Fruga faza projekta
Uroš Minoski 2019/0135

June 11, 2023

[26]:
```
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt
import numpy as np
import math as m
import scipy.signal as signal


def plotSpectrum(x, xlabel = "", ylabel="", title="", show=True):
    n = len(x)
    plt.plot(np.fft.fftshift(np.fft.fftfreq(n)),np.fft.fftshift(abs(np.fft.
 ↪fft(x))));
    plt.xlabel(xlabel); plt.ylabel(ylabel); plt.title(title);
    if show:
        plt.show()

def stemSpectrum(x, xlabel = "", ylabel="", title="", show=True):
    n = len(x)
    F = np.fft.fftshift(np.fft.fftfreq(n))
    X = np.fft.fftshift(abs(np.fft.fft(x)))
    plt.stem(F, X)#, use_line_collection=True);
    plt.xlabel(xlabel); plt.ylabel(ylabel); plt.title(title);
    if show:
        plt.show()

def plotSignal(x, xlabel = "", ylabel="", title="", show=True):
    plt.plot(x);
    plt.xlabel(xlabel); plt.ylabel(ylabel); plt.title(title);
    if show:
        plt.show()

def stemSignal(x, xlabel = "", ylabel="", title="", show=True):
    plt.stem(x)# ,use_line_collection=True);
```

```
    plt.xlabel(xlabel); plt.ylabel(ylabel); plt.title(title);
    if show:
        plt.show()
```

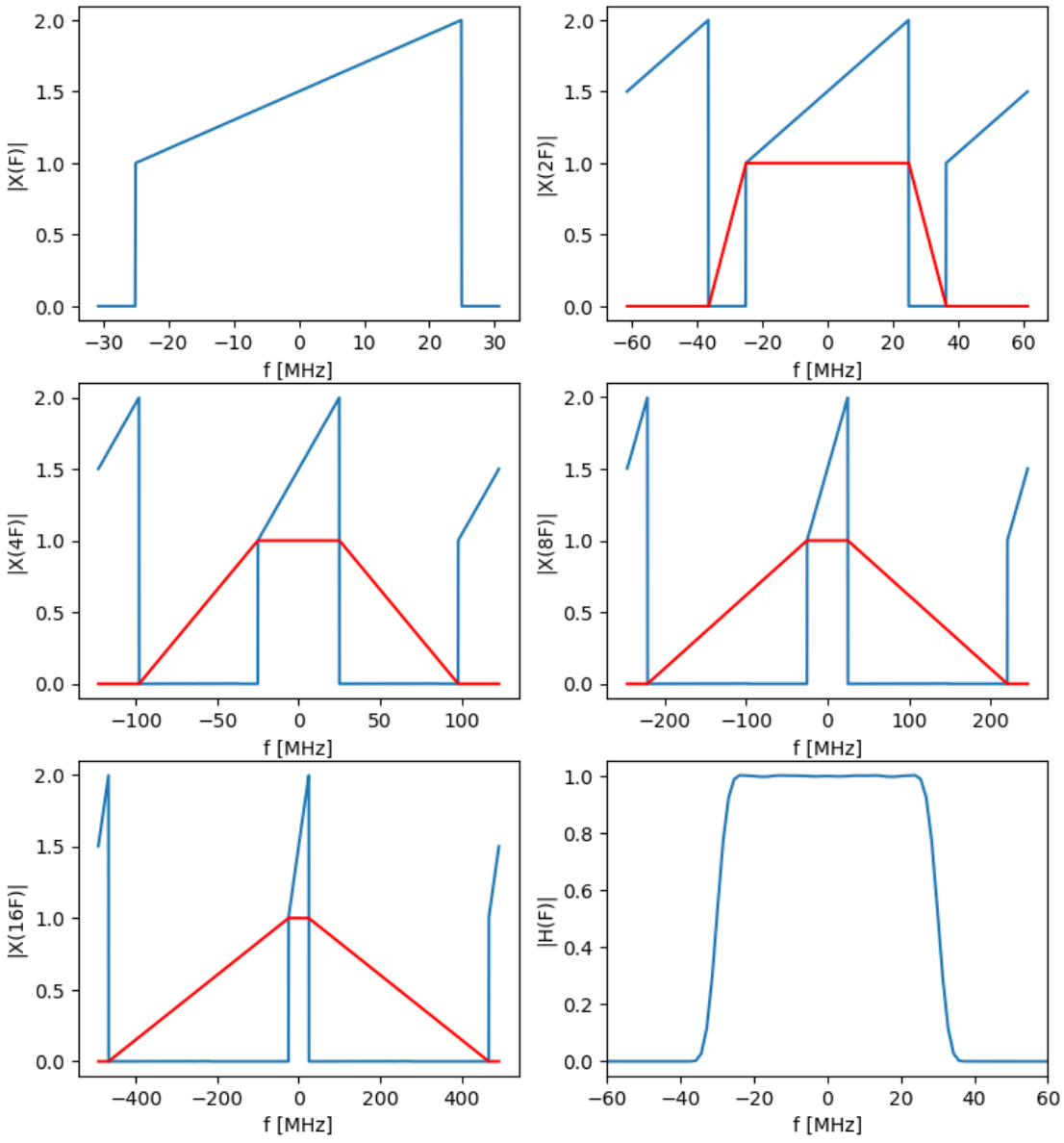
[27]:
```
def readSamples(fileName):
    samples = []
    inFile = open(fileName, "r")
    for line in inFile:
        samples.append( complex(line) )
    return np.array(samples)
```

