AI-assisted Design and Synthesis of Continuous-Time ADCs

Internal Report#4
AIR-CHIP Project (PID2022-138078OB-I00)

Author
Francisco Gómez Pulido

Distribution List
Gustavo Liñán-Cembrano
Jose M. de la Rosa

Date: February 19, 2025

Contents

1 Introduction and comments about this report

Date: Febrary 19, 2025

The goal of this report is to work with and familiarize oneself with the repository of Hampus Malberg. For this purpose, the report is structured as follows:

- **Section 2.1**: Installation of all necessary libraries and dependencies for using the repository.
- **Section 2.2**: A brief explanation of the main functions to be used.
- **Section 2.3**: Following Hampus' `tutorial.ipynb`, with explanations that clarify the code and the results obtained.
- **Section 2.4**: Study of the effect on $\Sigma\Delta$Ms when varying different parameters.
- **Sections 2.5**: Solving a small optimization problem, which involves obtaining the best parameter configuration based on certain specifications.
- **Section 2.6**: Focus on investigating which implementation (Active-RC or Gm-C) would be the best once certain parameters are obtained.
- **Section 3**: Conclusions and recommendations.

This report is submitted along with the notebooks used for the study. I recommend looking at the notebooks only, as this report may be quite heavy, since it shows step-by-step how I arrived at the proposed solution, displaying almost all the code and outputs received. If the reader is interested in understanding all the details of why each decision was made, then it is worth reading the report while looking at the corresponding notebooks, as both follow the same structure. At the end, I strongly recommend to read section 3.

Finally, as a comment, it is worth noting that the methodology and algorithms proposed in this report are not efficient at all, as the problems are solved by brute-force search, and of course, there are thousands of better ways to solve them. This is just a small approximation, more for familiarizing with the environment than for correctly solving the problems.

2 Hampus' repository

Date: Febrary 13, 2025

2.1 Installation of the environment

This tutorial is designed to be executed on **Windows**. If you are using Linux, the installation process may be a bit different, but the general steps remain quite similar.

2.1.1 Cloning the repository

To begin, we need to clone the repository that contains the necessary files for this project. Open a terminal or powershell and run the following command:

```
git clone https://github.com/hammal/IMSE2025.git
```

This command will create a local copy of the repository on your computer, allowing you to access all the project files.

2.1.2 Setting up the environment

For executing the code, I will use **JupyterLab**. I already had my Python environment set up, including Anaconda and other necessary tools. However, if you do not have them installed, setting them up is straightforward and does not take too much time.

Before running the project, we need to create a new **virtual environment** (venv). This step ensures that all dependencies are managed separately, preventing conflicts with other Python projects. Open a terminal with administrative privileges and execute:

```
python -m venv imse2025
```

Once this command is executed, a new folder named `imse2025` will appear in your working directory. This folder contains an isolated Python environment where we will install all required dependencies.

2.1.3 Activating the virtual environment

After creating the virtual environment, we need to activate it. Run the following command:

```
imse2025\Scripts\activate
```

This command executes a script inside the newly created virtual environment, enabling it for use. However, in some cases, you may encounter an error related to script execution policies. If this happens, you need to allow the execution of scripts temporarily by running:

```
Set-ExecutionPolicy Unrestricted -Scope Process
```

This command allows scripts to run in the current powershell session. After completing the setup, you can revert to the default policy (although this is usually done automatically I think) by executing:

```
Set-ExecutionPolicy Restricted -Scope Process
```

2.1.4 Installing dependencies

Once the virtual environment is activated, we need to install the required dependencies. To ensure we have the latest version of essential packages, run:

```
python -m pip install --upgrade pip ipykernel jupyterlab
```

Next, install a Jupyter kernel for our virtual environment so that it appears as an option inside JupyterLab:

```
python -m ipykernel install --user --name=imse2025 --display-name "IMSE 2025"
```

2.1.5 Installing CBADC package

Now, we need to install the **CBADC package**, which is essential for this project. While inside the virtual environment, we first check the available branches in the repository by running:

```
git ls-remote https://github.com/hammal/cbadc.git
```

This command lists all the available versions of the package. Make sure `feature/0.4.0` appears in the list. We then proceed with the installation by specifying the correct branch or version:

```
python -m pip install git+https://github.com/hammal/cbadc.git@feature/0.4.0
```

This process might takes a few minutes to complete.

2.1.6 Installing QR code generator (optional)

As an optional step, you can install a QR code generator package, which might be useful for certain applications. To do so, execute:

```
pip install --upgrade qrcode Pillow
```

2.1.7 Launching JupyterLab

After completing all installations, we are ready to start **JupyterLab**. To launch it, simply run:

```
jupyter lab
```

This command will open a new browser window with the JupyterLab interface. It is important to **keep the terminal open** while working with JupyterLab, as closing it will terminate the session. All modifications to your notebooks and scripts will be saved locally in the selected folder.

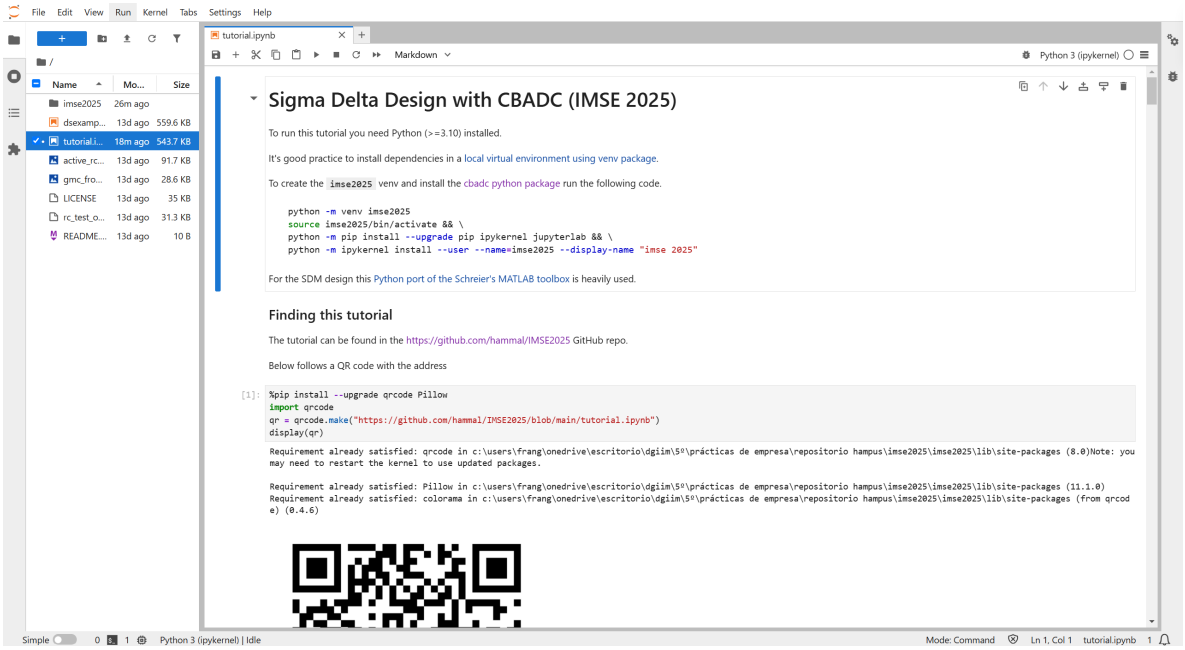


Figure 1: JupyterLab window.

2.2 Understanding cbadc's functions

Before starting the tutorial, it's beneficial to explore the main functions provided by the **cbadc library**. These functions are located in `cbadc/src/cbadc/delsig.py` and are essential for designing and analyzing ΣΔMs. Below is a list of key functions along with their descriptions:

- `synthesizeNTF`:
  - Purpose: Designs a Noise Transfer Function (NTF) for a ΣΔM based on specified order and out-of-band gain.
  - Usage: This function is used to create an NTF that shapes the quantization noise, pushing it out of the band of interest to improve signal quality within the desired frequency range.
- `realizeNTF`:
  - Purpose: Converts a given NTF into state-space matrices (ABCD) representing the modulator's loop filter.
  - Usage: This is useful for implementing the NTF in a practical modulator design by providing the necessary parameters for simulation and hardware realization.
- `stuffABCD`:
  - Purpose: Computes the ABCD state-space representation from modulator topology parameters.
  - Usage: Facilitates the analysis and implementation of different modulator structures by providing a standardized representation.
- `mapABCD`:
  - Purpose: Determines the coefficients for a specified modulator topology.
  - Usage: Assists in mapping the theoretical design to practical coefficients that can be used in implementation.
- `partitionABCD`:
  - Purpose: Divides the ABCD matrix into its constituent state-space matrices A, B, C, and D.
  - Usage: Useful for detailed analysis and understanding of the modulator's internal dynamics.
- `scaleABCD`:
  - Purpose: Scales the loop filter of a ΣΔM to optimize its dynamic range.
  - Usage: Ensures that the modulator operates efficiently within its intended signal range, preventing overload and distortion.
- `simulateDSM`:
  - Purpose: Simulates the behavior of a ΣΔM given an input signal and its ABCD representation.
  - Usage: Allows for the evaluation of modulator performance through time-domain simulations, aiding in design verification.
- `realizeNTF_ct`:
  - Purpose: Realizes a continuous-time loop filter that implements a specified NTF.

  - Usage: Essential for designing CT-ΣΔMs, providing the necessary parameters for implementation.

- `calculateTF`:
  - Purpose: Calculates the Noise Transfer Function (NTF) and Signal Transfer Function (STF) of a ΣΔM.
  - Usage: Provides insight into how the modulator processes signals and noise, which is crucial for performance analysis.

- `simulateSNR`:
  - Purpose: Determines the Signal-to-Noise Ratio (SNR) of a ΣΔM through simulation.
  - Usage: Helps in assessing the modulator's performance by quantifying the quality of the output signal relative to noise.

- `plotPZ`:
  - Purpose: Plots the poles and zeros of a transfer function.
  - Usage: Visualizes the frequency response characteristics of the modulator, aiding in stability and performance analysis.

