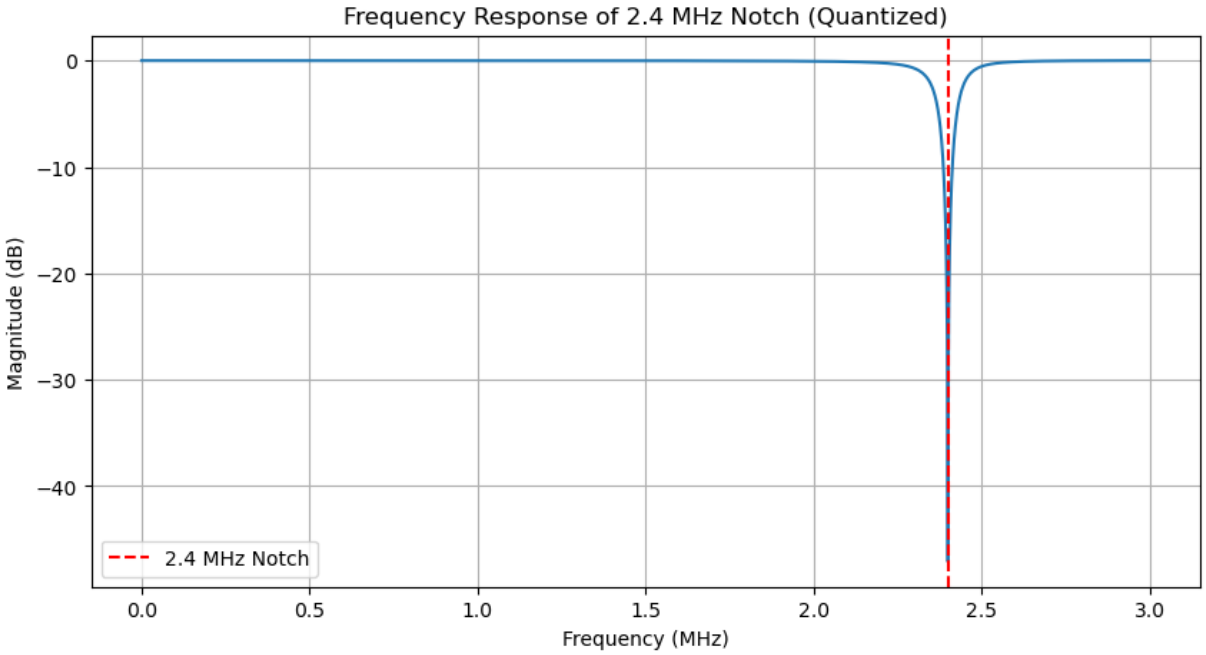
In [123…

```
b_q14 = np.loadtxt("notch_b_q14.txt", dtype=int)
a_q14 = np.loadtxt("notch_a_q14.txt", dtype=int)

frac_bits = 14
b_fixed = b_q14 / (2**frac_bits)
a_fixed = a_q14 / (2**frac_bits)

print("Loaded Q1.14 coefficients:")
print("b_q14 =", b_q14)
print("a_q14 =", a_q14)
print("b_fixed =", b_fixed)
print("a_fixed =", a_fixed)

# ================================================================
# 2. Generate test signal (1 MHz + small DC)
# ================================================================

fs = 6e6
t_2 = np.arange(0, 2048) / fs
# t = t* fs_in / fs
x = np.sin(2*np.pi*1e6*t_2) + 0.2*np.ones_like(t_2) + np.sin(2*np.pi*2.4*1e6


# ================================================================
# 4. Run both filters
# ================================================================

y = biquad_df2t(x, b_fixed, a_fixed)
```

```
plot_time_freq(x, y, fs, fs, "biquad iir",samples=200,L=1,M=1)
# ================================================================
# 6. Frequency response
# ================================================================

w, h = freqz(b_fixed, a_fixed, worN=4096, fs=fs)
plt.figure(figsize=(10,5))
plt.plot(w/1e6, 20*np.log10(np.abs(h)))
plt.axvline(2.4, color='r', linestyle='--', label='2.4 MHz Notch')
plt.title('Frequency Response of 2.4 MHz Notch (Quantized)')
plt.xlabel('Frequency (MHz)')
plt.ylabel('Magnitude (dB)')
plt.grid(True)
plt.legend()
plt.show()
```

```
Loaded Q1.14 coefficients:
b_q14 = [15725 25443 15725]
a_q14 = [16384 25443 15066]
b_fixed = [0.95977783 1.55291748 0.95977783]
a_fixed = [1.         1.55291748 0.91955566]
```