I

[28]:
```
from remezlp import *

fmax = 25e6
fs = 61.44e6
Fmax = fmax / fs

testsignal = readSamples("testsignal.txt")

AdB = 60.0
Amax = 20.0*np.log10(1.0+10.0**(-AdB/20.0))
Amin = 20.0*np.log10(1.0-10.0**(-AdB/20.0))
deltaPass = 10.0**(-AdB/20.0)
deltaStop = deltaPass

Fpass = [0, Fmax / 2, Fmax / 4, Fmax / 8, Fmax / 16]
Fstop = [0, (1 - Fmax) / 2, (1 - Fmax / 2) / 4, (1 - Fmax / 4) / 8, (1 - Fmax /␣
 ↪8) / 16]

h_tot = []

xi = testsignal

mul = [0, 2, 4, 8, 16]

plt.subplots(3,2, figsize = (9, 10));

# Attenuation in stopband in dB.
Adb = 60

# Pass frequencies for half-band filters in every stage of
# multistage implementation.
Fpass = [0, 0.203, 0.102, 0.051, 0.025]
```

```
for i in range(6):

    if(i == 0):
        N = len(testsignal)
        freqs = (np.arange(N)/(N) - 0.5) *fs/1e6
        plt.subplot(3,2,1)
        plt.plot(freqs, np.fft.fftshift(abs(np.fft.fft(testsignal))))
        plt.xlabel("f [MHz]")
        plt.ylabel("|X(F)|");

    elif(i == 5):
        hh = []

        h_new = np.zeros(len(h_tot[0])*8, dtype=complex) # argument␣
↪dtype=complex je obavezan!
        h_new[::8] = h_tot[0]
        hh.append(h_new)
        for i in range(1,4):
            h_tmp = np.zeros(len(h_tot[i])*int(8/(2**i)),dtype=complex) #␣
↪argument dtype=complex je obavezan!
            h_tmp[::int(8/(2**i))] = h_tot[i]
            hh.append(h_tmp)
            h_new = np.convolve(h_new, h_tmp)

        plt.subplot(3,2,6)

        N = len(h_new)
        freqs = freqs = (np.arange(N)/(N) - 0.5) * 16*fs/1e6
        plt.plot(freqs, np.fft.fftshift(abs(np.fft.fft(h_new))))
        plt.xlim([-60, 60])
        plt.xlabel("f [MHz]")
        plt.ylabel("|H(F)|");


    else:
        oversampled_signal = np.zeros(len(xi)*2,dtype=complex) # argument␣
↪dtype=complex je obavezan!
        oversampled_signal[::2] = xi

        N = len(xi)

        h = remezlp(Fpass[i], Fstop[i], deltaPass, deltaStop,␣
↪nPoints=8192*8,Nmax=500)
        h_tot.append(h)

        # Test signal se periodicno produzava za len(h)/2 odbiraka, koliko␣
↪iznosi kasnjenje filtra.
```

```
        oversampled_signal_perext = np.concatenate((oversampled_signal,␣
↪oversampled_signal[0:int(len(h/2))]))

        xi = signal.lfilter(h, 1, oversampled_signal_perext)
        xi = xi[len(h):]

        freqs = (np.arange(N*2)/(N*2) - 0.5) * mul[i]*fs/1e6

        M = mul[i]

        plt.subplot(3,2,i+1)
        plt.plot(freqs, np.fft.fftshift(abs(np.fft.fft(oversampled_signal))))
        plt.plot([-M*fs/2/1e6, -M*Fstop[i]*mul[i]/2*fs/1e6, -M*Fpass[i]*fs/1e6,␣
↪M*Fpass[i]*fs/1e6, M*Fstop[i]*mul[i]/2*fs/1e6, M*fs/2/1e6],[0, 0, 1, 1, 0, 0],␣
↪'r')
        plt.xlabel("f [MHz]")
        plt.ylabel("|X(" + str(mul[i]) + "F)|");

# plt.savefig('multistage.pdf')
```

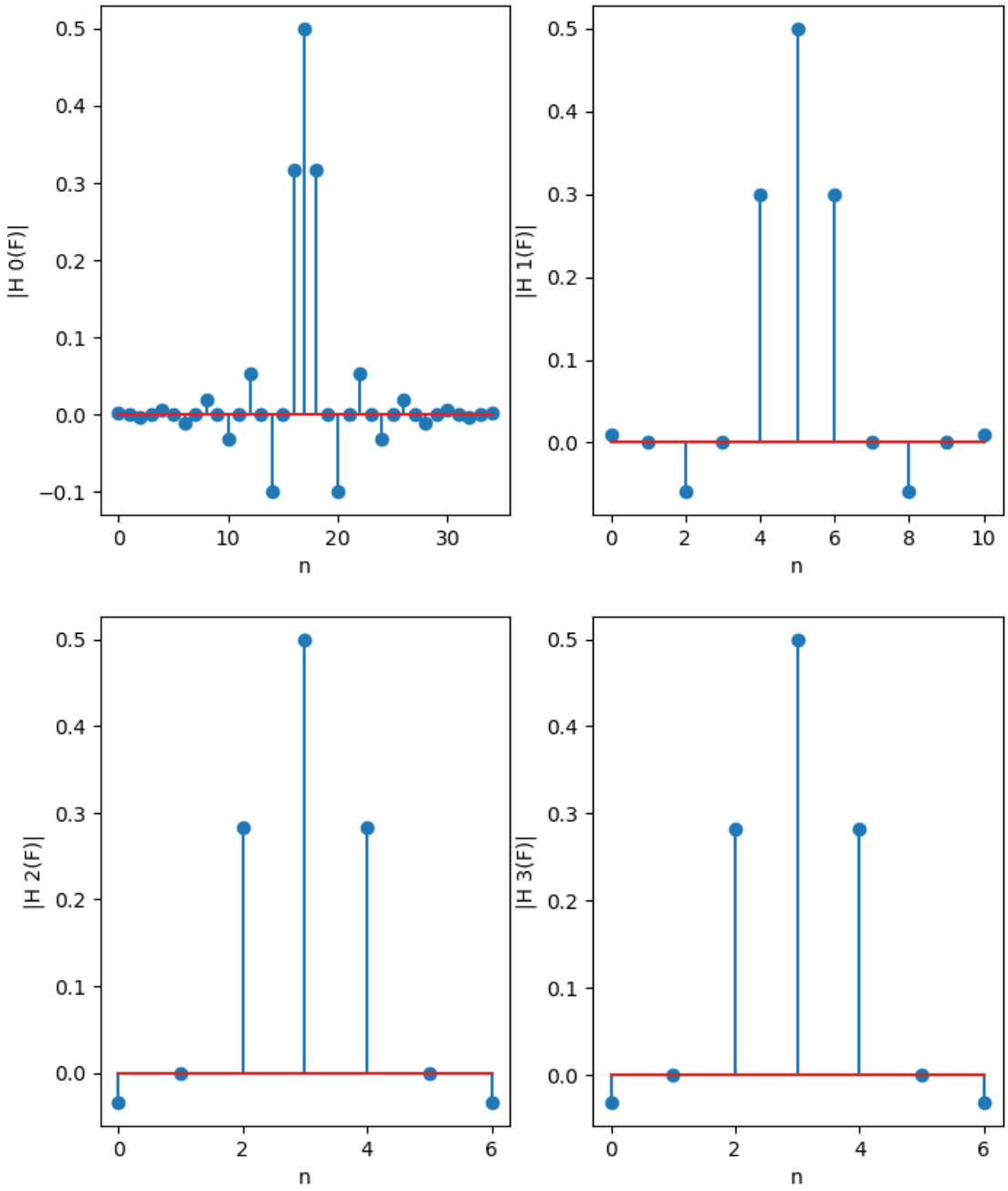


FIR half-band filters for every stage of multistage implementation. As can be seen, the most complex filter is the first one with order of 43 and most relaxed are the last two with orders of 6.