2.3 Tutorial.ipynb

2.3.1 Imports needed

```
import numpy as np
import cbadc as cb
import matplotlib.pyplot as plt
```

2.3.2 Design example for a LP CT-ΣΔM

First, we need to define the model parameters in Python:

```
order = 3 # order of the SDM
osr = 32 # oversampling ratio
nlev = 2 # number of levels of the DAC
f0 = 0. # input frequency (0 means that the input signal is a baseband signal)
Hinf = 1.5 # feedback filter gain (key role in controlling the pole-zero placement of NTF)
tdac = [0, 1] # DAC sampling times (two sampling instants for the feedback structure)
# form of the SDM in DT:
form = 'FB' # (feedback: the modulator includes a feedback filter)
dt_form = 'CRFB' # feedback structure
Bw = 10e6 # bandwidth
fs = Bw * osr * 2 # sampling frequency
print(f"fs = {fs/1e6} MHz")
```

```
fs = 640.0 MHz
```

We now synthesize the DT-ΣΔM and extract zeros and poles of its NTF:

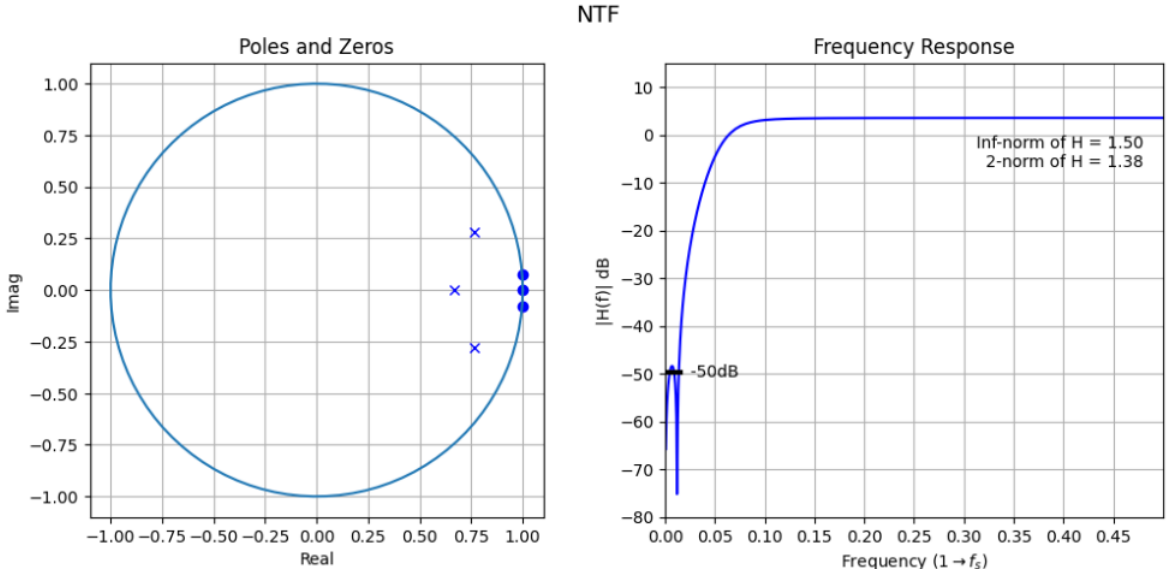
```
ntf0 = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0) # 2: Optimized zero placement

# extract zeros and poles from ntf0:
zeros = ntf0.zeros
poles = ntf0.poles

for z, p in zip(ntf0.zeros, ntf0.poles):
    print("(%f, %fj)\t(%f, %fj)" % (np.real(z), np.imag(z), np.real(p), np.imag(p)))
```

```
(1.000000, 0.000000j)        (0.764515, -0.280052j)
(0.997110, 0.075973j)        (0.764515, 0.280052j)
(0.997110, -0.075973j)        (0.668460, 0.000000j)
```

This output shows the extracted zeros and poles as complex numbers. These values are important for understanding the frequency response and noise shaping characteristics of the modulator.

This code will generate a plot showing the Noise Transfer Function (NTF):

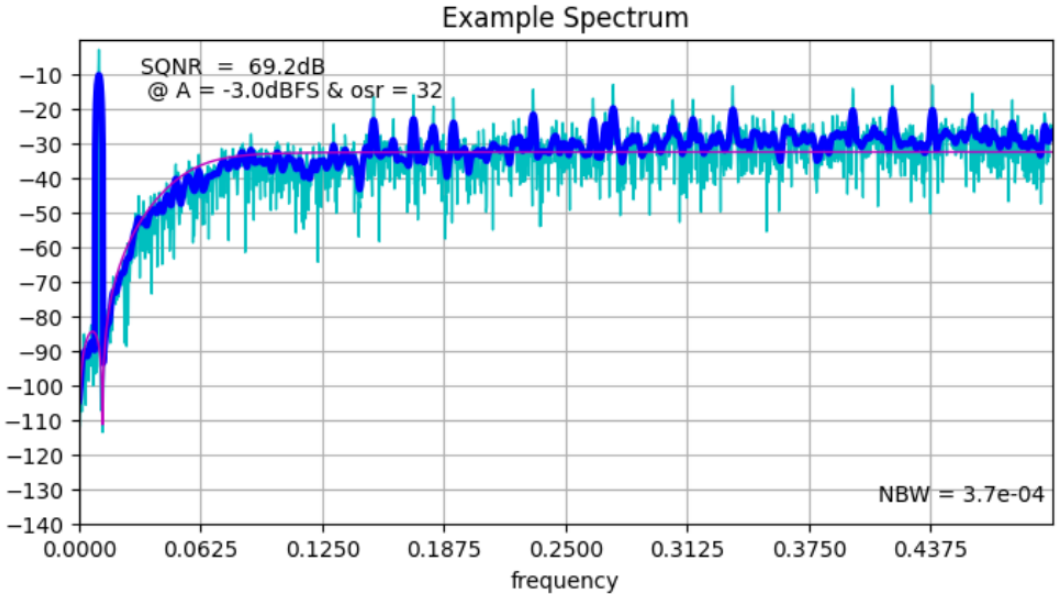
```
# plot zeros and poles:
# cb.delsig.plotPZ(ntf0, showlist=False)
# cb.delsig.changeFig(10, 1.5, 7)
cb.delsig.DocumentNTF(ntf0, osr, f0, False)
```



The plot is important for visualizing the noise shaping and frequency response of the modulator.

Left plot displays the **pole-zero diagram** of the NTF. The blue circle represents the unit circle in the complex plane. Blue crosses indicate locations of the zeros and blue dots represent the poles. The zeros are clustered near $z = 1$, which means that the NTF attenuates low-frequency noise. The poles are inside the unit circle, ensuring system stability. Right plot shows the magnitude response of the NTF in dB as a function of frequency. The NTF exhibits strong attenuation at low frequencies and high gain at high frequencies. This behavior ensures that quantization noise is pushed to higher frequencies, where it can be filtered out in a ΣΔM.

```
cb.delsig.PlotExampleSpectrum(ntf0, nlev-1, osr, f0)
plt.title('Example Spectrum');
```



The result would be an example spectrum plot that shows **how the modulator affects the input signal in terms of noise shaping**.

All this process synthesizes and visualizes the characteristics of a 3rd-order ΣΔM, showing how the poles and zeros of the NTF are placed and how the noise shaping affects the system's frequency response. The example spectrum helps visualize the impact of the modulation on the signal, giving insight into the performance of the modulator.

Hampus' library starts when there is an actual set of differential o difference equations to relate to. So, once we have the NTF synthesized, the next step is to realize the modulator's components. We will choose an architecture for the modulator and convert the NTF into a set of differential equations for the continuous-time system.

```
# decide an architecture: CRFB (Cascade Feedback Form)
a, g, b, c = cb.delsig.realizeNTF(ntf0, form="CRFB") # realizes de NTF for a given architecture
ABCD = cb.delsig.stuffABCD(a, g, b, c) # after realizing the NTF, the parameters a, g, b, c are
    # passed to stuffABCD, which computes the system matrices for the modulator. These matrixs
    # (A, B, C, D) represent the state-space formulation of the CT system

dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev) # this converts the CT system into DT
    # model for the modulator's frontend
print(dt_analog_frontend)
```

```
(AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.    0.    0.  ]
 [ 1.    1.   -0.01]
 [ 1.    1.    0.99]],
B=
[[ 0.04 -0.04]
 [ 0.24 -0.24]
 [ 0.8  -0.8 ]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
```

The output shows the state-space matrixs (A, B, C, D) of the modulator's analog filter, which represents the behavior of the continuous-time system. It also shows the digital control and analog signal parameters, which are used in the overall system model.

Next, we synthesize a continuous-time loop filter. Note that we rescale the coefficients with the following line of code:

```
ct_analog_frontend.dt = 1.0/fs
```

This rescaling ensures the time steps align with the sampling frequency (`fs`) of the system, though it doesn't affect the simulation results of the CT-ΣΔM.

```
# transform the DT NTF into its CT equivalent:
ABCDc, tdac2 = cb.delsig.realizeNTF_ct(ntf0, form, tdac) # realizes the CT version of the NTF
    # it outputs the ABCD matrixs and the DAC waveform
print(f"ABCDc = \n{ABCDc}")
print(f"tdac2 = {tdac2}")

ct_analog_frontend = cb.AnalogFrontend.ctsdm(ABCDc, tdac2, quantization_levels=nlev)
    # constructs a CT SDM frontend based on the computed ABCD matrixs and the DAC waveform
ct_analog_frontend.dt = 1.0/fs # rescale the system time (dt) to match fs
print(ct_analog_frontend)
```

```
ABCDc =
[[ 0.    0.    0.    0.04 -0.04]
 [ 1.    0.   -0.01  0.   -0.24]
 [ 0.    1.    0.    0.   -0.67]
 [ 0.    0.    1.    0.    0.  ]]
tdac2 = [[-1. -1.]
 [ 0.  1.]]
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 0.0e+00  0.0e+00  0.0e+00]
 [ 6.4e+08  0.0e+00 -3.7e+06]
 [ 0.0e+00  6.4e+08  0.0e+00]],
B=
[[ 2.84e+07 -2.84e+07]
 [ 0.00e+00 -1.54e+08]
 [ 0.00e+00 -4.29e+08]],
C=
[[0. 0. 1.]],
D=
[[0. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.5625e-09, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.5625e-09
)
```

This output shows the state-space matrices for the analog filter in the continuous-time domain, along with the digital control and analog signal parameters.

As a sanity check, to verify the behavior of the CT system, we can discretize the CT system and compare it with the original DT modulator. This is a standard method in Schreirer's toolbox to ensure CT implementation behaves as expected.

```
# discretizing the CT system:
dt_ct_analog_frontend = ct_analog_frontend.discretize(dt=ct_analog_frontend.dt) # converts the
```

```
    # CT system into a DT system with the appropiate step dt

ABCD = dt_ct_analog_frontend.ABCD # after discretizing the system, we extract the ABCD matrixs
    # for the discretized system
print(dt_ct_analog_frontend)
```

```
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.    0.    0.  ]
 [ 1.    1.   -0.01]
 [ 0.5   1.    1.  ]],
B=
[[ 0.04 -0.04]
 [ 0.02 -0.26]
 [ 0.01 -0.8 ]],
C=
[[0. 0. 1.]],
D=
[[0. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
```

The printed `dt_ct_analog_frontend` object reveals the internal structure of the discretized system:

- A matrix defines the state evolution, how the states evolve at discrete time steps.
- B matrix represents input coupling.
- C matrix selects the output state.
- D matrix remains zero, indicating there is no direct feedthrough from input to output.

The discretized system is now in the form suitable for simulation in the digital domain.