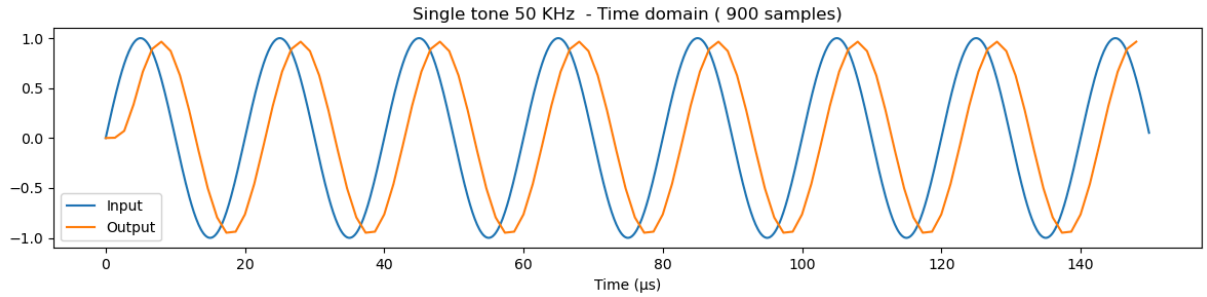


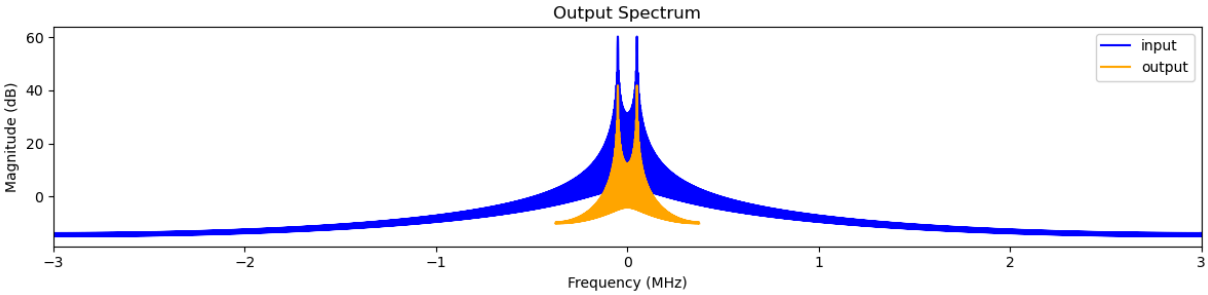
CIC + Compensation FIR

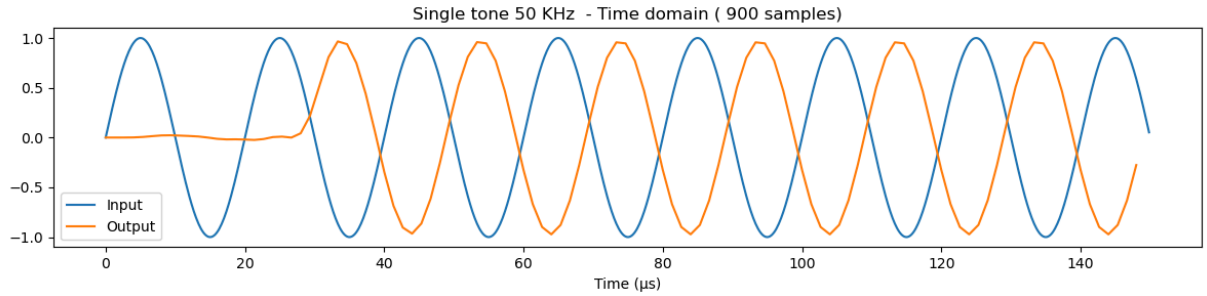
I/O Testing

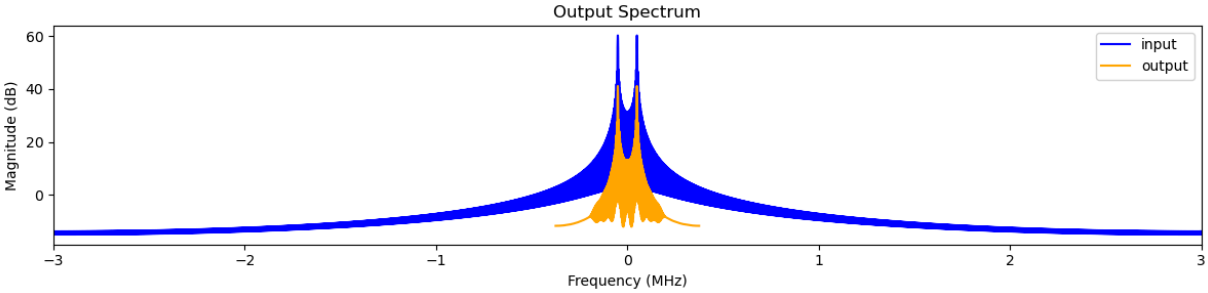
In [124…

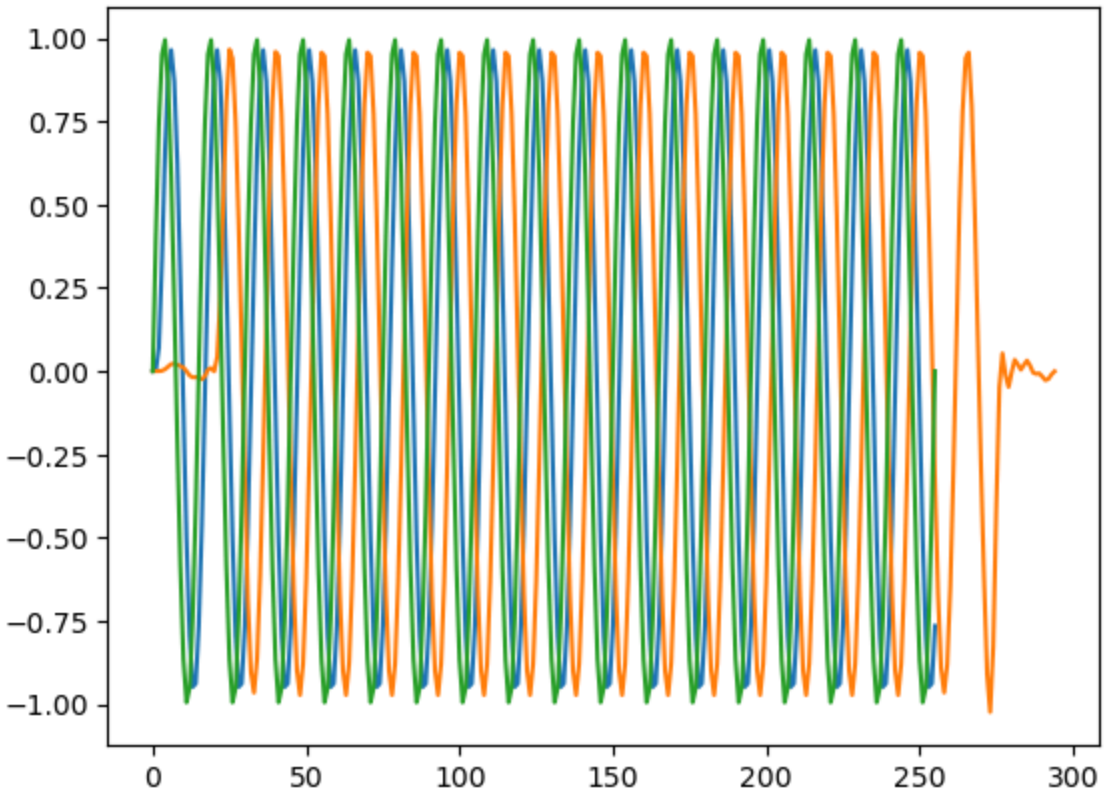
```
# ----------------- TESTING -----------------
fs_cic = 6e6
R = 8
# Different input signals
signals = {
    "Single tone 50 KHz ": np.sin(2*np.pi*5e4*t_2),
#     "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
#     "Two tones (1 MHz + 4 MHz)": np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.p
    #  "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e4*t_2) + np.sin(2*np.pi*2.5e4*t_2) + r
}

# Run tests
for name, sig in signals.items():
    y_1= cic_decimator(sig,R)
    y_2= cic_comp(sig,8)
    plot_time_freq(sig, y_1, fs_cic, fs_cic /R, name,900 ,1  , R  , 900)
    plot_time_freq(sig, y_2, fs_cic, fs_cic /R, name,900 , 1 , R  , 900)
    plt.plot(y_1,label="no comp")
    plt.plot(y_2,label=" comp")
    plt.plot(sig[::R],label="original")
input_cic_1 = (np.sin(2*np.pi*0.5e4*t) + np.sin(2*np.pi*2.5e4*t) + np.sin(2*
bypassed = cic_comp(input_cic_1,1)
plot_time_freq(input_cic_1, bypassed,fs_cic,fs_cic,"bypassed", 900,1,1,900)
plt.plot(bypassed)
plt.plot(input_cic_1* 1.08)
```

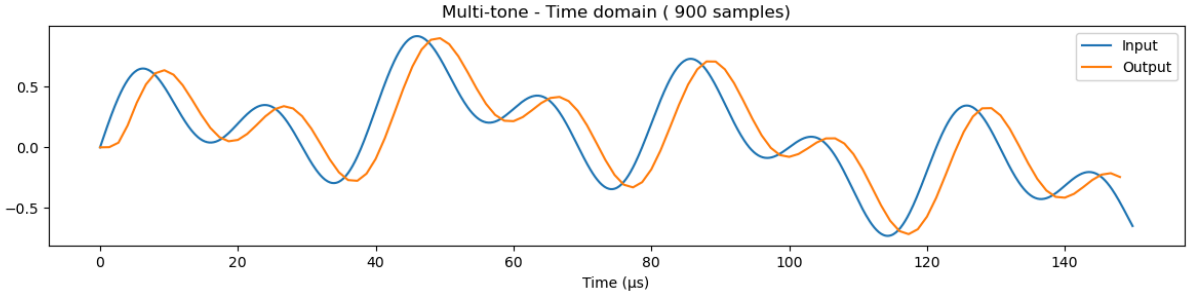

Single tone 50 KHz - Time domain (900 samples)
Input
Output
Time (µs)

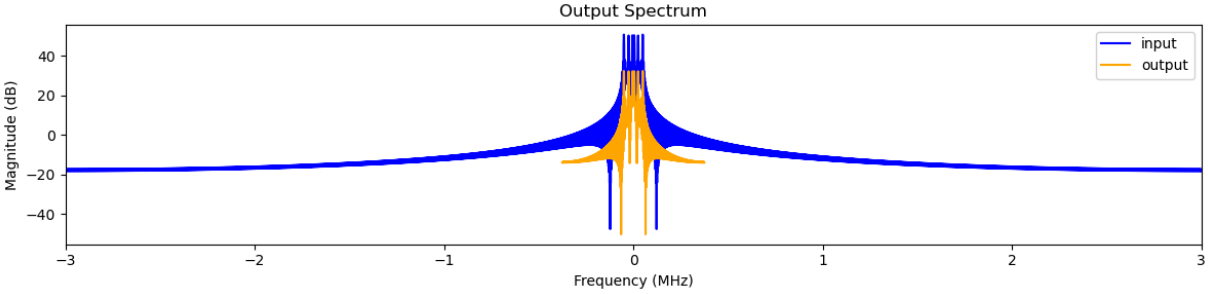

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

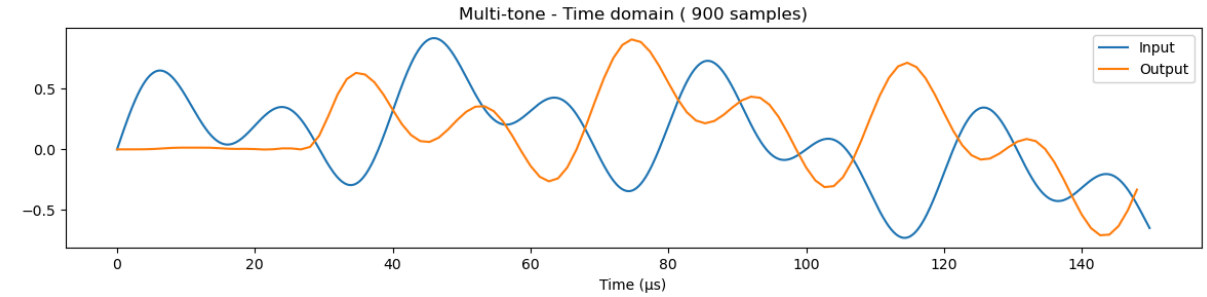

Single tone 50 KHz - Time domain (900 samples)
Input
Output
Time (µs)