[29]:
```
from remezhb import*

# Attenuation in stopband in dB.
Adb = 60

# Pass frequencies for half-band filters in every stage of
# multistage implementation.
Fpass = [0.203, 0.102, 0.051, 0.025]
```

```
plt.subplots(2,2, figsize = (8, 10));

for i in range(len(Fpass)):
    h = remezhb(Fpass[i], Adb)
    print("Order of " + str(i + 1) + ". filter is " + str(len(h) - 1))
    plt.subplot(2,2,i+1)
    plt.stem(h);

    plt.xlabel("n")
    plt.ylabel("|H " + str(i) + "(F)|");
```

```
Order of 1. filter is 34
Order of 2. filter is 10
Order of 3. filter is 6
Order of 4. filter is 6
```



[30]:
```
from remezhb import*

def find_firs(Fpass, Adb, points):
    h_tot = []

    # plt.subplots(2,2, figsize = (8, 10));
    HdB = []
```

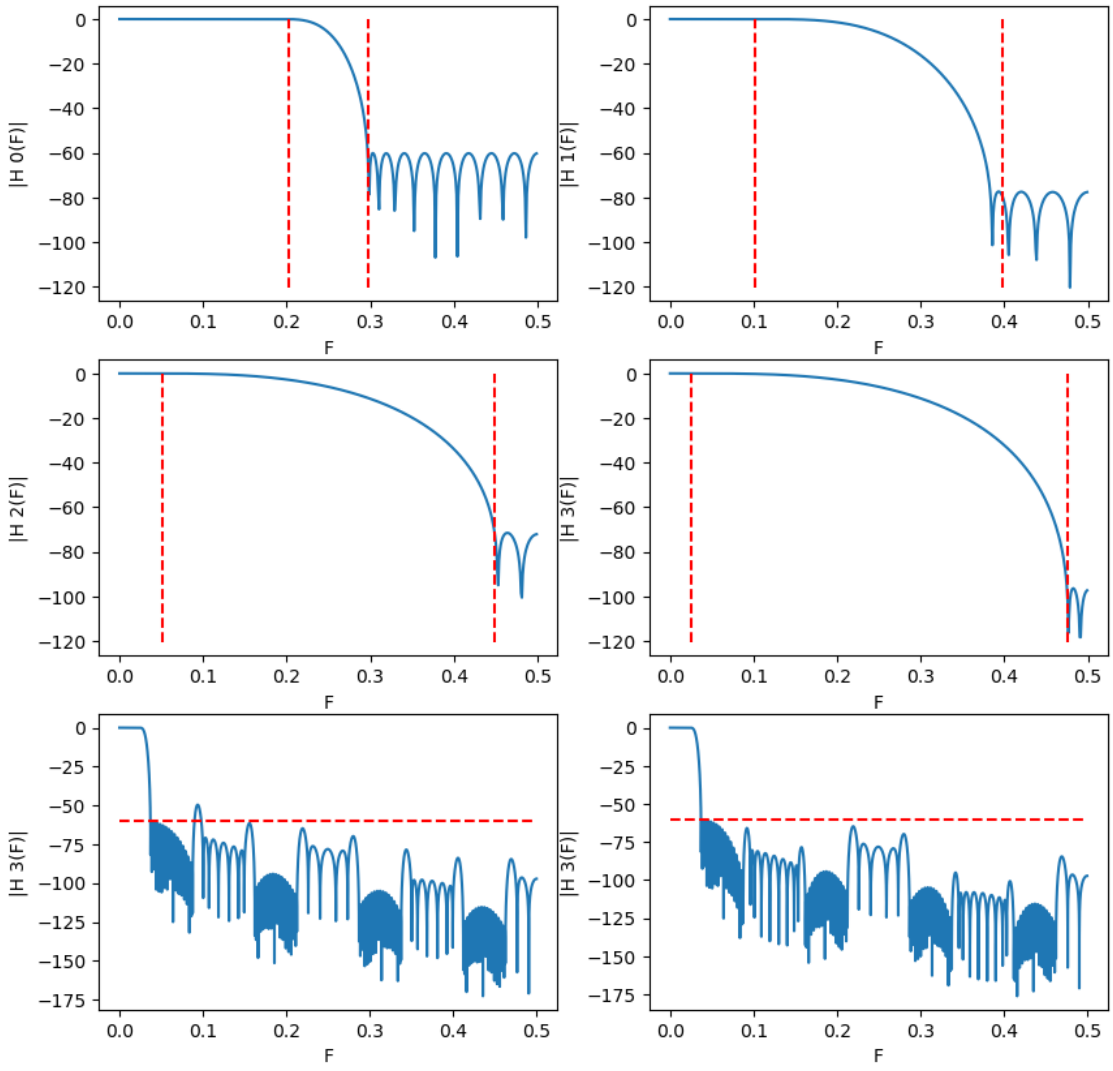
```
    h_tot = np.zeros(M)
    h_tot[0] = 1
    for i in range(len(Fpass)):
        h = np.zeros(points)
        h_HB = remezhb(Fpass[i], Adb)
        h[:len(h_HB)] = h_HB
        H = abs(np.fft.fft(h))
        HdB.append(20 * np.log10(H))

        h_tmp = np.zeros(len(h) * int(8 / 2**i), dtype = complex)
        h_tmp[::int(8/(2**i))] = h
        h_tot = np.convolve(h_tot, h_tmp)

    HdB_tot = 20 * np.log10(abs(np.fft.fft(h_tot)))

    return HdB, HdB_tot

def plot_firs(HdB, HdB_tot_init, HdB_tot_2):
    plt.subplots(3,2, figsize = (10, 10));
    for i in range(4):
        plt.subplot(3,2,i+1)
        freqs = np.arange(M) / M
        plt.plot(freqs[:M//2], HdB[i][:M//2])
        plt.plot([Fpass[i], Fpass[i]], [np.min(HdB), 0], 'r--');
        plt.plot([0.5 - Fpass[i], 0.5 - Fpass[i]], [np.min(HdB), 0], 'r--');
        plt.xlabel("F")
        plt.ylabel("|H " + str(i) + "(F)|");

    plt.subplot(3, 2, 5)
    freqs = (np.arange(16*M)) / (16*M)
    plt.plot(freqs[:16*M//2], HdB_tot_init[:16*M//2])
    plt.plot([0, 0.5], [-Adb, -Adb], 'r--');
    plt.xlabel("F")
    plt.ylabel("|H " + str(i) + "(F)|");

    plt.subplot(3, 2, 6)
    freqs = (np.arange(16*M)) / (16*M)
    plt.plot(freqs[:16*M//2], HdB_tot_2[:16*M//2])
    plt.plot([0, 0.5], [-Adb, -Adb], 'r--');
    plt.xlabel("F")
    plt.ylabel("|H " + str(i) + "(F)|");

    plt.savefig('multistage_FIR.pdf')
```

First four images shows spectre of every filter in multistage implementation. Complexities of filters are better seen here. Red vertical lines shows pass and stop digital frequencies.