Finally, we compare the discretized continuous-time system with the original discrete-time system to check if they match:

```
print(dt_analog_frontend)
```

```
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.    0.    0.  ]
 [ 1.    1.   -0.01]
 [ 1.    1.    0.99]],
B=
```

```
[[ 0.04 -0.04]
 [ 0.24 -0.24]
 [ 0.8  -0.8 ]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
```

The systems are similar but not identical. These deviations might cause slight variations in performance, but they should still produce comparable overall behavior. So this comparison confirms that the continuous-time system was successfully discretized and is now ready for digital simulation.

2.3.3 Simulation

Instead of comparing poles, zeros or impulse responses, we go straight for simulations to **evaluate the performance** of the discrete-time (DT), continuous-time (CT), and the discretized continuous-time (DT-CT) system. These simulations have these conditions:

- 25 input amplitudes simulated.
- The simulation length is 2^{13}.
- This results in 4M output data points (or 1.3M if you consider the states).

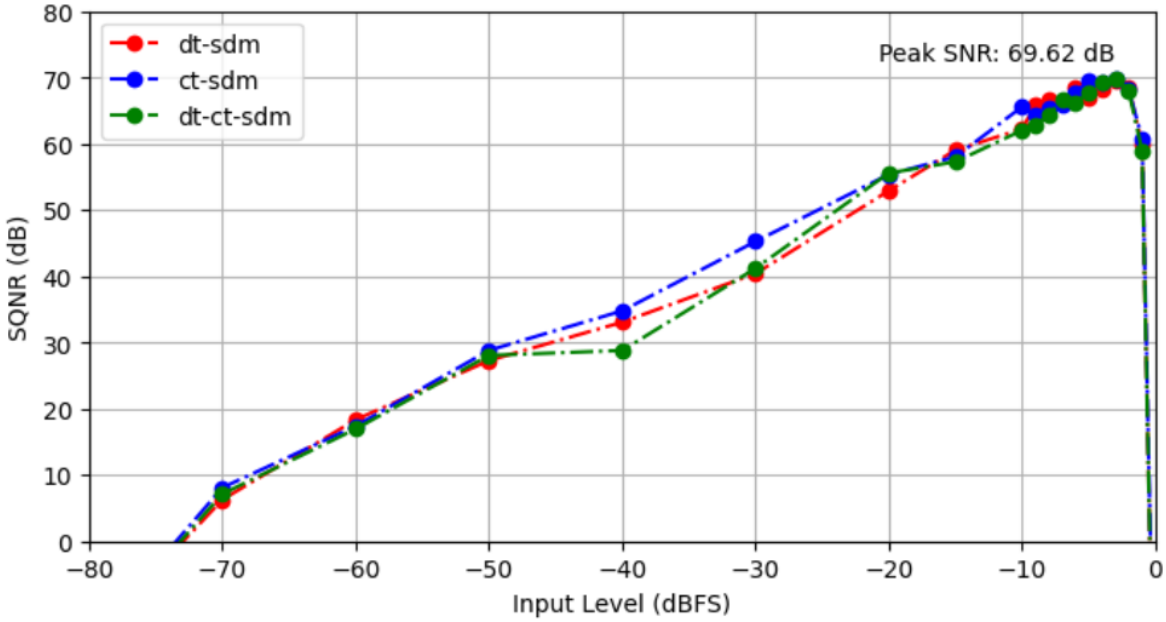
```
snr_dt, amp_dt, _ = dt_analog_frontend.simulateSNR(osr)
```

```
snr_ct, amp_ct, _ = ct_analog_frontend.simulateSNR(osr)
```

```
snr_dt_ct, amp_dt_ct, _ = dt_ct_analog_frontend.simulateSNR(osr)
```

The `simulateSNR` function is used to **simulate the Signal-to-Noise Ratio (SNR)** for each of the three systems (DT, CT, DT-CT). `snr_dt`, `snr_ct` and `snr_dt_ct` hold the SNR values for each simulation, while `amp_dt`, `amp_ct` and `amp_dt_ct` represent the corresponding input amplitudes.

```
plt.plot(amp_dt, snr_dt, 'o-.r', label='dt-sdm')
plt.plot(amp_ct, snr_ct, 'o-.b', label='ct-sdm')
plt.plot(amp_dt_ct, snr_dt_ct, 'o-.g', label='dt-ct-sdm')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.grid()

peak_amp_dt = np.argmax(snr_dt)
plt.text(amp_dt[peak_amp_dt], snr_dt[peak_amp_dt] + 3,
    f'Peak SNR: {snr_dt[peak_amp_dt]:.2f} dB', horizontalalignment='right')
plt.xlim((-80, 0))
plt.ylim((0, 80))
plt.legend()
```

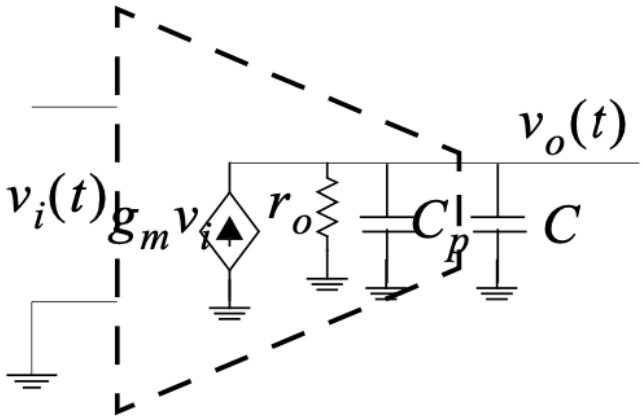


The plot compares the SQNR (Signal-to-Quantization Noise Ratio) performance of the three systems for different input amplitudes (dBFS). The peak SNR is identified and shown on the plot.

SQNR increases as the input level rises from -73 dBFS to approximately -4 dBFS. After this point, SQNR reaches its peak (69.62 dB, as indicated in the annotation) and then drops sharply. Across most of the input range, the three approaches exhibit very similar behavior.

This final step in the study provides empirical validation of the CT-ΣΔM's design by ensuring its behavior aligns with DT implementations.

2.3.4 Gm-C integrators



Gm-C integrator is defined in Python as a class (in `cbadc/analog_frontend.py`):

```
1 class GmC(AnalogFrontend):
2     """Transconductance-Capacitance Analog Frontend
```

```
    Parameters
    ----------
    analog_frontend: : :py:class:`cbadc.analog_frontend.AnalogFrontend`
        the analog frontend
    Cint: : :py:class:`numpy.ndarray`, shape=(N,)
        the larger integration capacitor
    Ro: : :py:class:`numpy.ndarray`, shape=(N,)
        the output resistance, defaults to np.zeros(N)
    Cp: : :py:class:`numpy.ndarray`, shape=(N,)
        the parasitic capacitance, defaults to np.zeros(N)
    v_n: : :py:class:`numpy.ndarray`, shape=(N,)
        the input referred noise density in V rms, defaults to np.zeros(N)
    v_out_min: : :py:class:`numpy.ndarray`, shape=(N,)
        the minimum output voltage, defalts to -np.inf * np.ones(N)
    v_out_max: : :py:class:`numpy.ndarray`, shape=(N,)
        the maximum output voltage, defaults to np.inf * np.ones(N)
    slew_rate: : :py:class:`numpy.ndarray`, shape=(N,)
        the slew rate, defaults to np.inf * np.ones(N)
    """
```

First, we need to define the component parameters:

```
Ro = np.ones(order) * 1e5 # output resistance, set to 100kOms for all integrators
Cp = np.ones(order) * 1e-15 # parasitic capacitance, set to 1 fF
Cint = np.ones(order) * 1e-12 # integrating capacitance, set to 1 pF
slew_rate = 1e9 * np.ones(order) # maximum rate of change of output voltage, set to 1e9 V/s
max_output_swing = 1.0 * np.ones(order) # maximum voltage swing of the output, set to +1V
min_output_swing = -max_output_swing # minimum voltage swing of the output, set to -1V
v_noise_rms = 1e-3 * np.ones(order) # RMS noise voltage, set to 1 mV
```

Then, we create the **Gm-C integrator model**. The constructor takes the analog frontend object, along with component values (capacitance, resistance, noise...). `simulateSNR(osr)` runs a Signal-to-Noise Ratio (SNR) simulation for a given oversampling ratio.

```
GmC = cb.GmC(
    ct_analog_frontend,
    Cint,
    Ro,
    Cp,
    v_n=v_noise_rms,
    slew_rate=slew_rate,
    v_out_max=max_output_swing,
    v_out_min=min_output_swing
)

snr_gmc, amp_gmc, _ = GmC.simulateSNR(osr)
print(GmC)
```

```
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[-9.99e+06  0.00e+00  0.00e+00]
 [ 6.40e+08 -9.99e+06 -3.70e+06]
 [ 0.00e+00  6.40e+08 -9.99e+06]],
B=
[[ 2.84e+07 -2.84e+07]
```

```
 [ 0.00e+00 -1.54e+08]
 [ 0.00e+00 -4.29e+08]],
C=
[[0. 0. 1.]],
D=
[[0. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.5625e-09, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]],
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.5625e-09
)
```

This is the state-space representation of the analog filter and its connection to the digital control and signal parameters. A, B, C, D are matrixs that define the continuous-time system dynamics. `digital_control` specifies the discrete-time properties, such as sampling period `dt` and DAC waveform. `analog_signal` defines properties such as signal offset and whether it's piecewise constant.

```
print(f"The GmC:\n{GmC.gm}")
```

```
The GmC:
[[ 0.  0.  0.  0. -0.]
 [ 0.  0. -0.  0. -0.]
 [ 0.  0.  0.  0. -0.]]
```

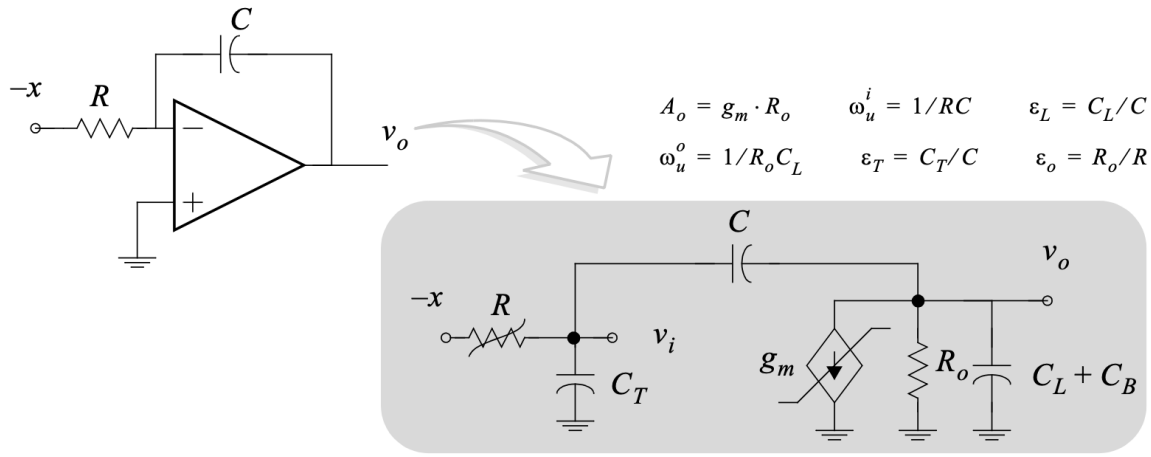
`GmC.gm` represents the **transconductance matrix** of the Gm-C integrator network. The values are mostly 0, which may indicate that transconductors have not been properly set up.

```
print(f"The DC_gain:\n{GmC.dc_gain}")
```

```
The DC_gain:
[[  0.     0.     0.     2.84  -2.84]
 [ 64.     0.    -0.37   0.   -15.36]
 [  0.    64.     0.     0.   -42.88]]
```

`GmC.dc_gain` represents the **DC gain matrix**, which describes how DC signals are amplified or attenuated through the integrators. The matrix structure indicates how different signals are coupled between various system states.