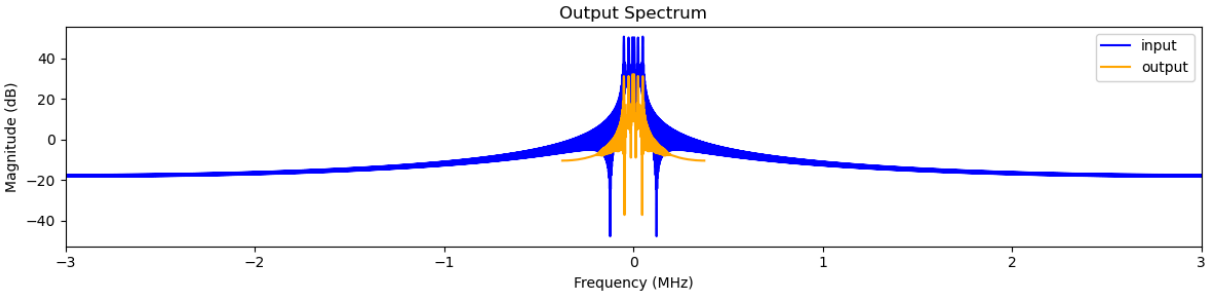

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

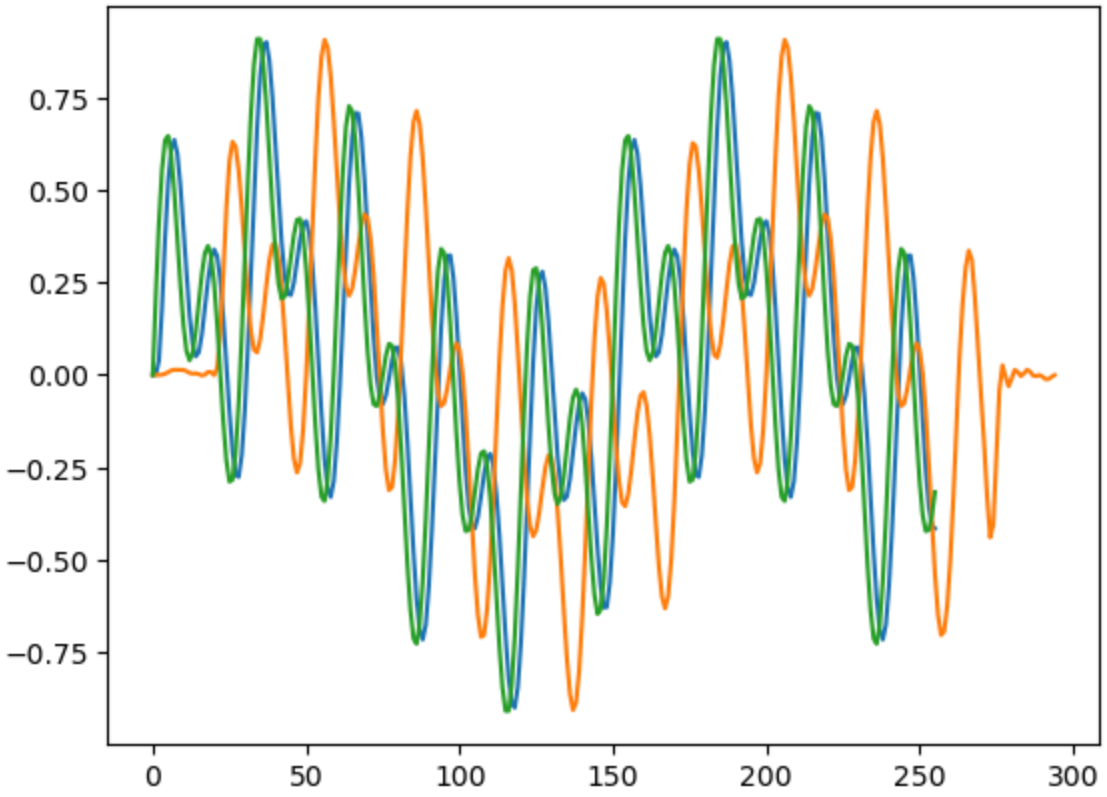

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
50
100
150
200
250
300

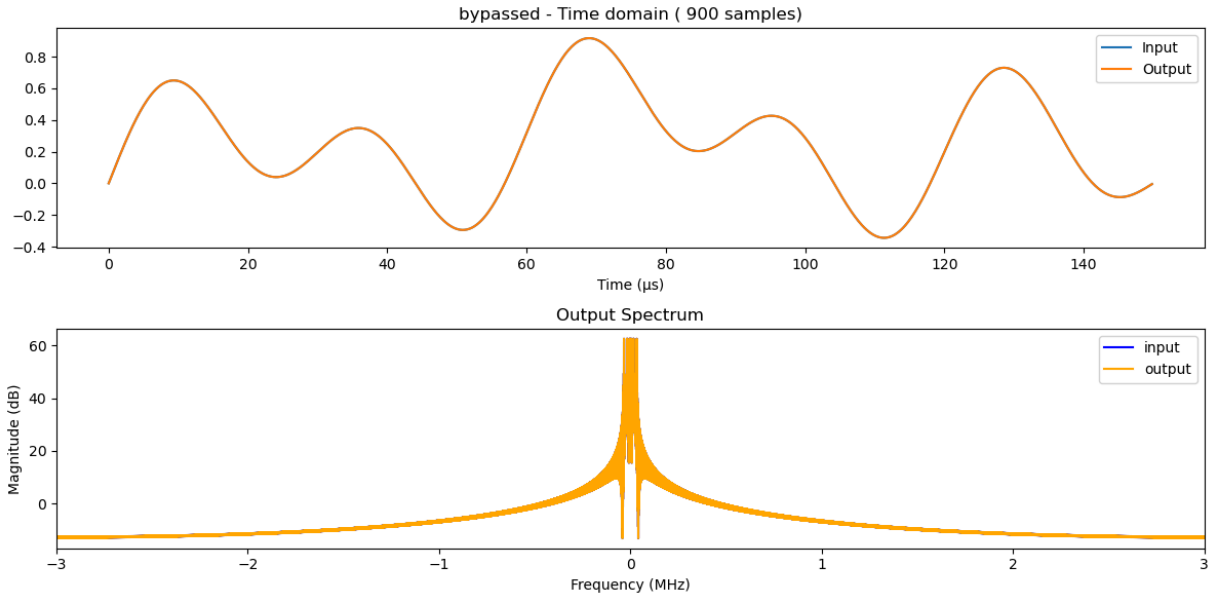

Multi-tone - Time domain (900 samples)
Input
Output
0.5
0.0
−0.5
0
20
40
60
80
100
120
140
Time (µs)


Output Spectrum
input
output
40
20
0
−20
−40
Magnitude (dB)
−3
−2
−1
0
1
2
3
Frequency (MHz)

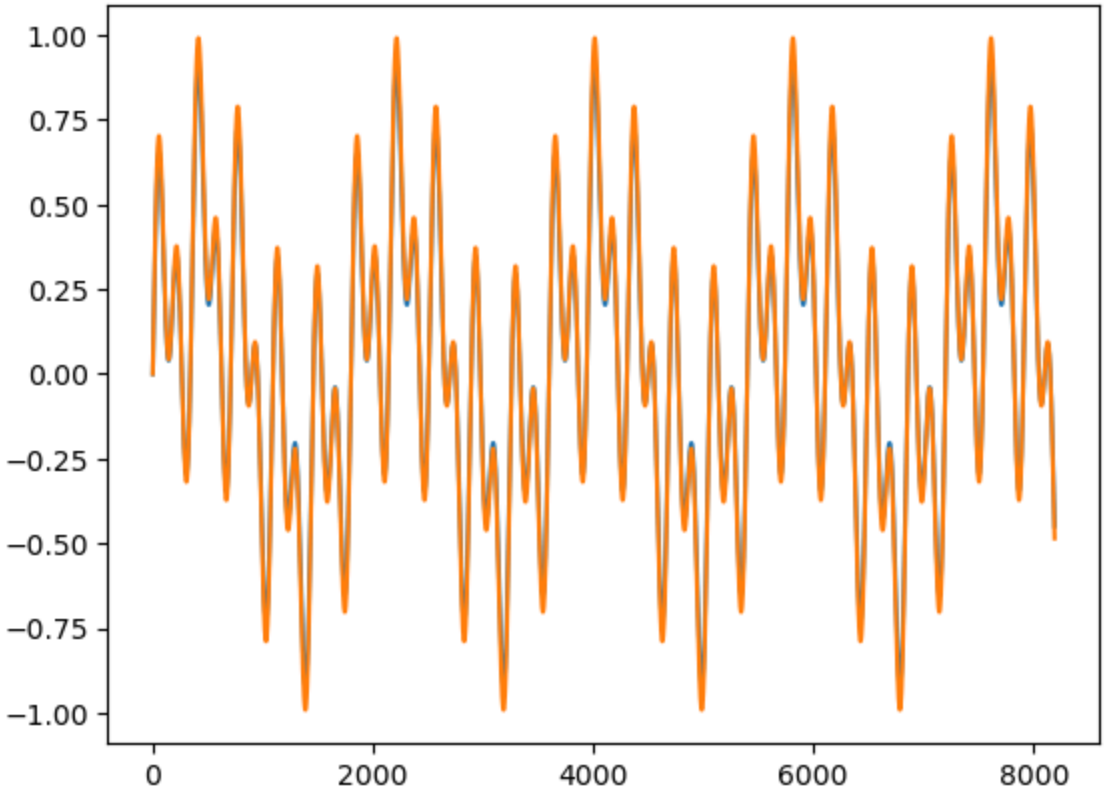

Multi-tone - Time domain (900 samples)
Input
Output
0.5
0.0
−0.5
0
20
40
60
80
100
120
140
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
40
20
0
−20
−40
−3
−2
−1
0
1
2
3
Frequency (MHz)


0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
50
100
150
200
250
300



Out[124… [<matplotlib.lines.Line2D at 0x1641e4f10>]



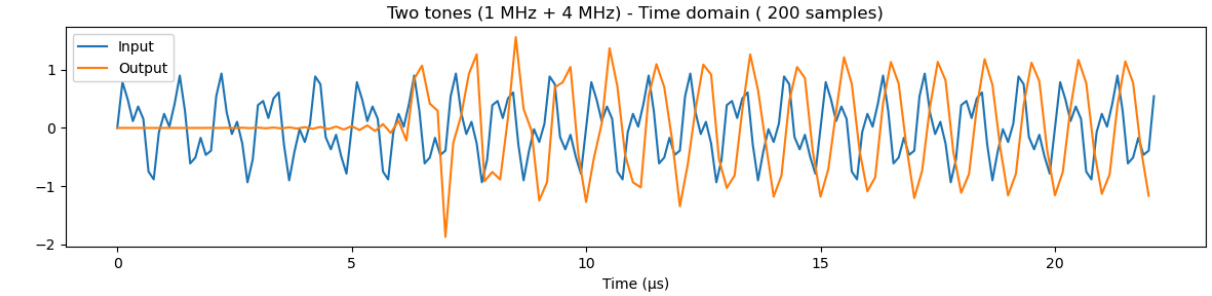
DFE (integrated)

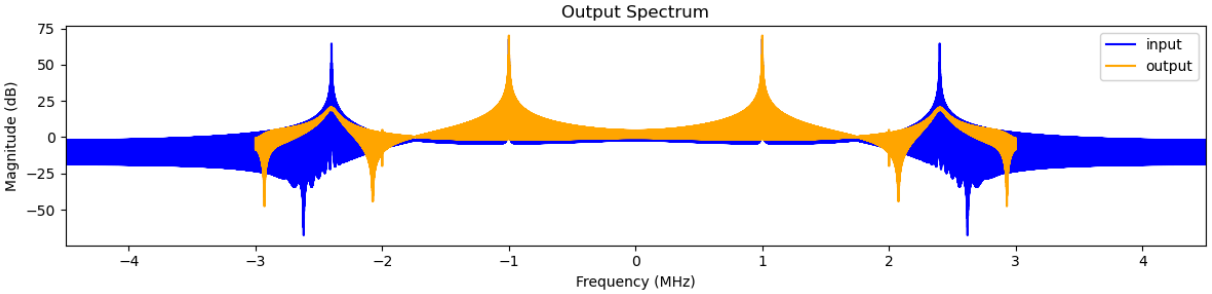
I/O Testing

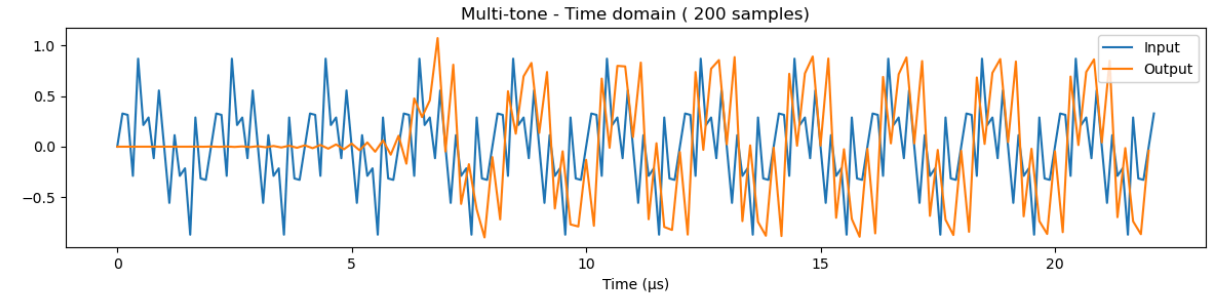
note: that when we use resampler be it interpolator or decimator the frequency DFT of the output is scaled be L/M of that resampling ratio so when we decimate by 4 it will be 1/4 of the input spectrum

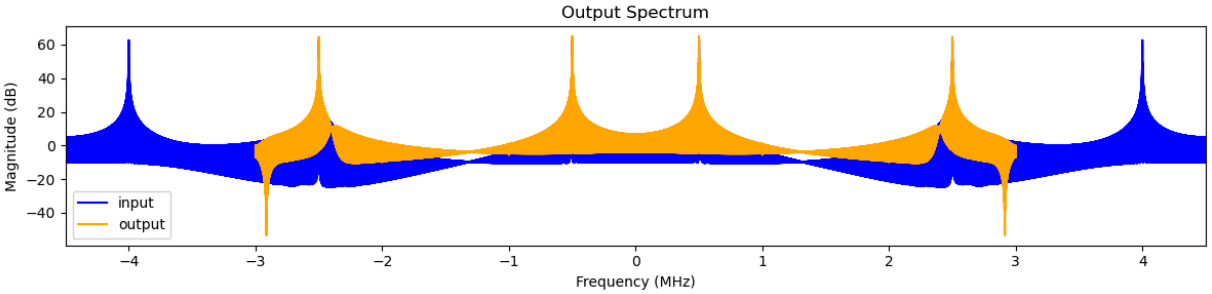
In [125…

```
fs_in = 9e6
fs_out = 6e6
R = 1
# Different input signals
signals = {
    # "Single tone 50 KHz ": np.sin(2*np.pi*5e4*t),
#     "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
    "Two tones (1 MHz + 4 MHz)": (np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.pi
    #  "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.si
}

# Run tests
for name, sig in signals.items():
    y = dfe_integrated(sig ,R)
    plot_time_freq(sig, y*2, fs_in, fs_out /R, name,200 , 2 , M*R  , 200)
```