It can be seen that euivalent filter fails to attenuate signal for 60 dB everywhere in stopband. The

reason for that is overlaping spectre of second and first filter. Solution would be to increase pass frequency of second filter for 0.015. That will add 2 more coefficients to filter and increase it's order to 14. Efect of that can be seen in the last image.

[31]:
```
# Attenuation in stopband in dB.
Adb = 60

# Pass frequencies for half-band filters in every stage of
# multistage implementation.
Fpass_init = [0.203, 0.102, 0.051, 0.025]
Fpass_2 = [0.203, 0.117, 0.051, 0.025]

M = 1024

HdB, HdB_tot_init = find_firs(Fpass_init, Adb, M)
HdB, HdB_tot_2 = find_firs(Fpass_2, Adb, M)
plot_firs(HdB, HdB_tot_init, HdB_tot_2)

# 2 dodatna odbirka, tj jedan razliciti, red je 14
```



II

[32]:
```
import math as m

def CORDIC_iteration(x, y, z, i, mode, scaled=False):
    """
    Calculate one iteration of CORDIC algorithm.
    x, y, z - inputs
    i - iteration number
    mode - "rotation" for CORDIC in rotation mode,
           otherwise calculates in vector mode
    scaled - True or False. If False, scaling factor is 1.
    Returns (x, y, z)
    """
```

```
    sgn = lambda a: (a>=0) - (a<0)
    if mode=="rotation":
        sigma = sgn(z)
    else:
        sigma = -sgn(y)
    xnew = x - sigma*y*2**(-i)
    ynew = y + sigma*x*2**(-i)
    z = z - sigma * m.atan(2**(-i))
    if scaled:
        ki = CORDIC_Ki(i)
        xnew *= ki
        ynew *= ki
    return (xnew, ynew, z)

def CORDIC_Ki(i):
    """
    Calculate the scaling factor for i-th iteration
    """
    k = 1.0/m.sqrt(1.0+2.0**(-2*i))
    return k

def CORDIC_Kn(n):
    """
    Calculate the overall scaling factor for n iterations
    """
    k = 1
    for i in range(0,n):
        k *= CORDIC_Ki(i)
    return k

def print_iteration(i, x, y, z):
    """
    Print intermediate results
    """
    res = '{0: <3}'.format(str(i))
    res += "{:12.8f} {:12.8f} {:12.8f}".format(x,y,z)
    print(res)

def prerotation(x, y, z):
    if((z >= m.pi / 2) and (z < 3 * m.pi / 2)):
        return -x, -y, (z - m.pi)
    elif((z >= 3 * m.pi / 2) and (z < 2 * m.pi)):
        return y, -x, (z - 3 * m.pi / 2)
    else:
        return x, y, z
```

[33]:
```
N    = 512
fs   = 16 * 61.44e6
f0   = (N//4) * fs / N

W    = 2 * m.pi * f0 / fs

n      = 10
angle  = 0
scaled = True
x_out  = []
y_out  = []

for i in range(N):
    if(i != 0):
        angle += W
    if(angle >= 2 * m.pi):
        angle -= 2 * m.pi

#     x = CORDIC_Kn(n)
    x = 1
    y = 0

    x, y, z = prerotation(x, y, angle)

    for j in range(0,n):
        x, y, z = CORDIC_iteration(x, y, z, j, "rotation", scaled)

    x_out.append(x)
    y_out.append(y)
```

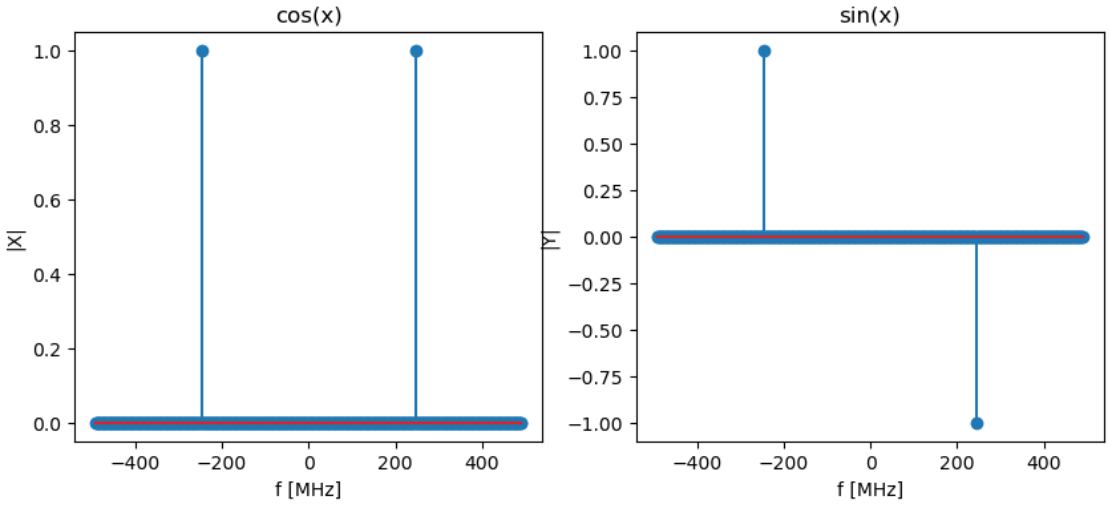
In order to get sinusoid signal at the output there is a need for NCO that will feed CORDIC mixer. Initial condition for NCO can be callculated using $W = 2\pi\frac{f_0}{f_s}$ where f_0 is desired frequency of output signal and initial condition for CORDIC algorithm is vector $(1, 0)^T$

Output of NCO feeds CORDIC mixer. That means that NCO's output represents angle of rotation of vector $(x, y)^T$. Angle must be contained in range $[0, 2\pi]$ and vector $(x, y)^T$ must be preorientated to ensure convergence of algorithm.

[34]:
```
N = len(x_out)
X = np.fft.fftshift(np.fft.fft(x_out))
Y = np.fft.fftshift(np.fft.fft(y_out))

plt.subplots(1,2, figsize = (10, 4));
freqs = (np.arange(N)/N - 0.5) * fs/1e6

plt.subplot(1,2,1)
plt.stem(freqs, X / np.max(X))
plt.xlabel("f [MHz]")
```