2.3.5 Active-RC integrators



Active-RC integrator is defined in Python as a class (in `cbadc/analog_frontend.py`):

```
class ActiveRC(AnalogFrontend):
    """Active RC Analog Frontend
    Parameters
    ----------
    analog_frontend: : :py:class:`cbadc.analog_frontend.AnalogFrontend`
        the analog frontend
    Cint: : :py:class:`numpy.ndarray`, shape=(N,)
        the integration capacitance.
    gm: : :py:class:`numpy.ndarray`, shape=(N,)
        the transconductance.
    Ro: : :py:class:`numpy.ndarray`, shape=(N,)
        the internal state resistance.
    Co: : :py:class:`numpy.ndarray`, shape=(N,)
        the internal state capacitance.
    """
```

First, we need to define the component parameters:

```
Ro = 1e6 * np.ones(order) # output resistance, set to 1 MOhm for all integrators
Co = np.ones(order) * 1e-15 # output capacitance, set to 1 fF
Cint = np.ones(order) * 1e-12 # integrating capacitance, set to 1 pF
gm = 1e-5 * np.ones(order) # transconductance, set to 10 micro-Siemens
```

Creating the **Active-RC integrator model** and simulating SNR:

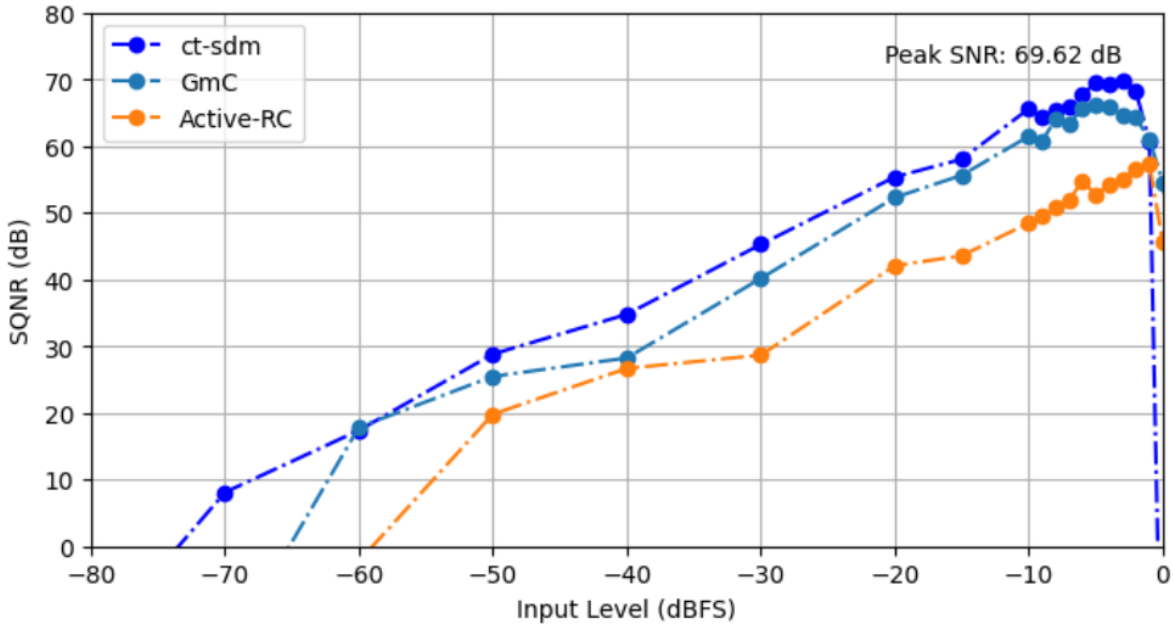
```
active_RC = cb.ActiveRC(
    ct_analog_frontend,
    Cint,
    gm,
    Ro,
    Co
)

snr_active_rc, amp_active_rc, _ = active_RC.simulateSNR(osr)
print(active_RC)
```

```
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[-1.00e+10  0.00e+00  0.00e+00 -1.00e+09 -0.00e+00 -0.00e+00]
 [ 0.00e+00 -1.06e+10  0.00e+00 -6.40e+08 -1.00e+09  3.70e+06]
 [ 0.00e+00  0.00e+00 -1.06e+10 -0.00e+00 -6.40e+08 -1.00e+09]
 [-1.00e+10 -0.00e+00 -0.00e+00 -1.00e+09 -0.00e+00 -0.00e+00]
 [-0.00e+00 -1.00e+10 -0.00e+00 -0.00e+00 -1.00e+09 -0.00e+00]
 [-0.00e+00 -0.00e+00 -1.00e+10 -0.00e+00 -0.00e+00 -1.00e+09]],
B=
[[ 2.84e+07 -2.84e+07]
 [ 0.00e+00 -1.54e+08]
 [ 0.00e+00 -4.29e+08]
 [ 0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00]
 [ 0.00e+00  0.00e+00]],
C=
[[ 0.  0.  0. -0. -0. -1.]],
D=
[[0. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.5625e-09, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=6,
L=1,
M=1,
dt=1.5625e-09
)
```

State-space representation, which defines the analog filter structure. The system has 6 states, meaning this is a higher-order filter than the previous Gm-C example.

2.3.6 Comparison

```python
# plt.plot(amp_dt, snr_dt, 'o-.r', label='dt-sdm')
plt.plot(amp_ct, snr_ct, 'o-.b', label='ct-sdm')
# plt.plot(amp_dt_ct, snr_dt_ct, 'o-.g', label='dt-ct-sdm')
plt.plot(amp_gmc, snr_gmc, 'o-.', label='GmC')
plt.plot(amp_active_rc, snr_active_rc, 'o-.', label='Active-RC')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.grid()

peak_amp_dt = np.argmax(snr_dt)
plt.text(amp_dt[peak_amp_dt], snr_dt[peak_amp_dt] + 3,
    f'Peak SNR: {snr_dt[peak_amp_dt]:.2f} dB', horizontalalignment='right')
plt.xlim((-80, 0))
plt.ylim((0, 80))
plt.legend()
```



This plot shows the SQNR (Signal-to-Quantization-Noise Ratio) as a function of the input level (dBFS, decibels relative to full scale) for three different architectures: CT-ΣΔM (ideal), Gm-C and Active-RC.

The SQNR indicates how well the signal is preserved compared to quantization noise. Higher values mean better performance. The SQNR increases with input level until a certain point, after which it starts to degrade. The text in the plot marks the point where CT-ΣΔM achieves its highest SQNR. This shows the maximum performance of the best-performing architecture. The Gm-C architecture performs better than the Active-RC architecture, but it doesn't reach the ideal performance.

2.3.7 Transient simulation

This section of the code performs transient simulations on three different architectures: CT-ΣΔM, Gm-C and Active-RC. The goal is to evaluate how these systems respond to a sinusoidal input signal and analyze their transient behavior.

```
size = 1 << 14 # simulation size, set to 2^14 samples

# define sinusoidal input signal:
amplitude = np.array([[1.0]], dtype=float) # 1.0 (full-scale signal)
freq = np.array([[GmC.fs / 128]], dtype=float) # sampling frequency divided by 128
sinusoidal = cb.Sinusoidal(amplitude, freq)
print(sinusoidal)

# asign the sinusoidal input to each architecture (so the three are simulated with the same
    # input conditions):
GmC.analog_signal = sinusoidal
ct_analog_frontend.analog_signal = sinusoidal
active_RC.analog_signal = sinusoidal
# ct_analog_frontend.A += - 1e-7 * np.eye(order)

# run the transient simulations:
ct_af_sim = ct_analog_frontend.simulate(size)
```

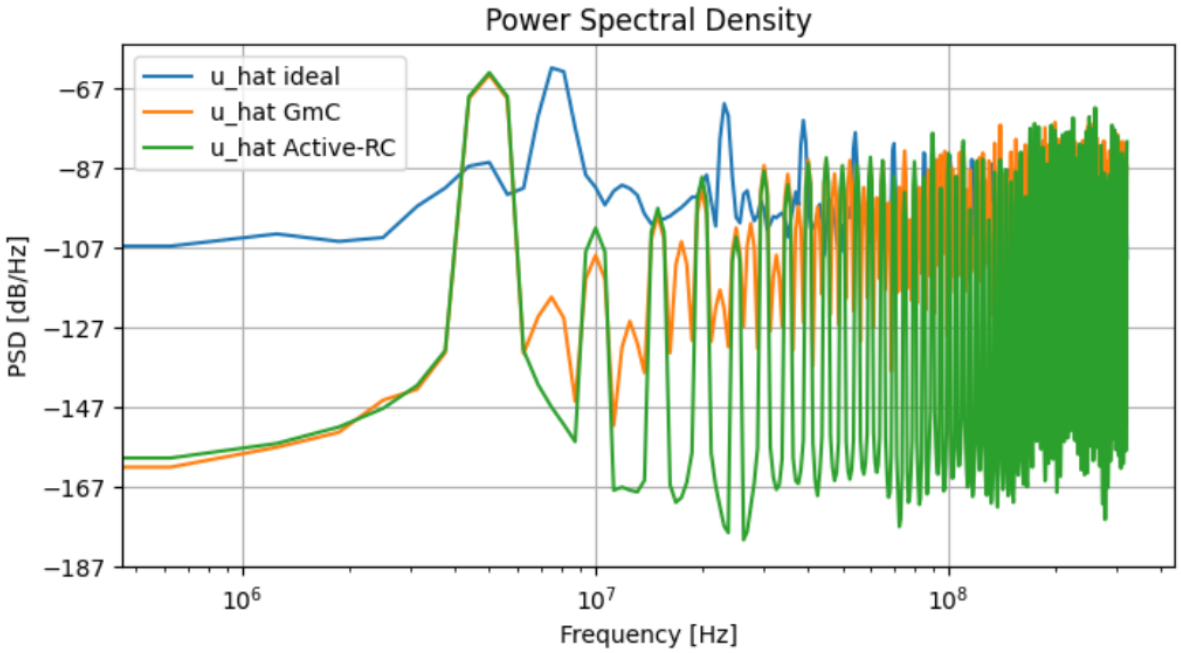
```
gmc_sim = GmC.simulate(size)
active_RC_sim = active_RC.simulate(size)
# each system processes the sinusoidal input signal, producing their respective transient
# outputs

# apply Wiener filtering for noise reduction:
ct_wiener_filter = ct_analog_frontend.wiener_filter(OSR=osr)
ct_u_hat = ct_wiener_filter.evaluate(ct_af_sim["s"])[:, 0, :]

gmc_wiener_filter = GmC.wiener_filter(OSR=osr)
gmc_u_hat = gmc_wiener_filter.evaluate(gmc_sim["s"])[:, 0, :]

active_rc_wiener_filter = active_RC.wiener_filter(OSR=osr)
active_rc_u_hat = active_rc_wiener_filter.evaluate(active_RC_sim["s"])[:, 0, :]
```

```
INFO:root:Simulating continuous-time analog frontend for sinusoidal input

Sinusoidal parameterized as:
amplitude = [[1.]],
frequency = [[5000000.]],
phase = [[0.]],
offset = [[0.]]
```

This confirms that the sinusoidal input signal has been set up correctly. The frequency is 5 MHz, assuming a sampling rate of 640 MHz, since $f_s/128 = 5$ MHz. No phase shift or DC offset is present in the input signal.