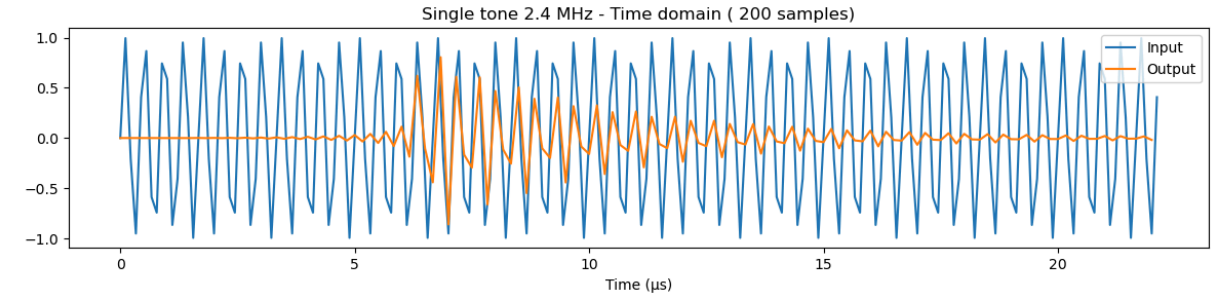


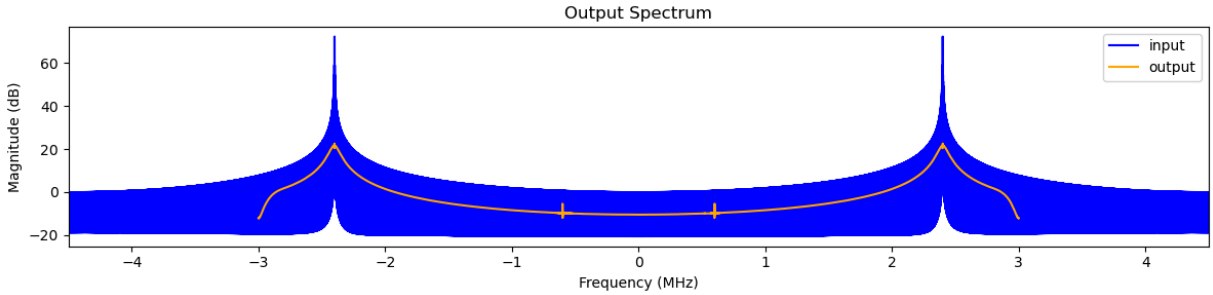


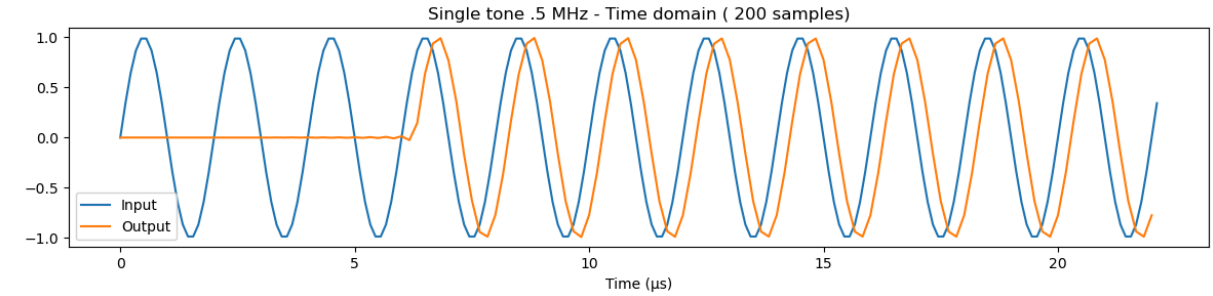
In [126…

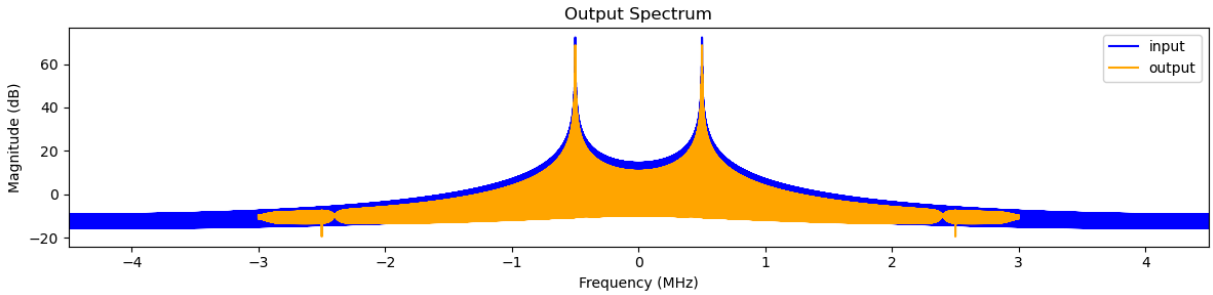
```
# ----------------- TESTING -----------------
```

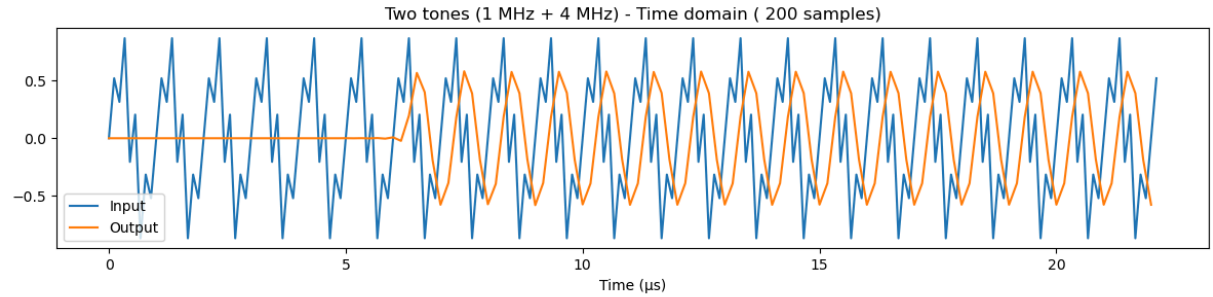
```
# Different input signals
signals = {
    "Single tone 2.4 MHz": np.sin(2*np.pi*2.4e6*t),
    "Single tone .5 MHz": np.sin(2*np.pi*5e5*t),
    "Two tones (1 MHz + 4 MHz)": (np.sin(2*np.pi*1e6*t) + 0.7*np.sin(2*np.pi
    # "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e6*t) + np.sin(2*np.pi*2.5e6*t) + np.si
}

# Run tests
for name, sig in signals.items():
    y = dfe_integrated(sig, 1)
    plot_time_freq(sig, y, fs_in, fs_out, name,200,L,M)



signals_2 = {
    "Single tone 2.4 KHz": np.sin(2*np.pi*2.4e3*t),
    # "Single tone .5 MHz": np.sin(2*np.pi*2e4*t),
    "Two tones (10 khz + 40 khz)": (np.sin(2*np.pi*1e4*t) + 0.7*np.sin(2*np.
    # "Wideband noise": np.random.randn(N),
    "Multi-tone": (np.sin(2*np.pi*0.5e4*t) + np.sin(2*np.pi*2.5e4*t) + np.si
}

print("stop_2 low freq sig at R = 2")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 2)
    plot_time_freq(sig, y, fs_in, fs_out/2, name+"at R = 2",20000,L,M*2,2000

print("stop_3 low freq sig at R = 4")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 4)
    plot_time_freq(sig, y, fs_in, fs_out/4, name+"at R = 4",20000,L,M*4,2000

print("stop_4 low freq sig at R = 8")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 8)
    plot_time_freq(sig, y, fs_in, fs_out/8, name+"at R = 8",20000,L,M*8,2000

print("stop_5 low freq sig at R = 16")

for name, sig in signals_2.items():
    y = dfe_integrated(sig, 16)
    plot_time_freq(sig, y, fs_in, fs_out/16, name+"at R = 16",20000,L,M*16,2
```

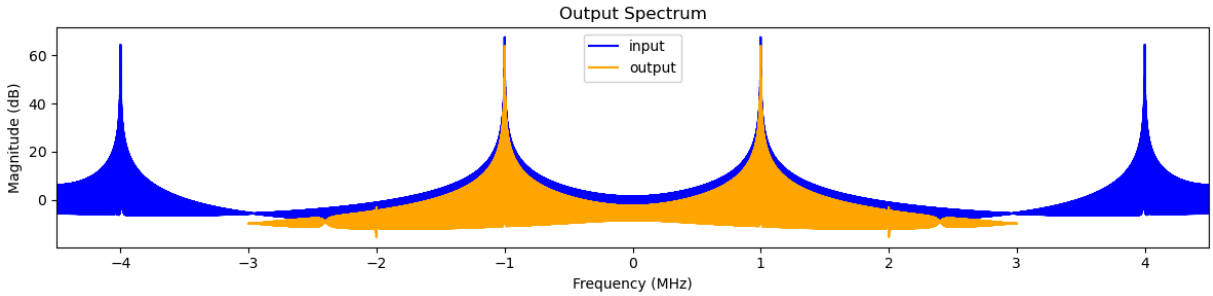

Single tone 2.4 MHz - Time domain (200 samples)
Input
Output
Time (µs)

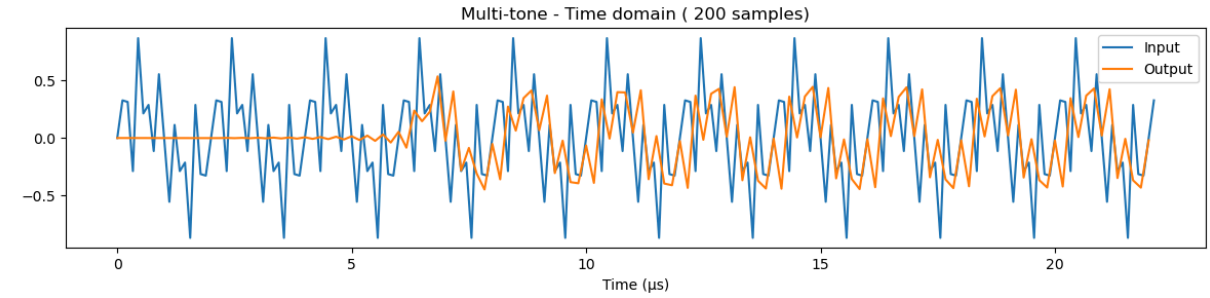

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

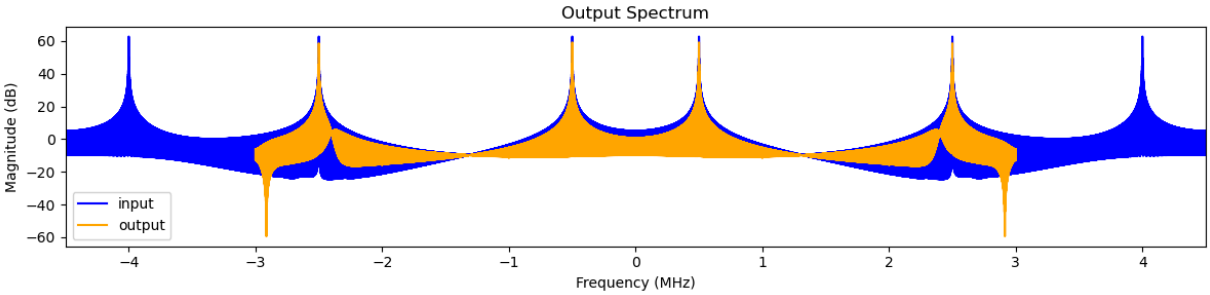

Single tone .5 MHz - Time domain (200 samples)
Input
Output
Time (µs)


Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

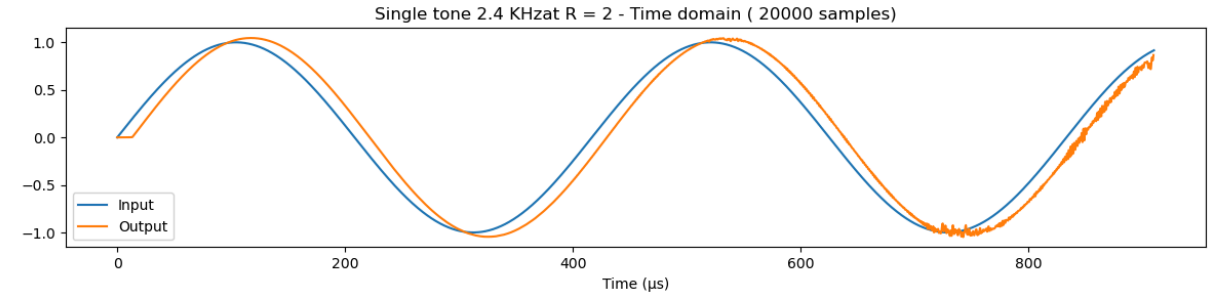

Two tones (1 MHz + 4 MHz) - Time domain (200 samples)
Input
Output
Time (µs)

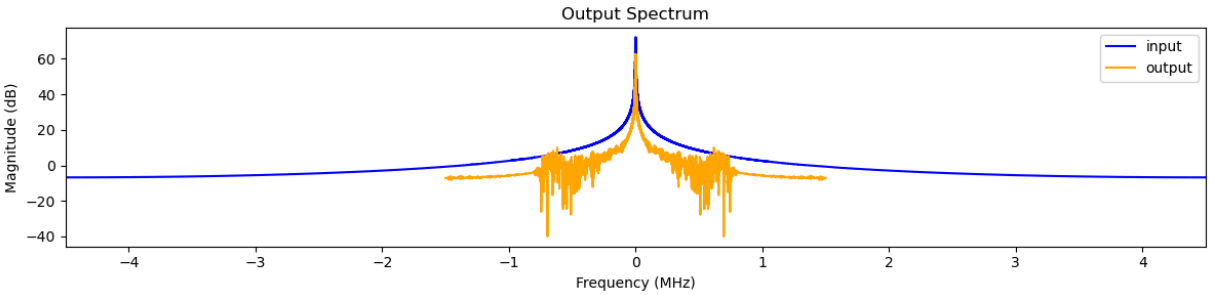

Output Spectrum
input
output
Magnitude (dB)
Frequency (MHz)