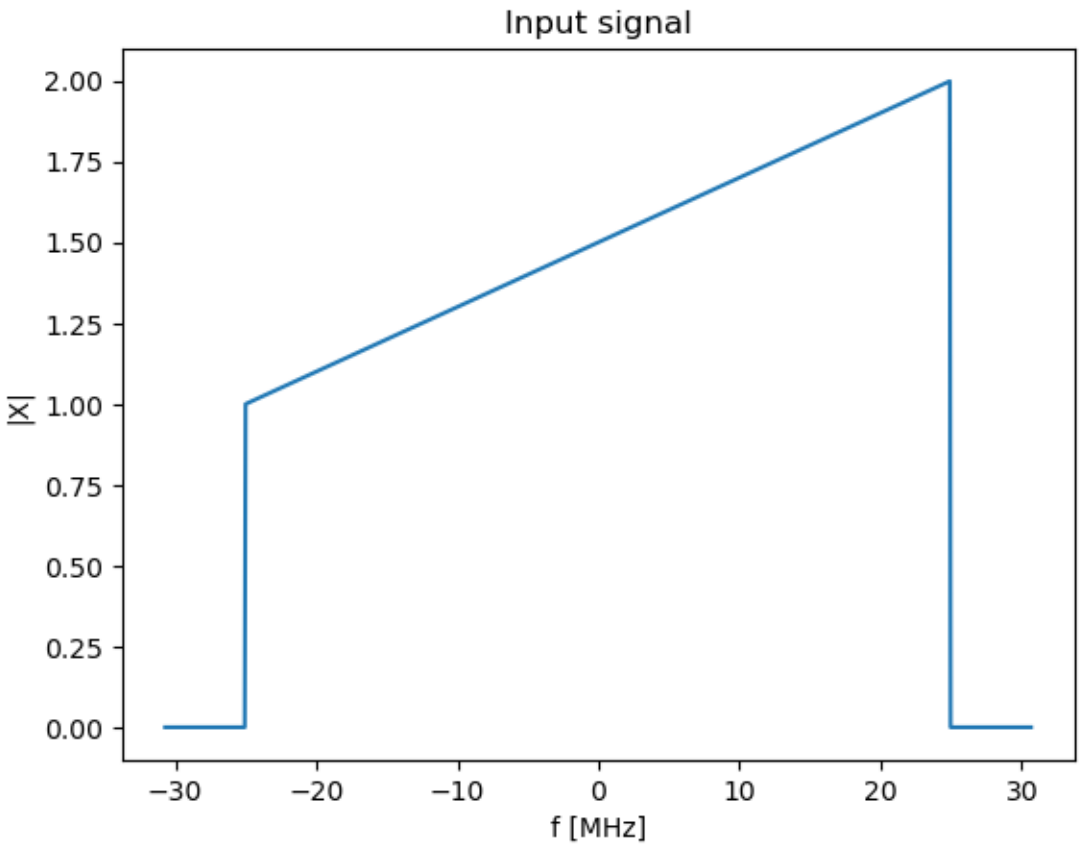
```
plt.ylabel("|X|");
plt.title("cos(x)");
plt.subplot(1,2,2)
plt.stem(freqs, Y / np.max(Y))
plt.xlabel("f [MHz]")
plt.ylabel("|Y|");
plt.title("sin(x)");

print("Output frequency is " + str(f0 / 1e6) + " MHz.")
```

```
Output frequency is 245.76 MHz.
```



III

[35]:
```
M = 16
fs = 61.44e6
testsignal = readSamples("testsignal.txt")

# x_tmp = np.e**(-1j * np.arange(len(testsignal)) * m.pi)
# testsignal = np.array(testsignal) * x_tmp

N = len(testsignal)
freqs = (np.arange(N)/N - 0.5) * fs/1e6
plt.plot(freqs, np.fft.fftshift(abs(np.fft.fft(testsignal))))
plt.xlabel("f [MHz]")
plt.ylabel("|X|");
plt.title("Input signal");
```



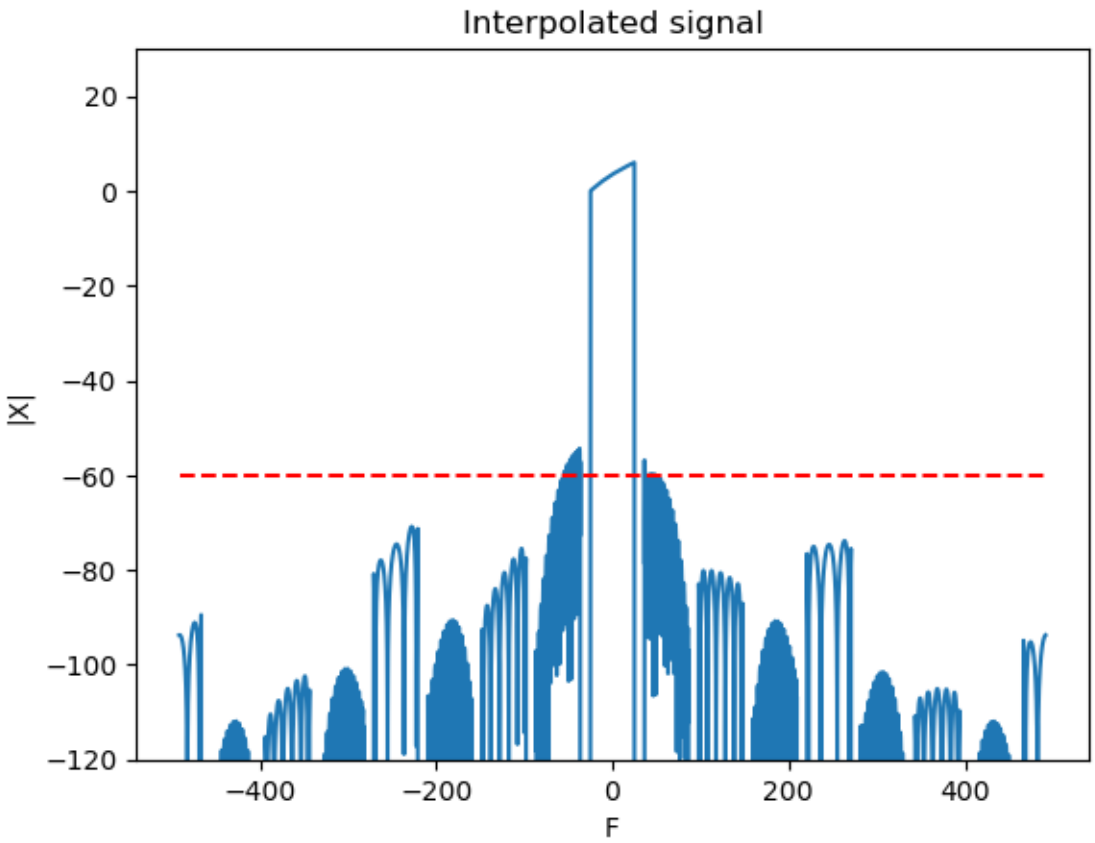
[36]:
```
def makePolyphase(coeff, M):
    # Split coefficients into polyphase components
    n = int(np.ceil(len(coeff)/M))
    tmp = np.zeros(n*M)
    tmp[:len(coeff)] = coeff
    polyCoeff = np.zeros((M,n))
    for i in range(M):
        polyCoeff[i,:] = tmp[i::M]
    return polyCoeff

def makeFIRs(Fpass, Adb):
    h = []
    for i in range(len(Fpass)):
        h.append(remezhb(Fpass[i], Adb))
    return np.array(h)

def filterPolyphase(x, firCoeff, I):
    FIRpoly = makePolyphase(firCoeff, I)
    xpoly = np.zeros(len(x)*I, dtype = complex)
    for i in range(I):
```

```
        tmp,zi = signal.lfilter(FIRpoly[i], 1.0, x,zi=np.
↪zeros(len(FIRpoly[i])-1))
        tmp,zi = signal.lfilter(FIRpoly[i], 1.0, x,zi=zi)
        xpoly[i::I] = tmp
    return xpoly

def multistagePolyphase(Fpass, Adb, x0, I):
    h = makeFIRs(Fpass, Adb)
    x1 = filterPolyphase(x0, h[0], I)
    x2 = filterPolyphase(x1, h[1], I)
    x3 = filterPolyphase(x2, h[2], I)
    x4 = filterPolyphase(x3, h[3], I)
    return x4
```

Interpolation is done in multistage polyphase implementation. Every stage increses sampling frequency by a factor of 2 so half band filters can be used.