This simulation provides time-domain responses for different integrator architectures under the same sinusoidal input. By applying Wiener filtering, the reconstructed signals can be analyzed to compare the performance of each architecture in terms of distortion, noise rejection, and transient behavior.

```
plt.figure()
plt.psd(ct_af_sim["s"].flatten(), NFFT=1024, Fs=1 / ct_analog_frontend.dt, label="u_hat ideal")
plt.psd(gmc_sim["s"].flatten(), NFFT=1024, Fs=1 / GmC.dt, label="u_hat GmC")
plt.psd(active_RC_sim["s"].flatten(), NFFT=1024, Fs=1 / active_RC.dt, label="u_hat Active-RC")
plt.legend()
plt.xlabel("Frequency [Hz]")
plt.ylabel("PSD [dB/Hz]")
plt.title("Power Spectral Density")
plt.xscale("log") # logarithmic axis (helps visualize a wide frequency range more effectively)
```

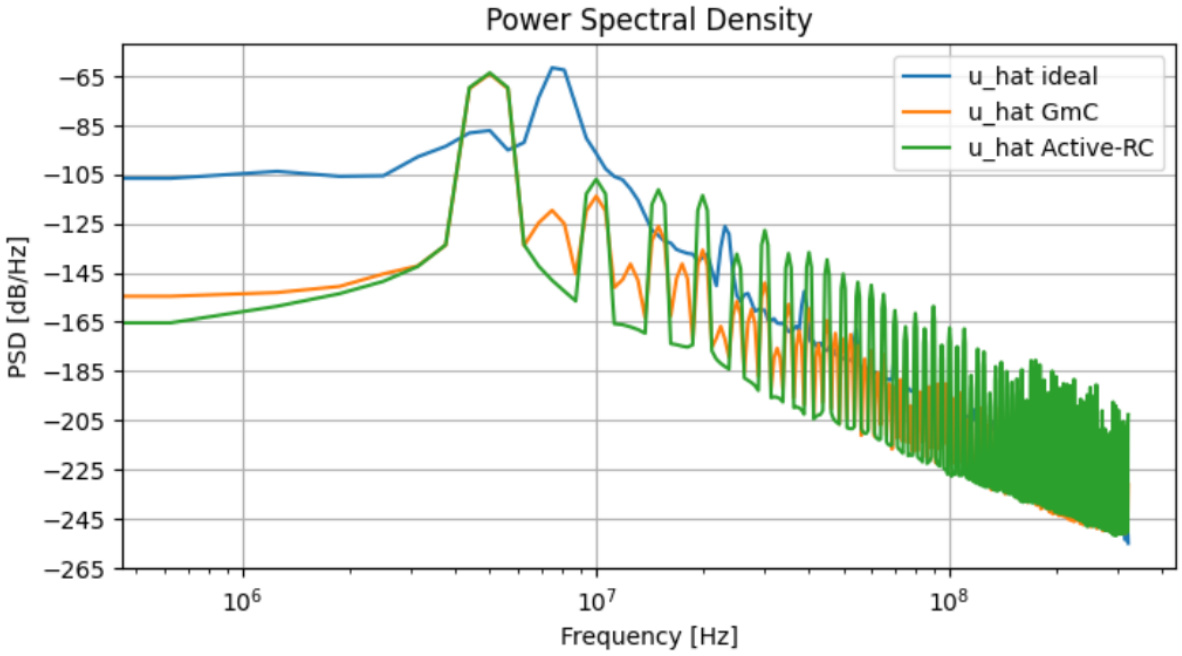


This plot shows the **Power Spectral Density (PSD)** of the reconstructed signals ($\hat{u}$) for Gm-C integrators, Active-RC integrators and the ideal $\hat{u}$ reconstructed signal. The PSD (dB/Hz) describes how power is distributed across different frequencies in the signal. It helps analyze noise shaping, distortion, and filtering performance of the circuits.

At low-frequency regions, both architectures show similar PSD behavior. Noise levels are relatively low and increase slightly as frequency increases. At high-frequency regions, both architectures exhibit increased noise at higher frequencies, but Active-RC implementation shows more fluctuations, likely due to high-frequency instability or non-ideal effects.

Conclusion: Gm-C may be a better choice for applications requiring lower mid-band noise and more predictable high-frequency behavior.

```
plt.figure()
plt.psd(ct_u_hat.flatten(), NFFT=1024, Fs=1 / ct_analog_frontend.dt, label="u_hat ideal")
plt.psd(gmc_u_hat.flatten(), NFFT=1024, Fs=1 / GmC.dt, label="u_hat GmC")
plt.psd(active_rc_u_hat.flatten(), NFFT=1024, Fs=1 / active_RC.dt, label="u_hat Active-RC")
plt.legend()
plt.xlabel("Frequency [Hz]")
plt.ylabel("PSD [dB/Hz]")
plt.title("Power Spectral Density")
plt.xscale("log")
```

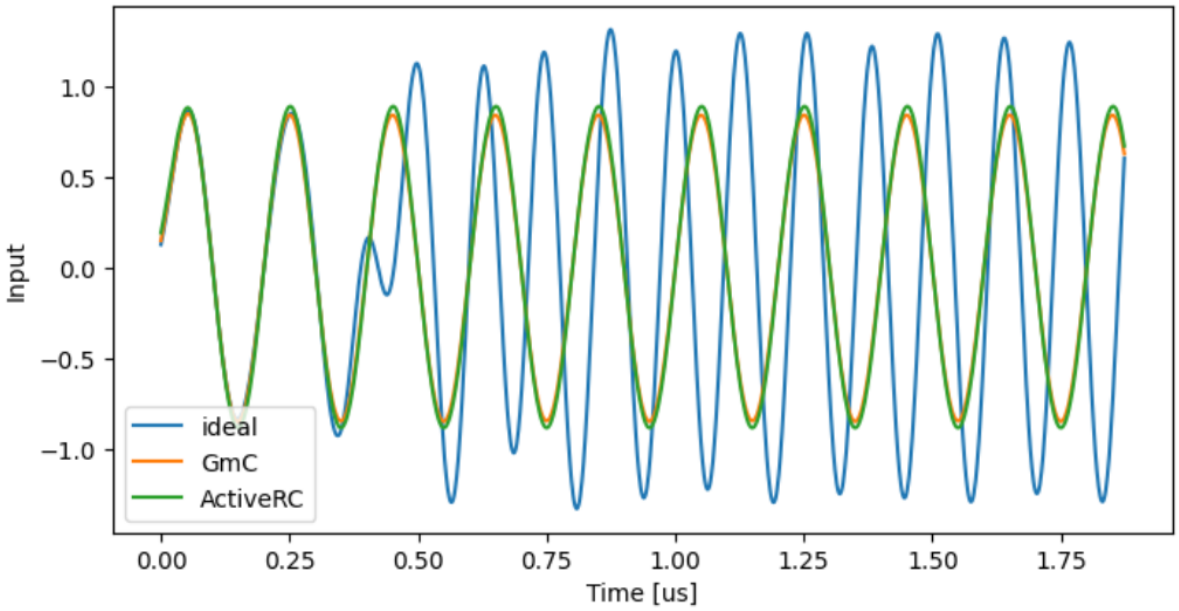


Now, the Active-RC architecture exhibits a strong noise peak at around 10^7 Hz, similar to the previous analysis. The Gm-C circuit follows the ideal signal more closely and has less noise variation compared to Active-RC. The ideal signal is smoother in the mid-frequency range, confirming that noise introduced by non-idealities affects the actual implementations.

Both GmC and Active-RC exhibit increasing noise at high frequencies, but the Active-RC implementation has more pronounced oscillatory behavior, indicating higher sensitivity to non-idealities at high frequencies.

Conclusion: Gm-C appears to be the better choice for minimizing mid-band noise and achieving a closer match to the ideal reconstruction.

```
plt.figure()
length = 1200
plt.plot(1e6 * ct_af_sim["t"][:length], ct_u_hat[:length], label="ideal")
plt.plot(1e6 * gmc_sim["t"][:length], gmc_u_hat[:length], label="GmC")
plt.plot(1e6 * active_RC_sim["t"][:length], active_rc_u_hat[:length], label="ActiveRC")
plt.legend()
plt.xlabel("Time [us]")
plt.ylabel("Input")
```



The graph compares the input signals of the three ΣΔMs. At lower frequencies, all three implementations overlap significantly, meaning they behave similarly in this range. As the frequency increases, the ideal signal starts to show more pronounced oscillations, while the Gm-C and Active-RC implementations smooth out these oscillations, likely due to real-world circuit effects like filtering and bandwidth constraints. The Gm-C and Active-RC signals are almost indistinguishable, suggesting they have similar performance in terms of maintaining the input characteristics.

2.4 ΣΔM design exploration

Date: Febrary 17, 2025

In this section, we run a new notebook from the beginning: `SDM_desing_exploration.ipynb`.