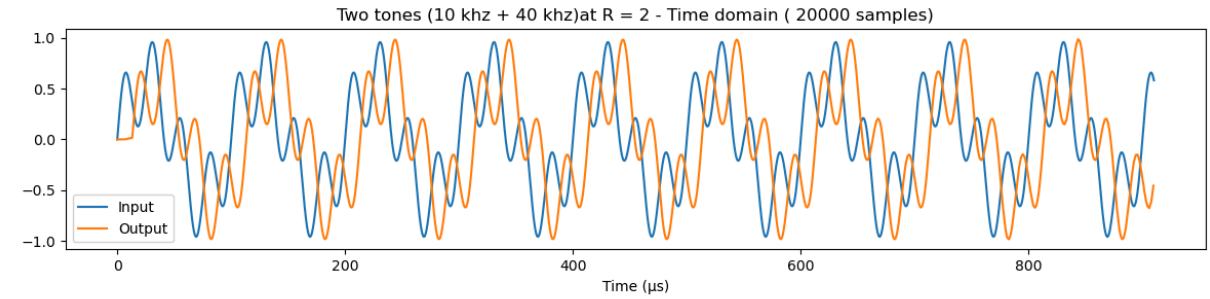


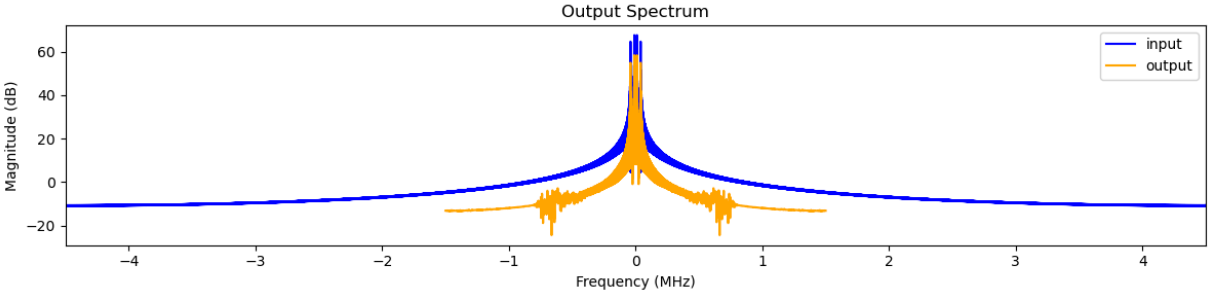


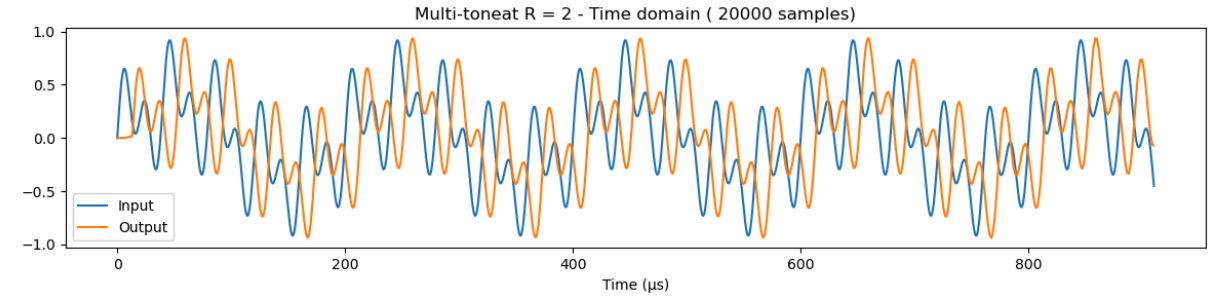
stop_2 low freq sig at R = 2

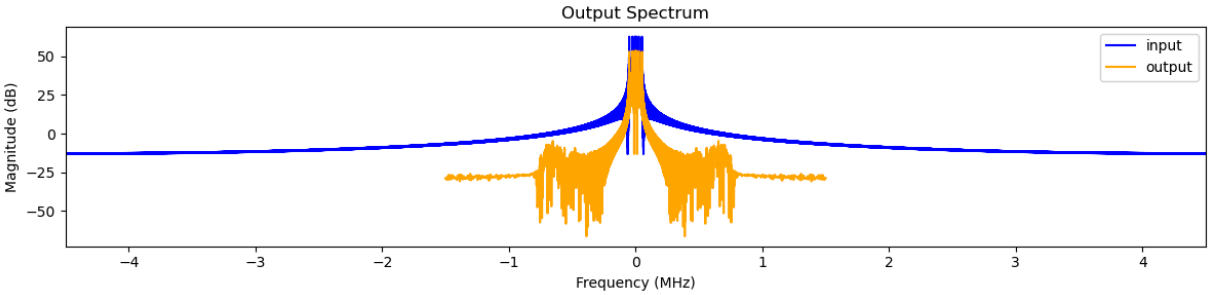




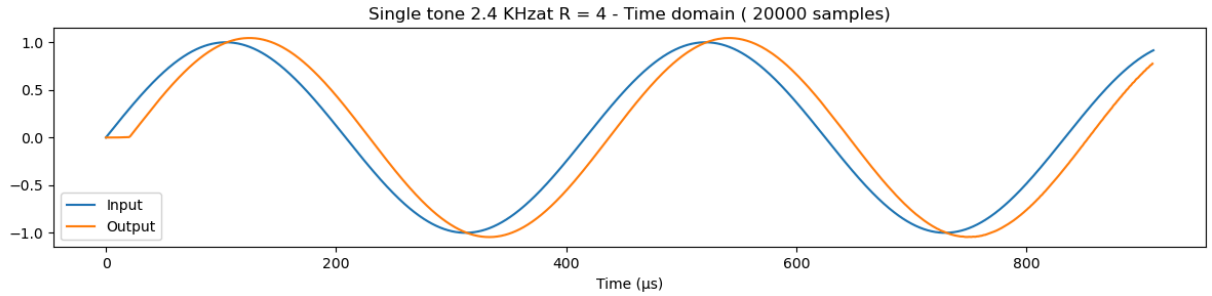


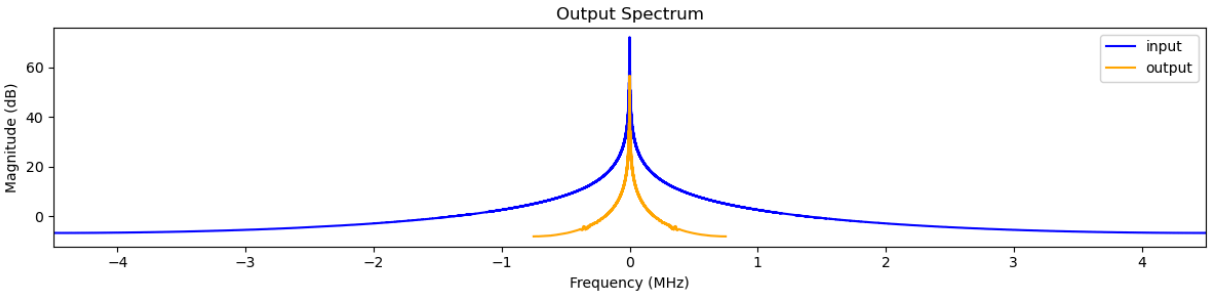


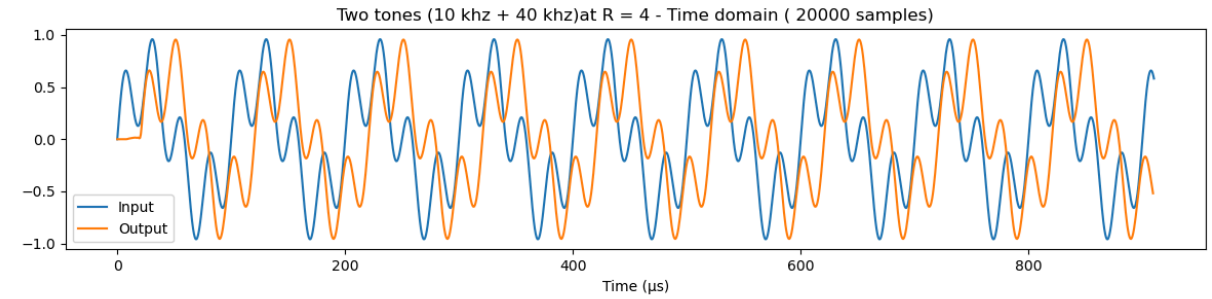


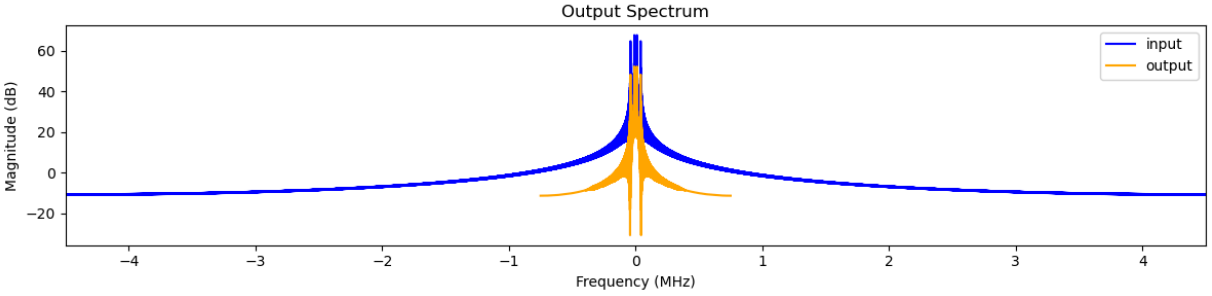


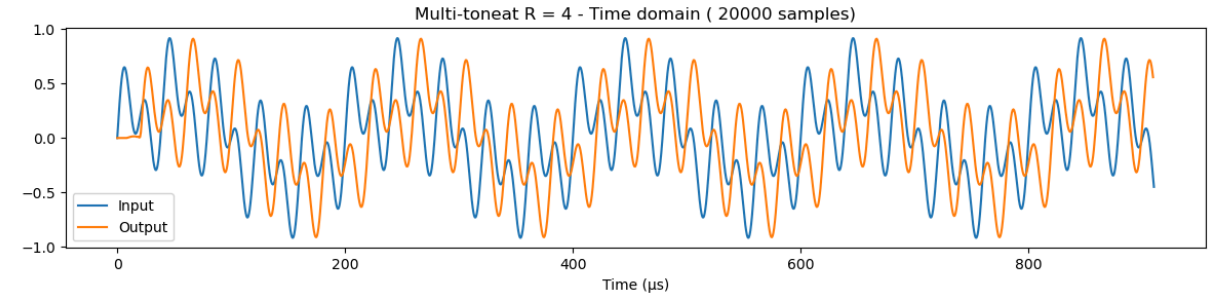
stop_3 low freq sig at R = 4

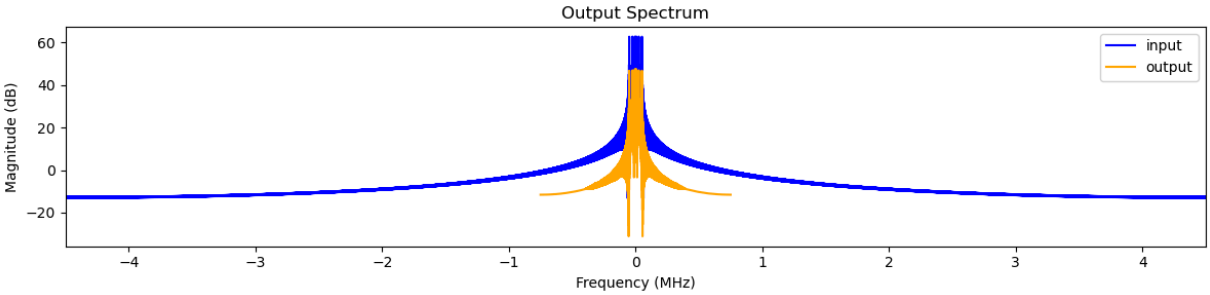




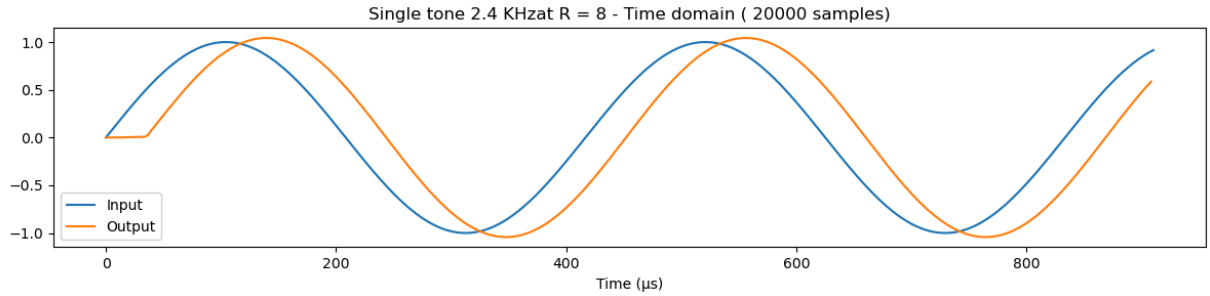


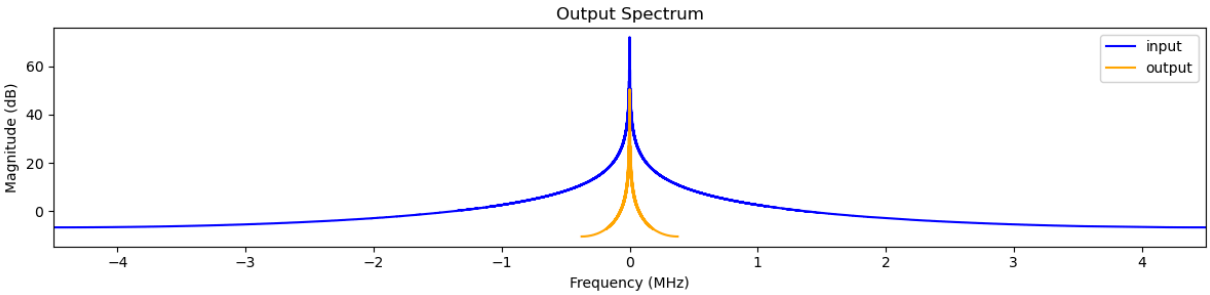


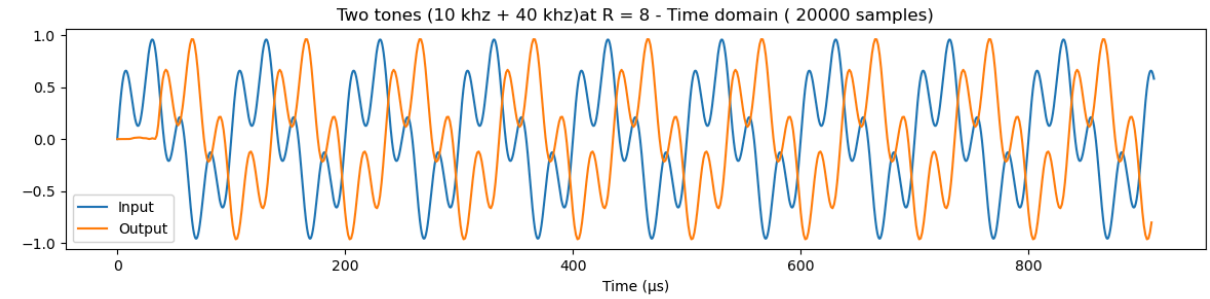


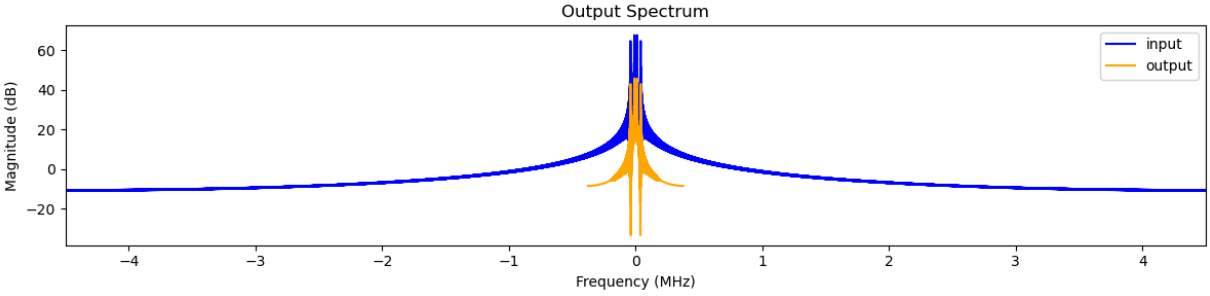


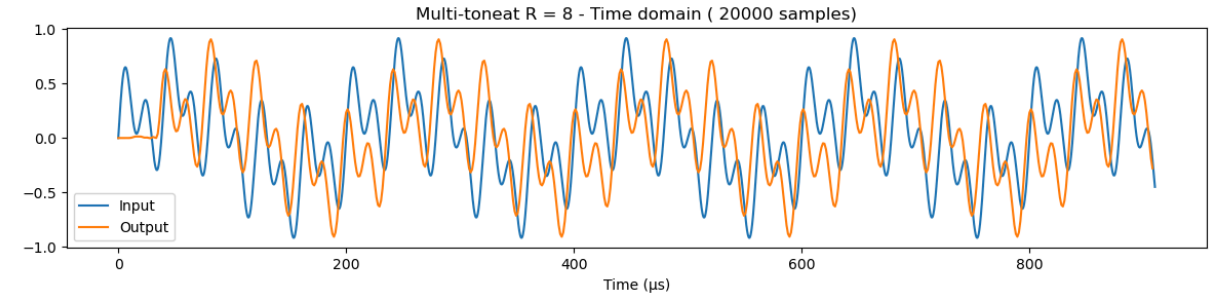
stop_4 low freq sig at R = 8

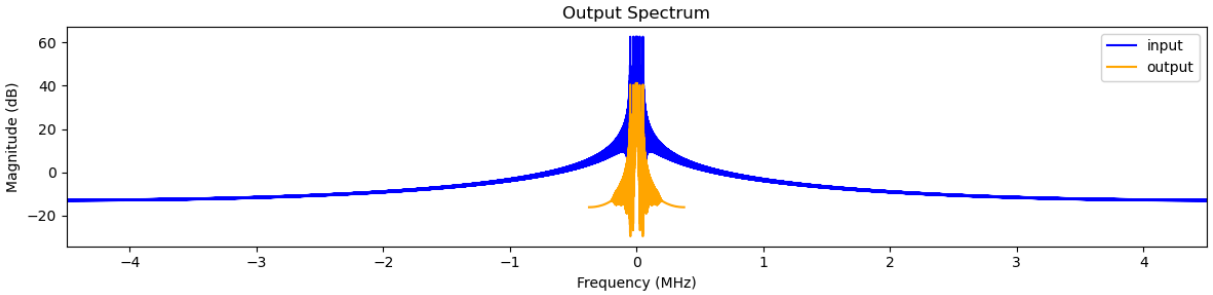




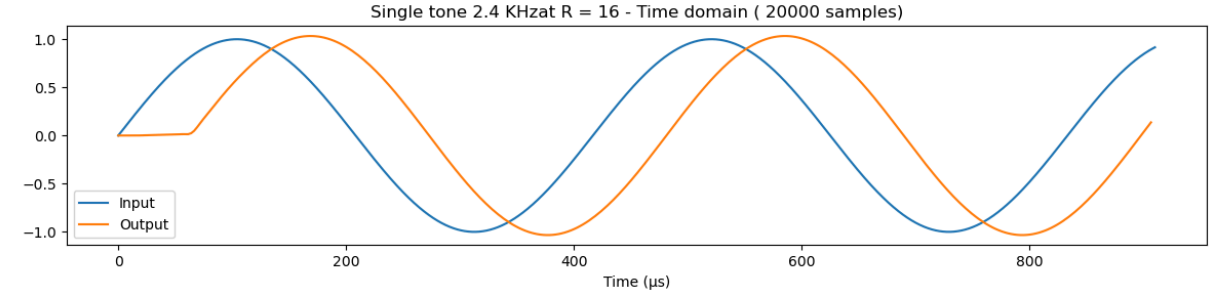


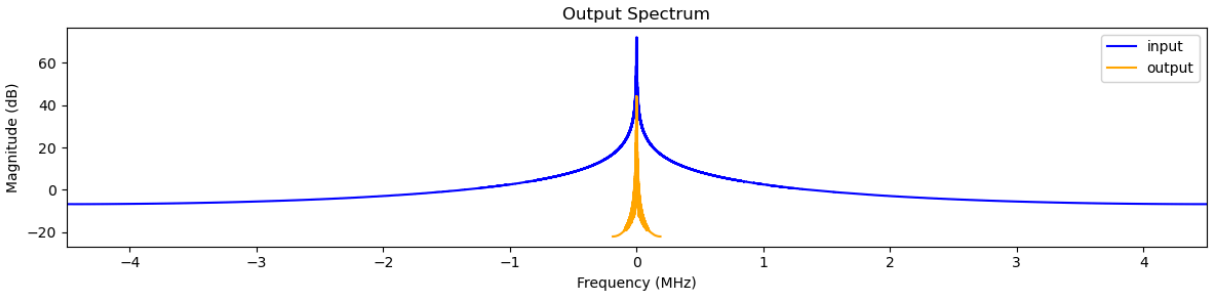


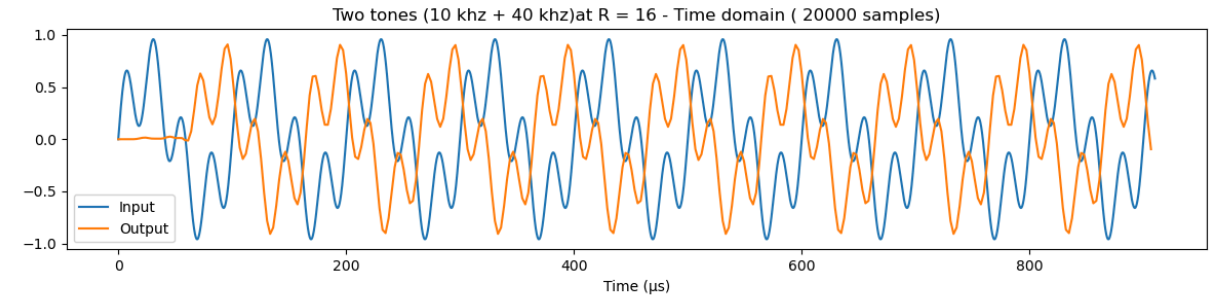


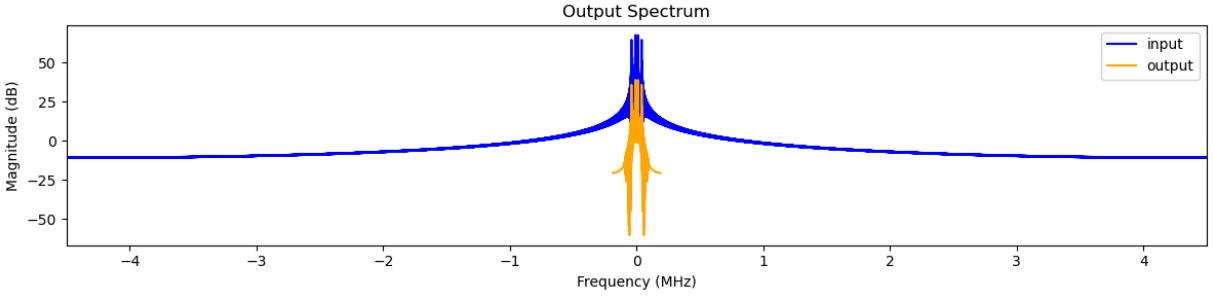


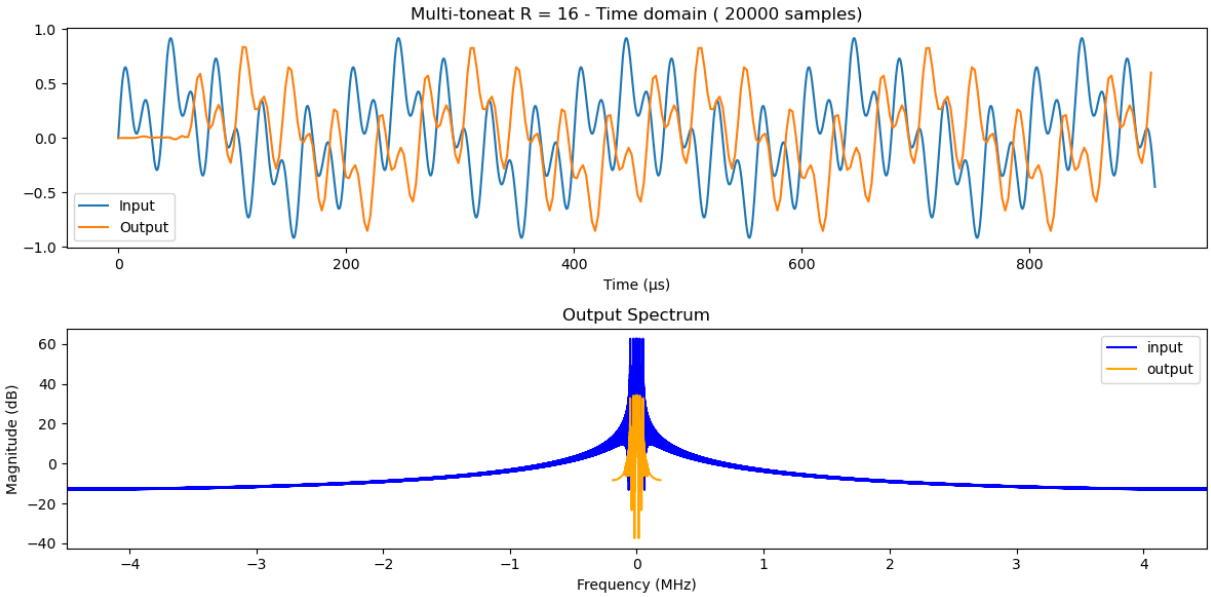
stop_5 low freq sig at R = 16











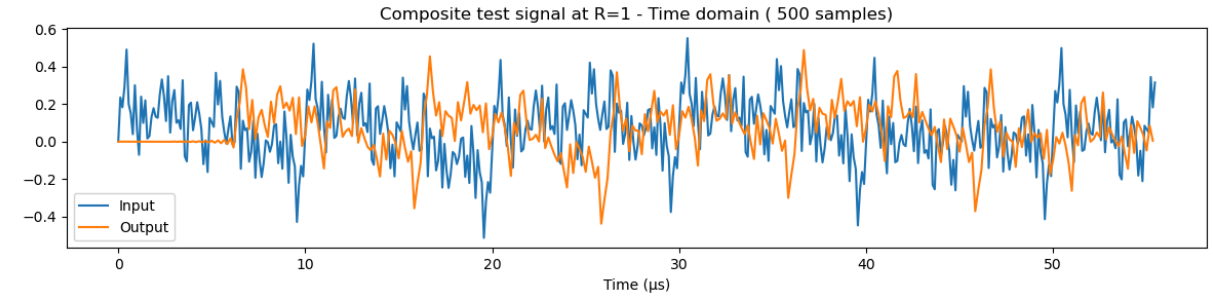
exhastive testbench

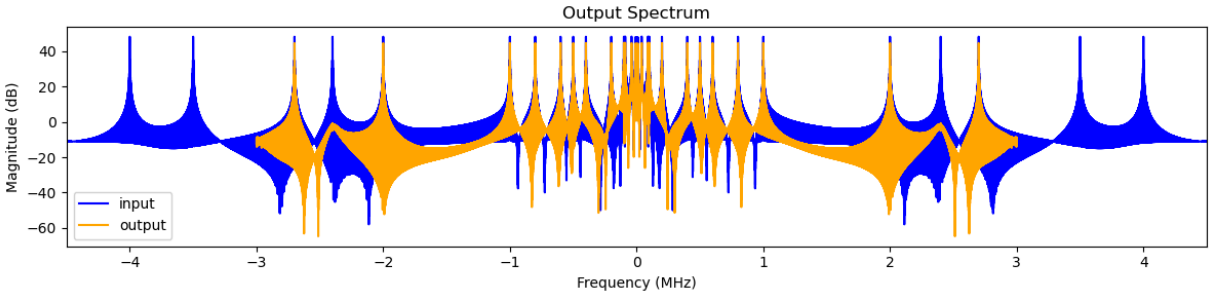