Polyphase interpolation by a factor of 2 is done in a way that even and odd filter's coeffitients are separated in two polyphase structures. At the end is comutator that switches between outputs of polyphase structures. Comutator switches from the top down every time a sample is transmited through polyphase structure. When comutator reaches end, it starts from the top structure again. In that way there are not unnecessary operations with zero.

[37]:
```
Fpass = [0.203, 0.117, 0.051, 0.025]
Adb = 60
testsignal = readSamples("testsignal.txt")
I = 2

xFiltered = multistagePolyphase(Fpass, Adb, testsignal, 2)

freqs = (np.arange(N*M)/(N*M) - 0.5) * M*fs/1e6
XrdB = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(xFiltered))))
plt.plot(freqs, XrdB);
plt.xlabel("F"); plt.ylabel("|X|"); plt.title("Interpolated signal")
plt.plot([freqs[0], freqs[-1]], [-60, -60], 'r--');
plt.ylim(-120,30);
```



[38]:
```
import math as m

def CORDIC_iteration(x, y, z, i, mode, scaled=False):
    """
    Calculate one iteration of CORDIC algorithm.
    x, y, z - inputs
    i - iteration number
    mode - "rotation" for CORDIC in rotation mode,
           otherwise calculates in vector mode
    scaled - True or False. If False, scaling factor is 1.
    Returns (x, y, z)
    """
    sgn = lambda a: (a>=0) - (a<0)
    if mode=="rotation":
        sigma = sgn(z)
    else:
        sigma = -sgn(y)
    xnew = x - sigma*y*2**(-i)
    ynew = y + sigma*x*2**(-i)
    z = z - sigma * m.atan(2**(-i))
```

```
    if scaled:
        ki = CORDIC_Ki(i)
        xnew *= ki
        ynew *= ki
    return (xnew, ynew, z)

def CORDIC_Ki(i):
    """
    Calculate the scaling factor for i-th iteration
    """
    k = 1.0/m.sqrt(1.0+2.0**(-2*i))
    return k

def CORDIC_Kn(n):
    """
    Calculate the overall scaling factor for n iterations
    """
    k = 1
    for i in range(0,n):
        k *= CORDIC_Ki(i)
    return k

def print_iteration(i, x, y, z):
    """
    Print intermediate results
    """
    res = '{0: <3}'.format(str(i))
    res += "{:12.8f} {:12.8f} {:12.8f}".format(x,y,z)
    print(res)

def prerotation(x, y, z):
    if((z >= m.pi / 2) and (z < 3 * m.pi / 2)):
        return -x, -y, (z - m.pi)
    elif((z >= 3 * m.pi / 2) and (z < 2 * m.pi)):
        return y, -x, (z - 3 * m.pi / 2)
    else:
        return x, y, z
```

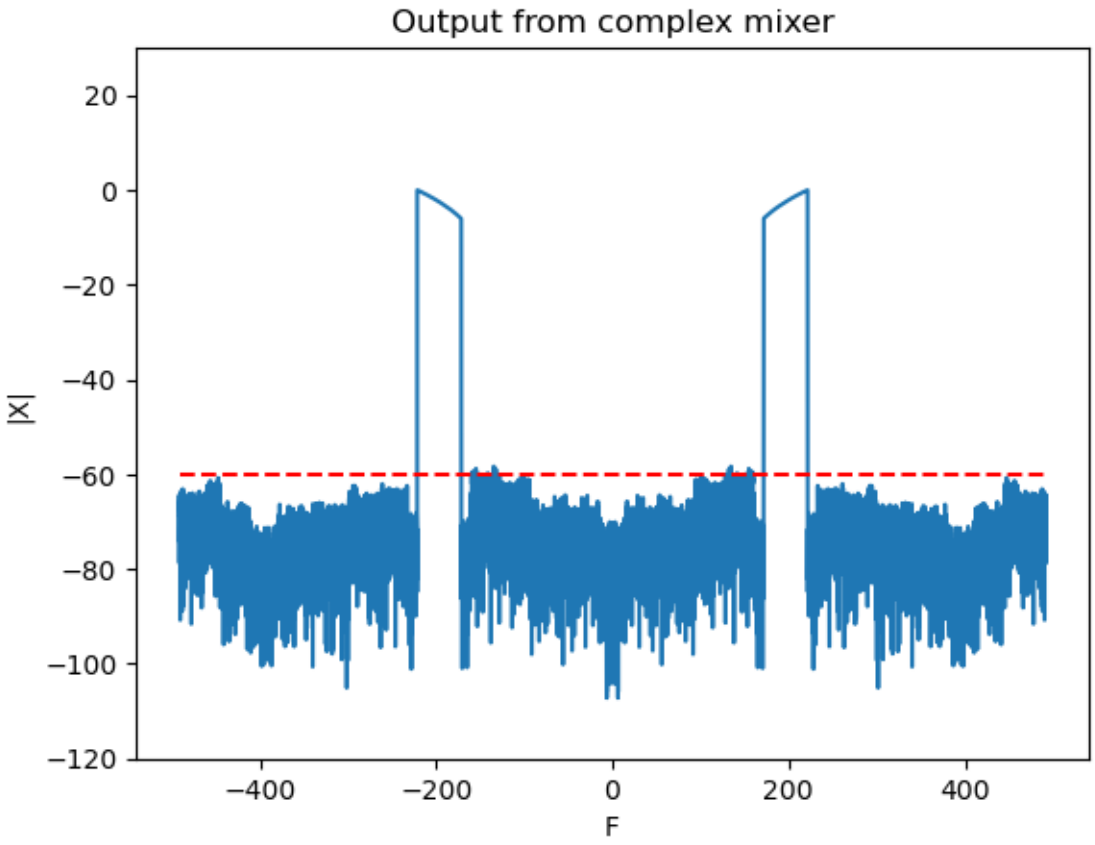
[39]:
```
def CORDIC_mixer(x4, fs_new , f0):
    Nx    = len(x4.real)
    W     = 2 * m.pi * f0 / fs_new

    n      = 10
    angle  = 0
    scaled = True
    x_out  = []
    y_out  = []
```

```
    for i in range(Nx):
        if(i != 0):
            angle += W

        if(angle >= 2 * m.pi):
            angle -= 2 * m.pi

        x = x4.real[i]
        y = x4.imag[i]

        x, y, z = prerotation(x, y, angle)

        for j in range(0,n):
            x, y, z = CORDIC_iteration(x, y, z, j, "rotation", scaled)

        x_out.append(x)
        y_out.append(y)

    return x_out, y_out
```