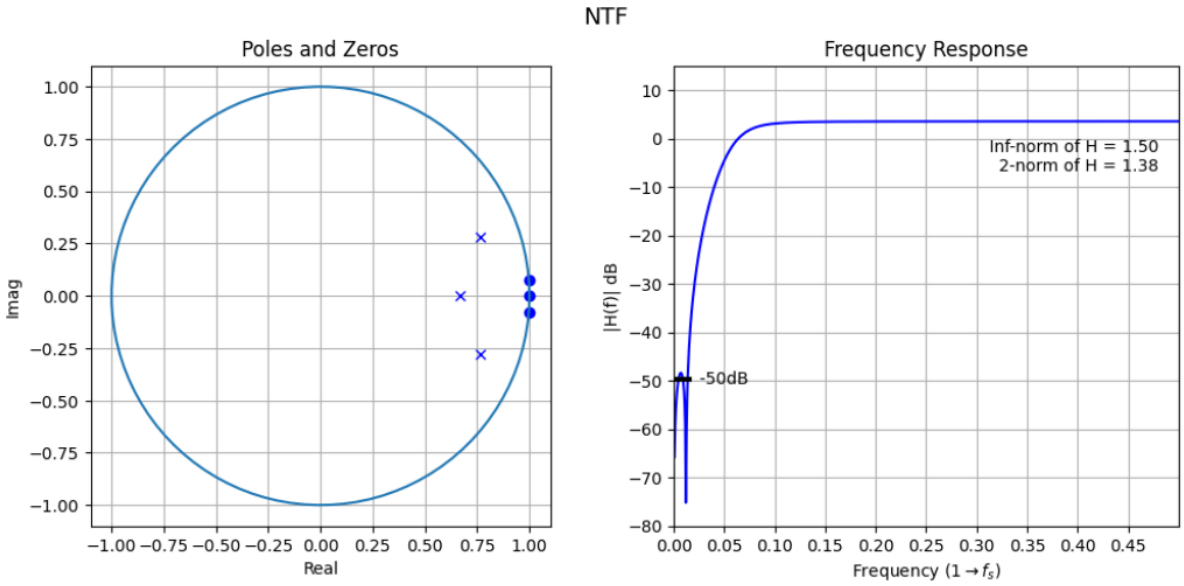
```
%pip install git+https://github.com/hammal/cbadc.git@feature/0.4.0
```

```
# import necessary libraries:
import numpy as np
import cbadc as cb
import matplotlib.pyplot as plt
```

```
# defining base parameters:
order = 3 # modulator order
osr = 32 # oversampling ratio
nlev = 2 # number of DAC levels
f0 = 0.0 # input frequency (0 for baseband)
Hinf = 1.5 # feedback gain
tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
dt_form = "CRFB" # time-domain structure
Bw = 10e6 # bandwidth of interest

# compute the sampling frequency:
fs = Bw * osr * 2
print(f"Sampling frequency: {fs / 1e6} MHz")
```

```
# synthesize the Noise Transfer Function (NTF):
ntf0 = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)
```

```
# visualize poles and zeros of the NTF:
cb.delsig.DocumentNTF(ntf0, osr, f0, False)
```



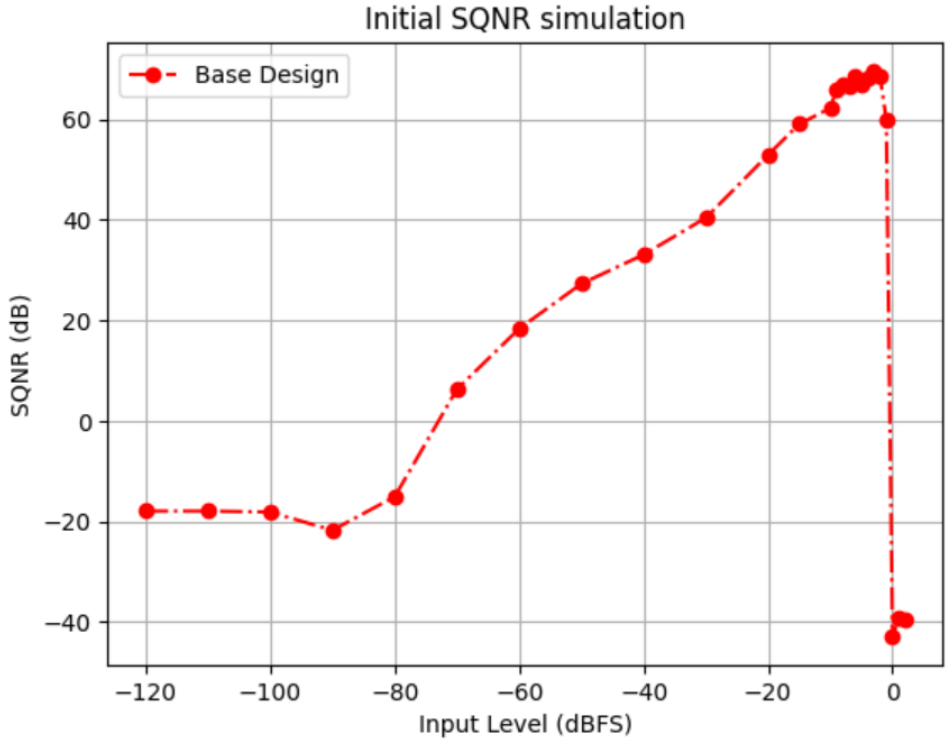

```
# realize the NTF in state-space representation:
a, g, b, c = cb.delsig.realizeNTF(ntf0, form="CRFB")
ABCD = cb.delsig.stuffABCD(a, g, b, c)
```

```python
# create discrete-time analog frontend model:
dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)
print(dt_analog_frontend)
```

```
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00578018]
 [ 1.          1.          0.99421982]],
B=
[[ 0.04438739 -0.04438739]
 [ 0.2398605  -0.2398605 ]
 [ 0.7967304  -0.7967304 ]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
```

```python
# run an initial SNR simulation as a baseline:
snr_base, amp_base, _ = dt_analog_frontend.simulateSNR(osr)
```

```python
# plot the initial SQNR results:
plt.figure()
plt.plot(amp_base, snr_base, 'o-.r', label='Base design')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.title('Initial SQNR simulation')
plt.grid()
plt.legend()
plt.show()
```



Now we have a structured base for the ΣΔM design exploration, the goal of this section is to systematically analyze **how different design parameters affect the performance** of the ΣΔM. Specifically, we aim to:

1. **Study the impact of key parameters** such as:
   - Modulator order.
   - Oversampling ratio.
   - Loop gain.
2. **Compare different architectures** of the ΣΔM.
3. **Optimize the design** based on SNR performance.

2.4.1 Effect of varying the modulator order

In this study, we analyze the behavior of ΣΔMs by varying the order of the modulator. The objective is to observe how increasing the order affects the pole-zero distribution and the resulting SNR.

```
# defining base model parameters:
osr = 32 # oversampling ratio
nlev = 2 # number of DAC levels
f0 = 0. # input frequency (0 for baseband)
Hinf = 1.5 # feedback gain
tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
dt_form = "CRFB" # time-domain structure
Bw = 10e6 # bandwidth of interest

```

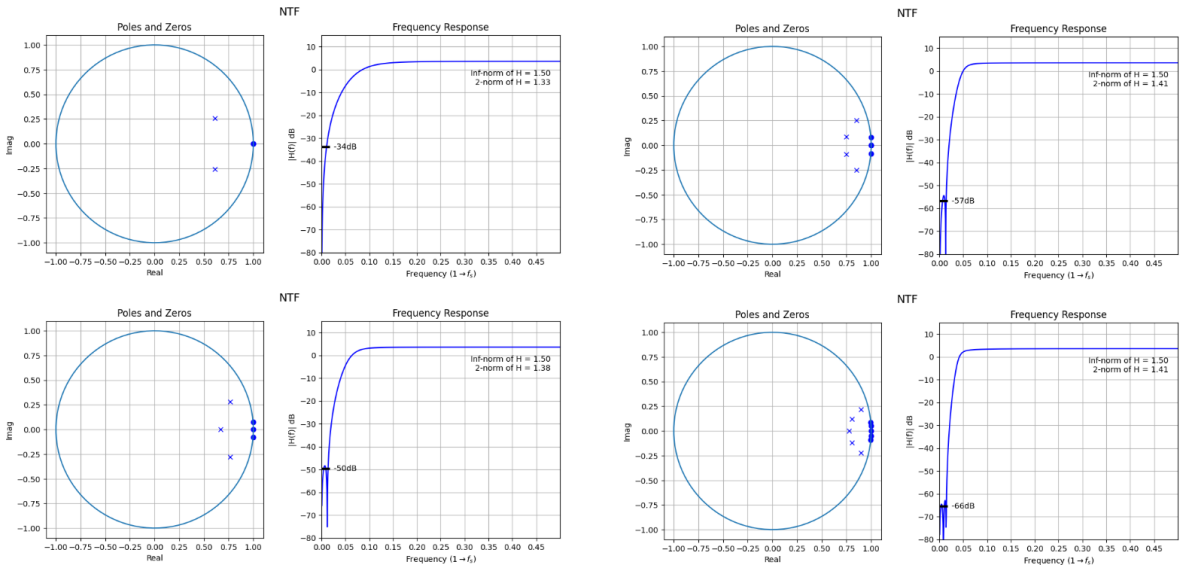
```python
# compute the sampling frequency:
fs = Bw * osr * 2
print(f"Sampling frequency: {fs / 1e6} MHz\n\n")

# explore different modulator orders:
orders = [2, 3, 4, 5]
snr_results_order = {}

for order in orders:
    print("###################################")
    print(f"Testing modulator order: {order}")
    print("###################################")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)

    # visualize poles and zeros of the NTF:
    cb.delsig.DocumentNTF(ntf, osr, f0, False)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form="CRFB")
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)
    print(dt_analog_frontend)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    snr_results_order[order] = (amp, snr)

    print("\n\n================================================================================\n\n")

# plot the SNR comparison:
plt.figure()
for order, (amp, snr) in snr_results_order.items():
    plt.plot(amp, snr, 'o-.', label=f'Order {order}')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.title('SNR comparison for different modulator orders')
plt.grid()
plt.legend()
plt.show()
```

```
INFO:root:Simulating discrete-time analog frontend
Sampling frequency: 640.0 MHz


###################################
Testing modulator order: 2
###################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1. -0.]
 [ 1.  1.]],
```