In [127…

```
# ----------------- TESTING: Composite signal -----------------

# Frequencies to test (Hz)
# - Low freq: below final output
# - High freq: near CIC nulls / aliasing points
test_freqs = [
    2.4e3,    # low
    1e4, 4e4, 9e4 ,# low
    1e5, 2e5, 4e5, 5e5, 6e5, 8e5,   # mid
    1e6, 2.4e6, 2.7e6, 3.5e6, 5e6 , 7e6 # high / near CIC attenuation points
]

# Generate single composite signal
sig = sum(np.sin(2*np.pi*f*t) for f in test_freqs) / len(test_freqs)

# Decimation factors to test
R_list = [1, 2, 4, 8, 16]

# Expected behavior dictionary for annotation
expected_behavior = {
    1: "No decimation, all frequencies above 3 MHz and at 2.4 MHz attenuated
    2: "Frequencies above 1.5 MHz attenuated by CIC, others pass",
    4: "Frequencies above 0.75 MHz attenuated, aliasing may occur",
    8: "Frequencies above 0.375 MHz attenuated, strong aliasing for >0.75 MH
    16:"Frequencies above 0.1875 MHz strongly attenuated, only very low freq
}

# Loop over R
for R in R_list:
    print(f"\n--- Decimation R = {R} ---")
    print("Expected:", expected_behavior[R])

    y = dfe_integrated(sig, R)
```