[40]:
```
Fpass = [0.203, 0.117, 0.051, 0.025]
Adb = 60
testsignal = readSamples("testsignal.txt")
I = 2

xFiltered = multistagePolyphase(Fpass, Adb, testsignal, 2)

Ny = len(xFiltered)
fs_new = 16 * 61.44e6
f0 = Ny//5 * fs / N
print("Central frequency is " + str(f0 / 1e6) + " MHz")

x_out, y_out = CORDIC_mixer(xFiltered, fs_new, f0)

freqs = (np.arange(N*M)/(N*M) - 0.5) * M*fs/1e6
XrdB = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(x_out))))

# plt.plot(freqs, XrdB[::-1]);
# plt.xlabel("F"); plt.ylabel("|X|"); plt.title("Output from complex mixer")
# plt.plot([freqs[0], freqs[-1]], [-60, -60], 'r--');
# plt.ylim(-120,30);

# x_tmp = (-1)**(np.arange(len(x_out)))
# x_out = np.array(x_out) * x_tmp

# freqs = (np.arange(N*M)/(N*M) - 0.5) * M*fs/1e6
```

```
# XrdB = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(x_out))))

plt.plot(freqs, XrdB);
plt.xlabel("F"); plt.ylabel("|X|"); plt.title("Output from complex mixer")
plt.plot([freqs[0], freqs[-1]], [-60, -60], 'r--');
plt.ylim(-120,30);
```

```
Central frequency is 196.56 MHz
```

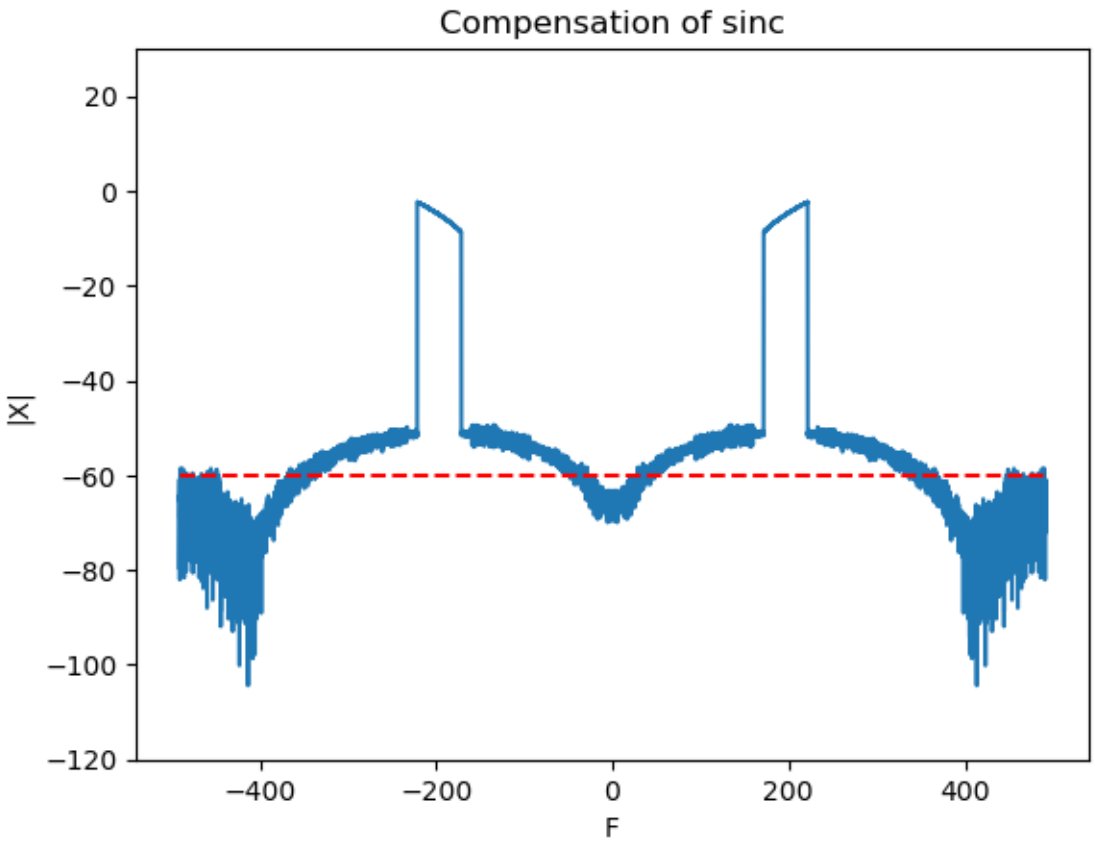


[41]:
```
from scipy.signal import *

def makeFIRsinc(fmax):
    n = 8 # number of points for approximation
    f = fmax * np.arange(0, n+1, 1)/n
    H_sinc = np.sin(np.pi*f)/(np.pi*f+1e-15)
    H_sinc[0] = 1.0
    f_goal = np.zeros(2*(n+1))
    f_goal[0::2] = f
    f_goal[1::2] = f+1e-3
    f_goal *= 2         # In firls function Nyquist frequency is 1
```

```
    goal = np.zeros(2*(n+1))
    goal[0::2] = 1.0/H_sinc
    goal[1::2] = 1.0/H_sinc
    h_invsinc = firls(9, f_goal, goal)
    return h_invsinc

def sincCompensation(x):
    h_invsinc = [ 0.00170522, -0.00583712,  0.01786389, -0.06833815,  0.
 ↪81251125, -0.06833815,0.01786389, -0.00583712,  0.00170522]
#     h_invsinc = makeFIRsinc(fmax = 0.4)
    y = np.convolve(x, h_invsinc)
    return y[len(h_invsinc) - 1:]
```

In order to compensate sinc function, in DAC, output signal should be filtered with FIR filter for sinc compensation. It can be seen that filtering the signal will degrade attenuation in stopband.