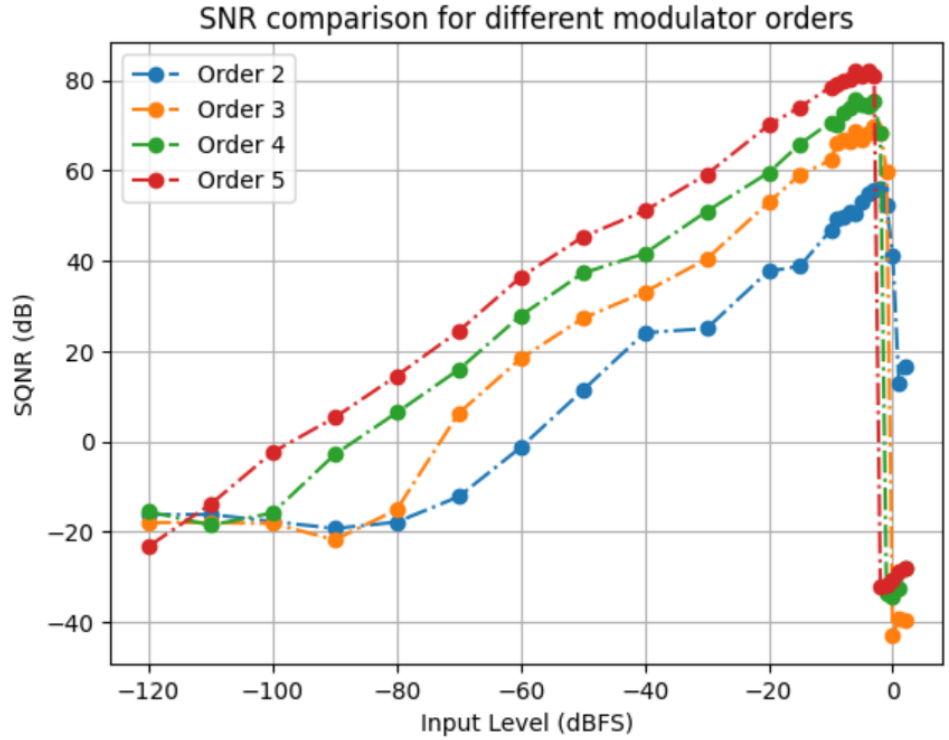
```
B=
[[ 0.21637021 -0.21637021]
 [ 0.77485177 -0.77485177]],
C=
[[0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=2,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================


##################################
Testing modulator order: 3
##################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00578018]
 [ 1.          1.          0.99421982]],
B=
[[ 0.04438739 -0.04438739]
 [ 0.2398605  -0.2398605 ]
 [ 0.7967304  -0.7967304 ]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================
```

```
##################################
Testing modulator order: 4
##################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.         -0.          0.          0.        ]
 [ 1.          1.          0.          0.        ]
 [ 0.          1.          1.         -0.00688054]
 [ 0.          1.          1.          0.99311946]],
B=
[[ 0.00623566 -0.00623566]
 [ 0.05924456 -0.05924456]
 [ 0.24493638 -0.24493638]
 [ 0.80201979 -0.80201979]],
C=
[[0. 0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=4,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================


##################################
Testing modulator order: 5
##################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.          0.          0.        ]
 [ 1.          1.         -0.00279396  0.          0.        ]
 [ 1.          1.          0.99720604  0.          0.        ]
 [ 0.          0.          1.          1.         -0.00790937]
 [ 0.          0.          1.          1.          0.99209063]],
B=
[[ 6.75559806e-04 -6.75559806e-04]
 [ 8.37752565e-03 -8.37752565e-03]
 [ 6.33294166e-02 -6.33294166e-02]
 [ 2.44344030e-01 -2.44344030e-01]
 [ 8.02273699e-01 -8.02273699e-01]],
C=
```

```
[[0. 0. 0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=5,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.

================================================================================
```



These plots illustrate the locations of the poles and zeros of the NTF for each order of the ΣΔM. As the order increases, the zeros of the NTF cluster more tightly around the signal band, improving noise shaping. Higher-order modulators introduce additional poles, which must remain within the unit circle to ensure stability.



SNR improves significantly with increasing order due to better noise shaping. Higher-order modulators push quantization noise further out of the signal band, enhancing overall performance.

2.4.2 Effect of varying the oversampling ratio

Another crucial aspect of ΣΔM performance is the oversampling ratio (OSR). By increasing OSR, more quantization noise is pushed outside the signal bandwidth, which can then be filtered out, leading to better SNR.

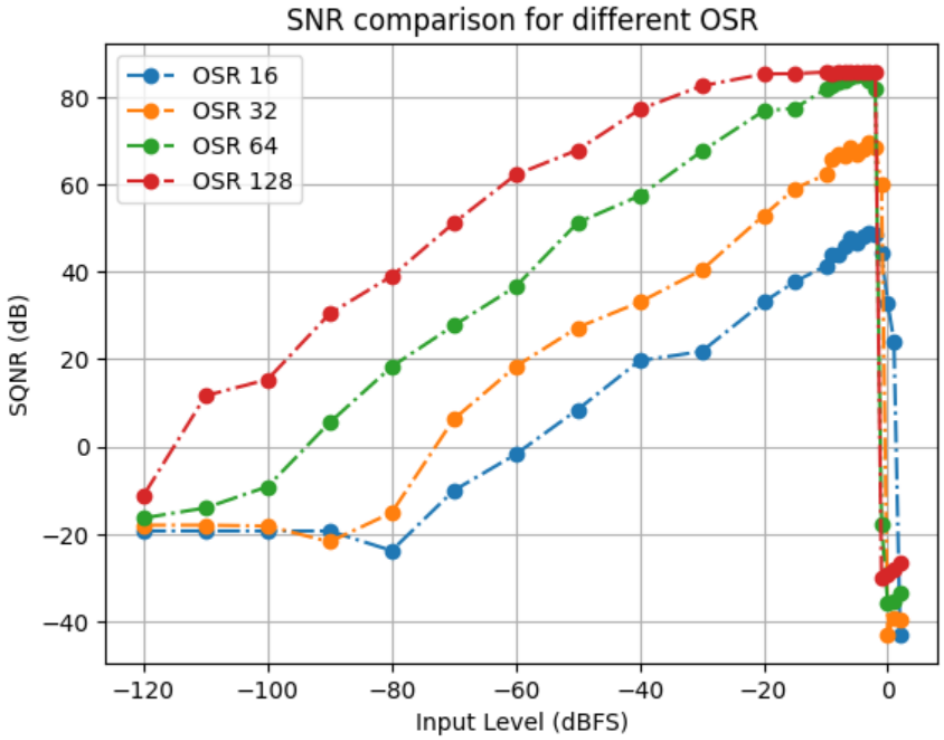
Now we investigate the effect of different OSR values (16, 32, 64, 128) while keeping other modulator parameters constant.

```
# defining base model parameters:
order = 3 # modulator order
nlev = 2 # number of DAC levels
f0 = 0. # input frequency (0 for baseband)
Hinf = 1.5 # feedback gain
tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
dt_form = "CRFB" # time-domain structure
Bw = 10e6 # bandwidth of interest

# explore different OSR:
osr_values = [16, 32, 64, 128]
snr_results_osr = {}

for osr in osr_values:
    print("####################################")
    print(f"Testing OSR: {osr}")
```

```
    print("###################################")

    # compute the sampling frequency:
    fs = Bw * osr * 2
    print(f"\nSampling Frequency: {fs / 1e6} MHz\n")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form="CRFB")
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    snr_results_osr[osr] = (amp, snr)

    print("\n\n================================================================================\n\n")

# plot the SNR comparison:
plt.figure()
for osr, (amp, snr) in snr_results_osr.items():
    plt.plot(amp, snr, 'o-.', label=f'OSR {osr}')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.title('SNR comparison for different OSR')
plt.grid()
plt.legend()
plt.show()
```

```
INFO:root:Simulating discrete-time analog frontend
###################################
Testing OSR: 16
###################################

Sampling Frequency: 320.0 MHz

WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

================================================================================


###################################
Testing OSR: 32
###################################

Sampling Frequency: 640.0 MHz

WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend
```

```
======================================================================


###################################
Testing OSR: 64
###################################

Sampling Frequency: 1280.0 MHz

WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================


###################################
Testing OSR: 128
###################################

Sampling Frequency: 2560.0 MHz

WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.

======================================================================
```



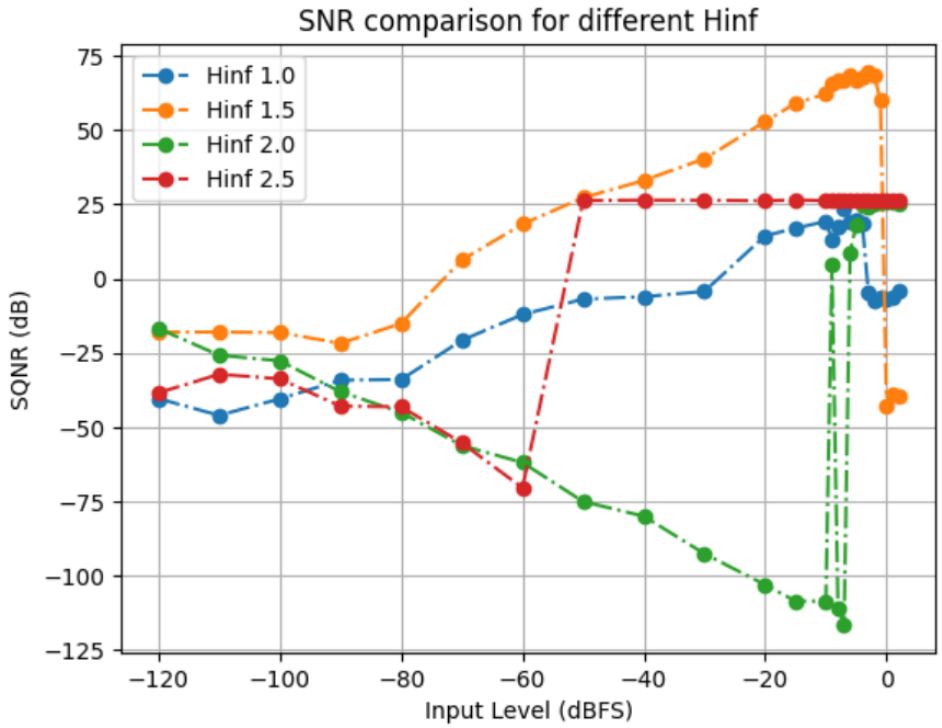
The plotted results show a clear trend: **as OSR increases, the Signal-to-Quantization Noise Ratio (SQNR) improves**.

2.4.3 Effect of varying the gain of the modulator

The feedback gain parameter, Hinf, significantly influences the noise shaping performance of an SDM. By modifying Hinf, we can adjust the trade-off between noise shaping and stability.

The experiment explores different values of Hinf (1.0, 1.5, 2.0, and 2.5) while keeping the order and oversampling ratio constant.

```
# defining base model parameters:
order = 3 # modulator order
osr = 32 # oversampling ratio
nlev = 2 # number of DAC levels
f0 = 0. # input frequency (0 for baseband)
tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
dt_form = "CRFB" # time-domain structure
Bw = 10e6 # bandwidth of interest

# compute the sampling frequency:
fs = Bw * osr * 2
print(f"Sampling frequency: {fs / 1e6} MHz\n\n")

# explore different Hinf gain values:
Hinf_values = [1.0, 1.5, 2.0, 2.5]
snr_results_hinf = {}

for Hinf in Hinf_values:
    print("####################################")
    print(f"Testing Hinf Gain: {Hinf}")
    print("####################################")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form="CRFB")
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    snr_results_hinf[Hinf] = (amp, snr)

    print("\n\n================================================================================\n\n")

# plot the SNR comparison:
plt.figure()
for Hinf, (amp, snr) in snr_results_hinf.items():
    plt.plot(amp, snr, 'o-.', label=f'Hinf {Hinf}')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.title('SNR comparison for different Hinf')
plt.grid()
plt.legend()
plt.show()
```



The simulation shows that SNR varies with different Hinf values. **The optimal performance (the highest SQNR) is achieved for** $H_\infty = 1.5$. For values lower than 1.5, the noise shaping effect is weaker, leading to lower SNR. For values higher than 1.5, performance degrades, possibly due to stability issues or increased noise amplification.