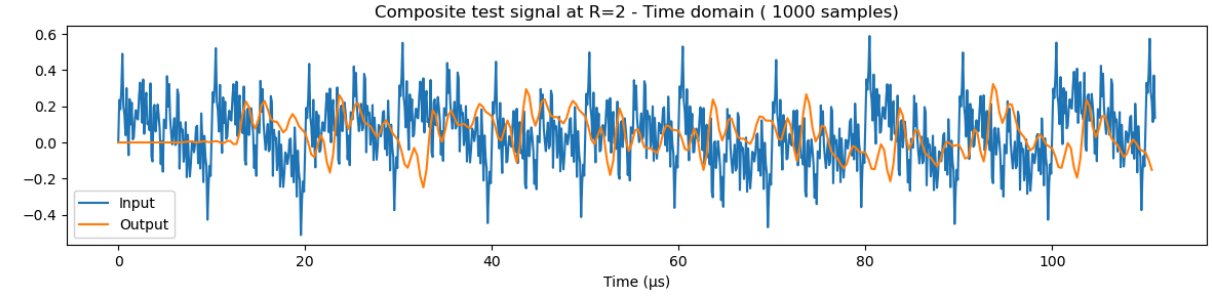
```
    fs_out_R = 6e6 / R
    plot_time_freq(sig, y, 9e6, fs_out_R, f"Composite test signal at R={R}",
```

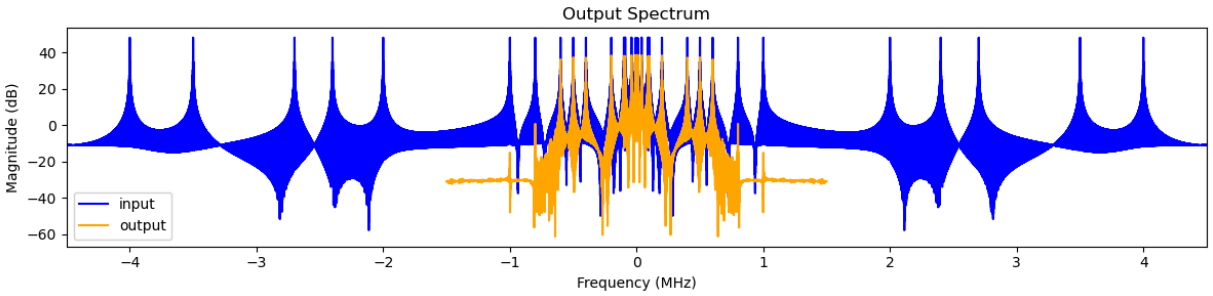
```
--- Decimation R = 1 ---
Expected: No decimation, all frequencies above 3 MHz and at 2.4 MHz attenuat
ed by polyphase resampler pass
```



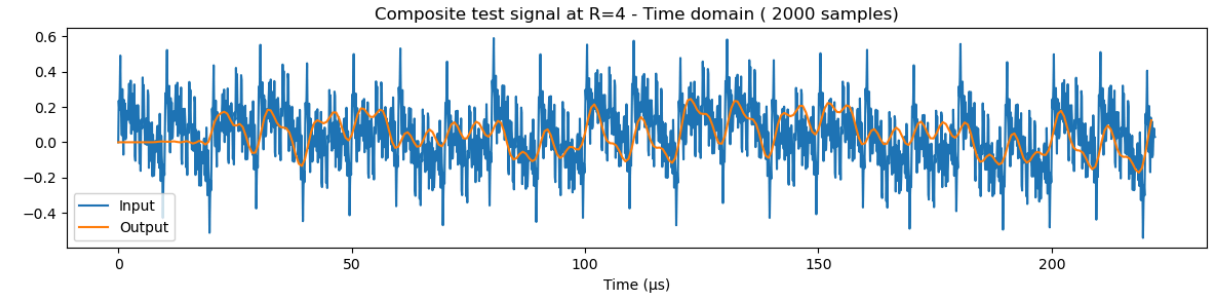


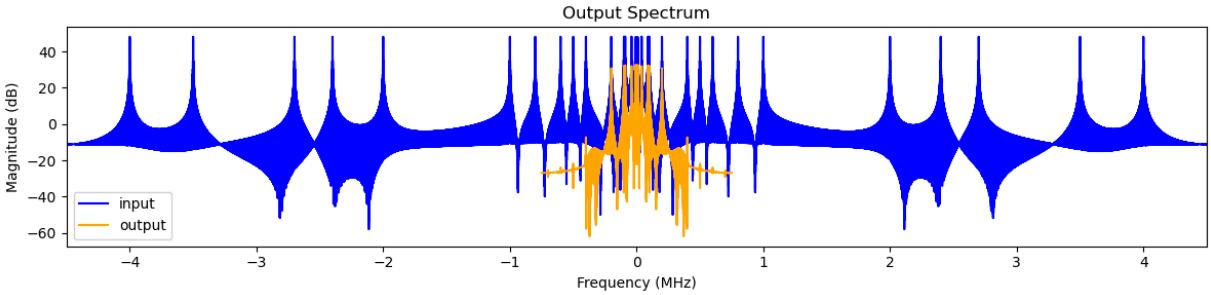

```
--- Decimation R = 2 ---
Expected: Frequencies above 1.5 MHz attenuated by CIC, others pass
```



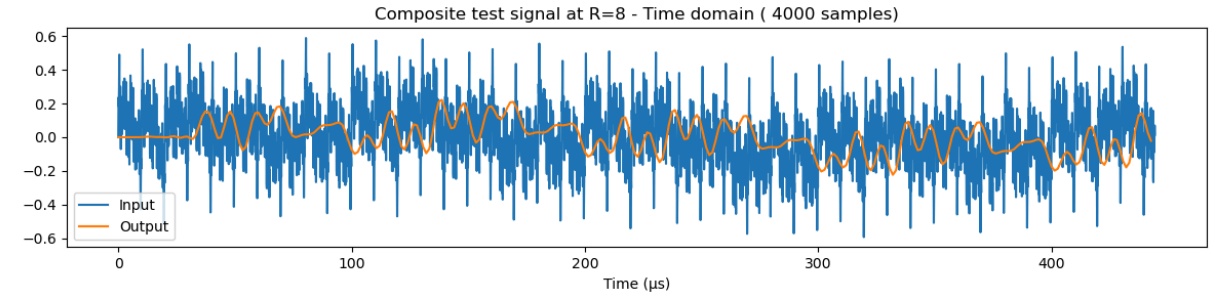


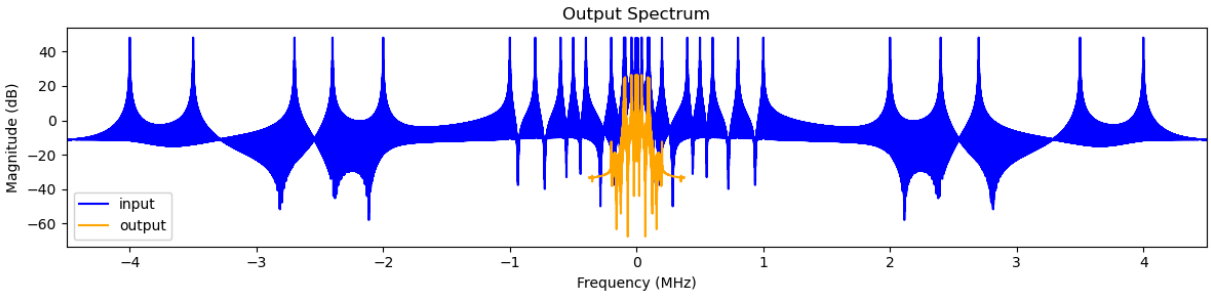

```
--- Decimation R = 4 ---
Expected: Frequencies above 0.75 MHz attenuated, aliasing may occur
```



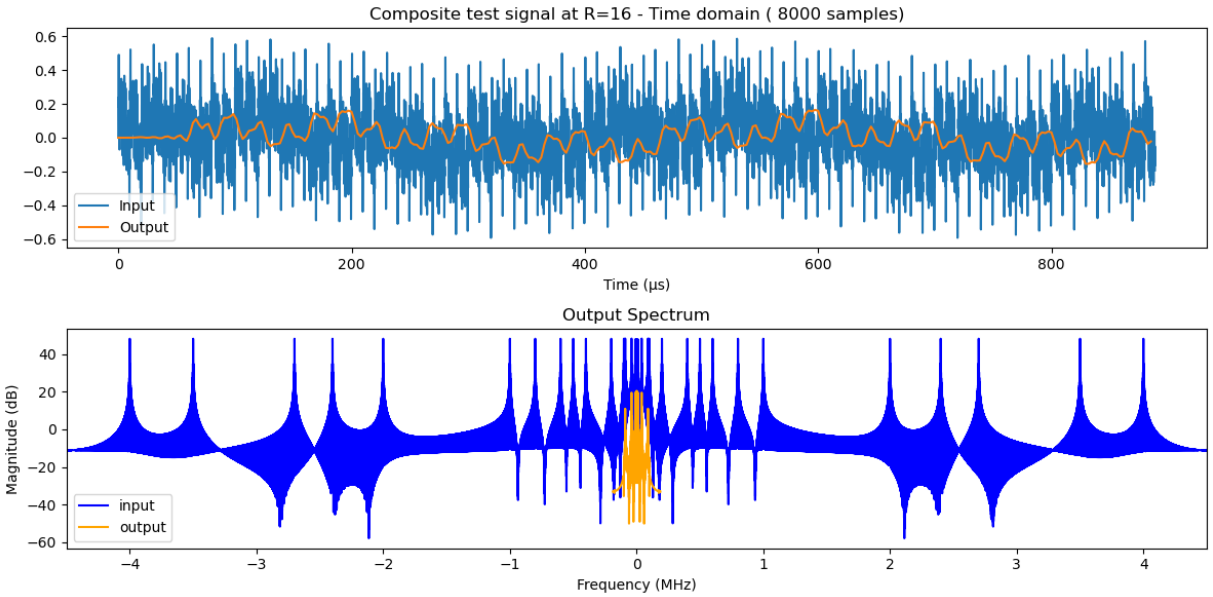


--- Decimation R = 8 ---
Expected: Frequencies above 0.375 MHz attenuated, strong aliasing for >0.75 MHz





--- Decimation R = 16 ---
Expected: Frequencies above 0.1875 MHz strongly attenuated, only very low frequencies survive


Composite test signal at R=16 - Time domain (8000 samples)
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Input
Output
0
200
400
600
800
Time (µs)

Output Spectrum
Magnitude (dB)
40
20
0
−20
−40
−60
input
output
−4
−3
−2
−1
0
1
2
3
4
Frequency (MHz)