[42]:
```
Fpass = [0.203, 0.117, 0.051, 0.025]
Adb = 60
testsignal = readSamples("testsignal.txt")
I = 2

Ny = len(xFiltered)
fs_new = 16 * 61.44e6
f0 = Ny//5 * fs / N

xFiltered = multistagePolyphase(Fpass, Adb, testsignal, 2)
x_out, y_out = CORDIC_mixer(xFiltered, fs_new, f0)
y = sincCompensation(x_out)

freqs = (np.arange(N*M)/(N*M) - 0.5) * M*fs/1e6
XrdB = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(y))))

plt.plot(freqs, XrdB);
plt.xlabel("F"); plt.ylabel("|X|"); plt.title("Compensation of sinc")
plt.plot([freqs[0], freqs[-1]], [-60, -60], 'r--');
plt.ylim(-120,30);
```



IV

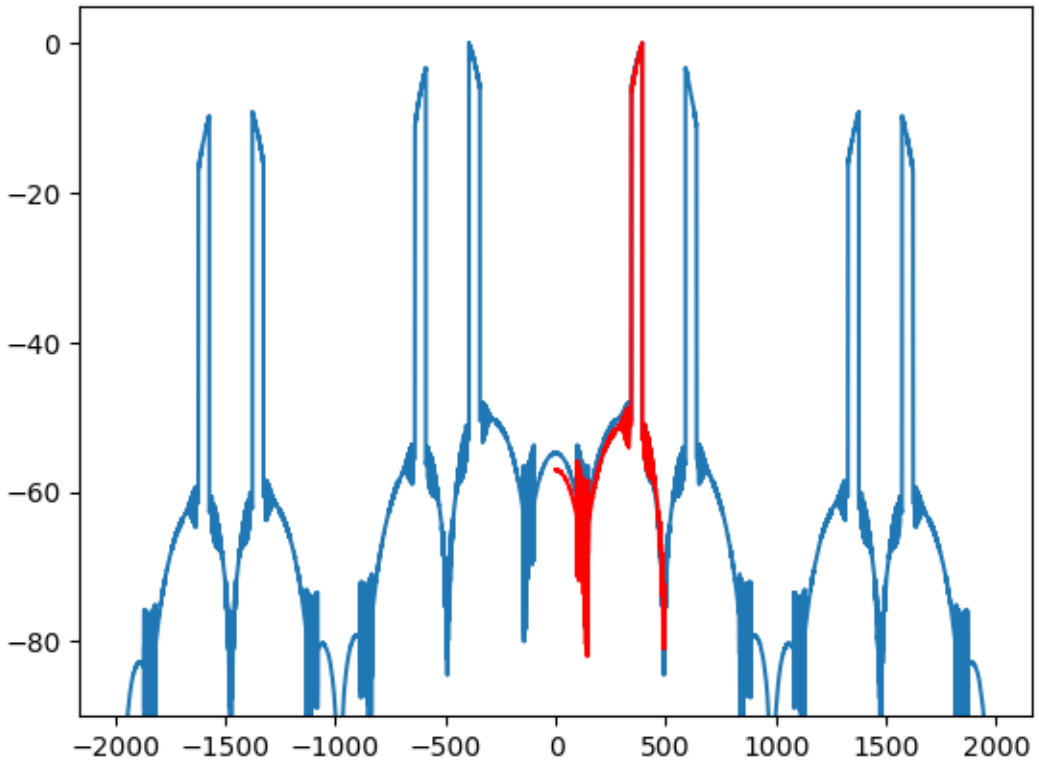
[49]:
```
Fpass = [0.203, 0.117, 0.051, 0.025]
Adb = 60
testsignal = readSamples("testsignal.txt")
I = 2
Ny = len(xFiltered)
fs_new = 16 * 61.44e6
f0_nrz = 368.64e6
f0_rf = 122.88e6

xFiltered = multistagePolyphase(Fpass, Adb, testsignal, 2)
xFiltered_rf = multistagePolyphase(Fpass, Adb, testsignal, 2)
x_out_nrz, y_out_nrz = CORDIC_mixer(xFiltered, fs_new, f0_nrz)
x_out_rf, y_out_rf = CORDIC_mixer(xFiltered_rf, fs_new, f0_rf)
y_nrz = sincCompensation(x_out_nrz)

# Inversion of spectrum
x_tmp = (-1)**np.arange(len(x_out_rf))
x_out_rf = np.array(x_out_rf) * x_tmp
```

```
freqs = (np.arange(N*M)/(N*M) - 0.5) * M*fs/1e6
YrdB_nrz = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(y_nrz))))
XrdB_nrz = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(x_out_nrz))))
XrdB_rf = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(x_out_rf))))
```

Zero order hold filter reconstructs signal in first Nyquist zone. In that way maximum signal frequency that can be reconstructed is $\frac{f_s}{2}$. Compensation for sinc is done here.

Red signal in an image shows which portion of spectre will be reconstructed using analog filter.

[50]:
```
# NRZ

M = 16
N = len(testsignal)
NyqZones = 4
nrz = np.ones(NyqZones)

xr_upsampled = np.zeros((len(y_nrz)-1)*NyqZones+1)
xr_upsampled[::NyqZones] = y_nrz

xr_nrz = np.convolve(nrz, xr_upsampled)
XrnrzdB = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(xr_nrz))))
freqs_nrz = (np.arange(N*M*len(nrz))/(N*M*len(nrz)) - 0.5) * M*fs*len(nrz)/1e6

plt.plot(freqs_nrz, XrnrzdB-np.max(XrnrzdB))
plt.plot(freqs[len(YrdB_nrz)//2:], YrdB_nrz[len(YrdB_nrz)//2:]-np.
 ↪max(YrdB_nrz),'r')
plt.ylim(-90,5);
```



Zero order hold filter with return to zero is used in reconstruction in second Nyqvist zone. In this way, maximum frequency of signal that can be reconstructed is f_s.

Spectre is inverted with respect to the x-axis. To fix this we should multiply digital signal by $x[n] = (-1)^n$.

Red signal in an image shows which portion of spectre will be reconstructed using analog filter.

[51]:
```
# RF

# x_tmp = (-1)**np.arange(len(x_out))
# x_out_inverted = np.array(x_out) * x_tmp

NyqZones = 4
rf = np.concatenate([np.ones(int(NyqZones/2)), -1.0*np.ones(int(NyqZones/2))])

xr_upsampled = np.zeros((len(x_out_rf)-1)*NyqZones+1)
xr_upsampled[::NyqZones] = x_out_rf

xr_rf = np.convolve(rf, xr_upsampled)
XrrfdB = 20*np.log10(np.fft.fftshift(abs(np.fft.fft(xr_rf))))
freqs_rf = (np.arange(N*M*len(rf))/(N*M*len(rf)) - 0.5) * M*fs*len(rf)/1e6
```

```
plt.plot(freqs_rf, XrrfdB[::-1]-np.max(XrrfdB))
plt.plot(freqs[:len(XrdB_rf)//2]+M*fs/1e6, XrdB_rf[:len(XrdB_rf)//2]-np.
 ↪max(XrdB_rf),'r')
plt.ylim(-80,0);
```