2.4.4 Effect of varying the form of the modulator

Now we experiment varying the architecture of the modulator, chosing between all the forms available on Hampus' repository: CRFB, CRFF, CIFB, CIFF, CRFBD, CRFFD.

```
# defining base model parameters:
order = 3 # modulator order
osr = 32 # oversampling ratio
nlev = 2 # number of DAC levels
f0 = 0. # input frequency (0 for baseband)
Hinf = 1.5 # feedback gain
tdac = [0, 1] # DAC sampling times
Bw = 10e6 # bandwidth of interest

# compute the sampling frequency:
fs = Bw * osr * 2
print(f"Sampling frequency: {fs / 1e6} MHz\n\n")

# explore different forms of the modulator:
forms = ["CRFB", "CRFF", "CIFB", "CIFF", "CRFBD", "CRFFD"]
snr_results_forms = {}

for form in forms:
    print("####################################")
    print(f"Testing modulator form: {form}")
```

```
    print("####################################")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form)
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)
    print(dt_analog_frontend)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    snr_results_forms[form] = (amp, snr)

    print("\n\n================================================================================\n\n")

# plot the SNR comparison:
plt.figure()
for form, (amp, snr) in snr_results_forms.items():
    plt.plot(amp, snr, 'o-.', label=f'{form} Architecture')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.title('SNR comparison for different architectures')
plt.grid()
plt.legend()
plt.show()
```

```
INFO:root:Simulating discrete-time analog frontend
Sampling frequency: 640.0 MHz


####################################
Testing modulator form: CRFB
####################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00578018]
 [ 1.          1.          0.99421982]],
B=
[[ 0.04438739 -0.04438739]
 [ 0.2398605  -0.2398605 ]
 [ 0.7967304  -0.7967304 ]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
```

```
N=3,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================


###################################
Testing modulator form: CRFF
###################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00578018]
 [ 1.          1.          0.99421982]],
B=
[[ 1.         -0.5568699 ]
 [ 0.         -0.2398605 ]
 [ 0.         -0.28102908]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================


###################################
Testing modulator form: CIFB
###################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00577183]
 [ 1.          1.          0.99422817]],
B=
```

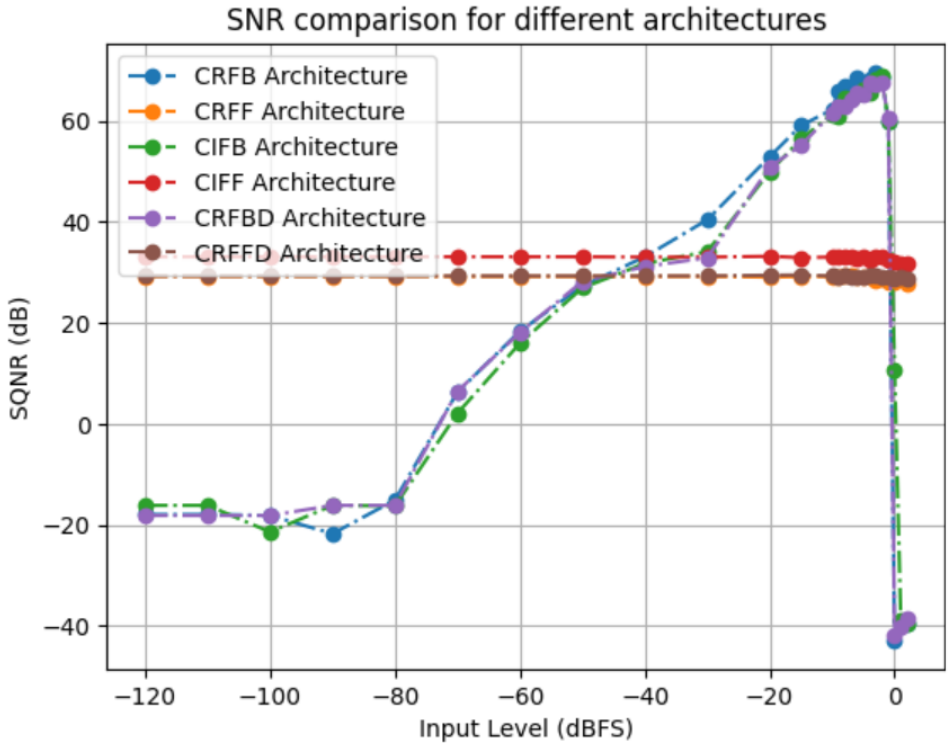
```
[[ 0.04438739 -0.04438739]
 [ 0.28425624 -0.28425624]
 [ 1.08676683 -1.08676683]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================


###################################
Testing modulator form: CIFF
###################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00577183]
 [ 1.          1.          0.99422817]],
B=
[[ 1.         -0.80251059]
 [ 0.         -0.28425624]
 [ 0.         -0.32401167]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================
```

```
###################################
Testing modulator form: CRFBD
###################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00578018]
 [ 1.          1.          0.99421982]],
B=
[[ 0.04438739 -0.04438739]
 [ 0.2398605  -0.2398605 ]
 [ 1.0365909  -1.0365909 ]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend

======================================================================


###################################
Testing modulator form: CRFFD
###################################
AnalogFrontend(
  analog_filter=StateSpace(
A=
[[ 1.          0.          0.        ]
 [ 1.          1.         -0.00578018]
 [ 1.          1.          0.99421982]],
B=
[[ 1.         -0.7967304 ]
 [ 0.         -0.2398605 ]
 [ 0.         -0.27964264]],
C=
[[0. 0. 1.]],
D=
[[1. 0.]]),
)
digital_control=DigitalControl(M=1, dt=1.0, dac_waveform=nrz),
analog_signal=AnalogSignal(offset=[[0.]], L=1, piecewise_constant=True),
```

```
state_covariance=None,
output_covariance=None,
N=3,
L=1,
M=1,
dt=1.0
)
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.

======================================================================
```

Comparison of state-space matrixs:

- In **Feedback** (FB) architectures, the B matrix entries contain more balanced positive and negative terms, allowing for greater control over noise shaping. Additionally, the A matrix has more off-diagonal terms, indicating a more interconnected structure that helps improve noise quantization modeling.
- In **Feedforward** (FF) architectures, the B matrix shows greater disparity in values, with more dominant terms in the first row, suggesting less effective noise shaping control. The A matrix has fewer off-diagonal interactions, indicating less internal feedback, which aligns with the more stable but lower SNR performance observed in the graph.



- **Feedback** (FB) architectures: The CRFB, CIFB, and CRFBD architectures exhibit similar behavior, with a progressive improvement in SNR as the input level increases. The SNR reaches an optimal point around -5 dBFS, with values exceeding 60 dB. However, as the input level approaches 0 dBFS, a sharp drop in performance occurs due to modulator overload.

- **Feedforward** (FF) architectures: The CRFF, CIFF, and CRFFD architectures show a completely different behavior. The SNR remains more stable compared to feedback architectures, without a significant increase as the input level rises. However, feedforward architectures tend to saturate earlier and do not achieve the maximum SNR levels observed in feedback architectures. In particular, CRFF and CRFFD show lower performance than CIFF, with an SNR seemingly limited to around 30-40 dB, indicating that they may not be ideal for effective noise reduction.

In conclusion, **if the goal is to maximize SNR over a wide input range, feedback architectures appear to be the best choices**. On the other hand, **feedforward architectures offer stability but at the cost of lower SNR performance**. The choice of architecture should depend on the trade-off between stability and SNR maximization, where feedback architectures provide better overall performance at the expense of increased implementation complexity.

2.5 Brute-force search to find optimal design parameters

Let's try to solve this problem: **find the optimal design parameters for a ΣΔM that meets the following requirements**:

- **Target SNR: 90 dB**.
- **Bandwidth: 20 kHz**.

To achieve this, we will **brute-force** search through different parameter combinations and identify the best configuration that maximizes SNR while keeping design complexity manageable.

We will vary the following parameters to determine the best configuration:

- Modulator order: between 2 and 6.
- Oversampling ratio: range 16 to 256.
- Number of DAC levels: 2 or 3.
- Hinf gain: between 1.0 and 2.5.

We will iterate through all possible combinations of these parameters and run SNR simulations for each case. The best configuration will be the one that achieves at least 90 dB SNR with the lowest complexity (for instance, lower order and OSR if multiple options).

2.5.1 First approach

```
%pip install git+https://github.com/hammal/cbadc.git@feature/0.4.0

# import necessary libraries:
import numpy as np
import cbadc as cb
import matplotlib.pyplot as plt
from itertools import product
```

```
# target specifications:
target_SNR = 90
Bw = 20e3
```

```
# parameter search space:
orders = [2, 3, 4, 5, 6] # modulator order
osr_values = [16, 32, 64, 128, 256] # oversampling ratio
nlev_values = [2, 3] # number of DAC levels
Hinf_values = [1.0, 1.5, 2.0, 2.5] # feedback gain

tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
```

```
# store results:
best_config = None
best_snr = 0
all_results = []
```

```
# brute-force search:
for order, osr, nlev, Hinf in product(orders, osr_values, nlev_values, Hinf_values):
    # compute the sampling frequency:
    fs = Bw * osr * 2
```

```
    print(f"Sampling frequency: {fs / 1e6} MHz")

    print("###################################")
    print(f"Testing: order={order}, osr={osr}, nlev={nlev}, Hinf={Hinf}")
    print("###################################")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, 0.0)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form="CRFB")
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # Create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    max_snr = max(snr)

    # store result:
    all_results.append((order, osr, nlev, Hinf, max_snr))

    # check if this configuration meets the target SNR:
    if max_snr >= target_SNR and (best_config is None or max_snr > best_snr):
        best_snr = max_snr
        best_config = (order, osr, nlev, Hinf)
```

```
INFO:root:Simulating discrete-time analog frontend
Sampling frequency: 0.64 MHz
###################################
Testing: order=2, osr=16, nlev=2, Hinf=1.0
###################################
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend
Sampling frequency: 0.64 MHz
###################################
Testing: order=2, osr=16, nlev=2, Hinf=1.5
###################################
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend
Sampling frequency: 0.64 MHz
###################################
Testing: order=2, osr=16, nlev=2, Hinf=2.0
###################################
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
INFO:root:Simulating discrete-time analog frontend
Sampling frequency: 0.64 MHz
###################################
Testing: order=2, osr=16, nlev=2, Hinf=2.5
###################################
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
```

```
INFO:root:Simulating discrete-time analog frontend
...
```

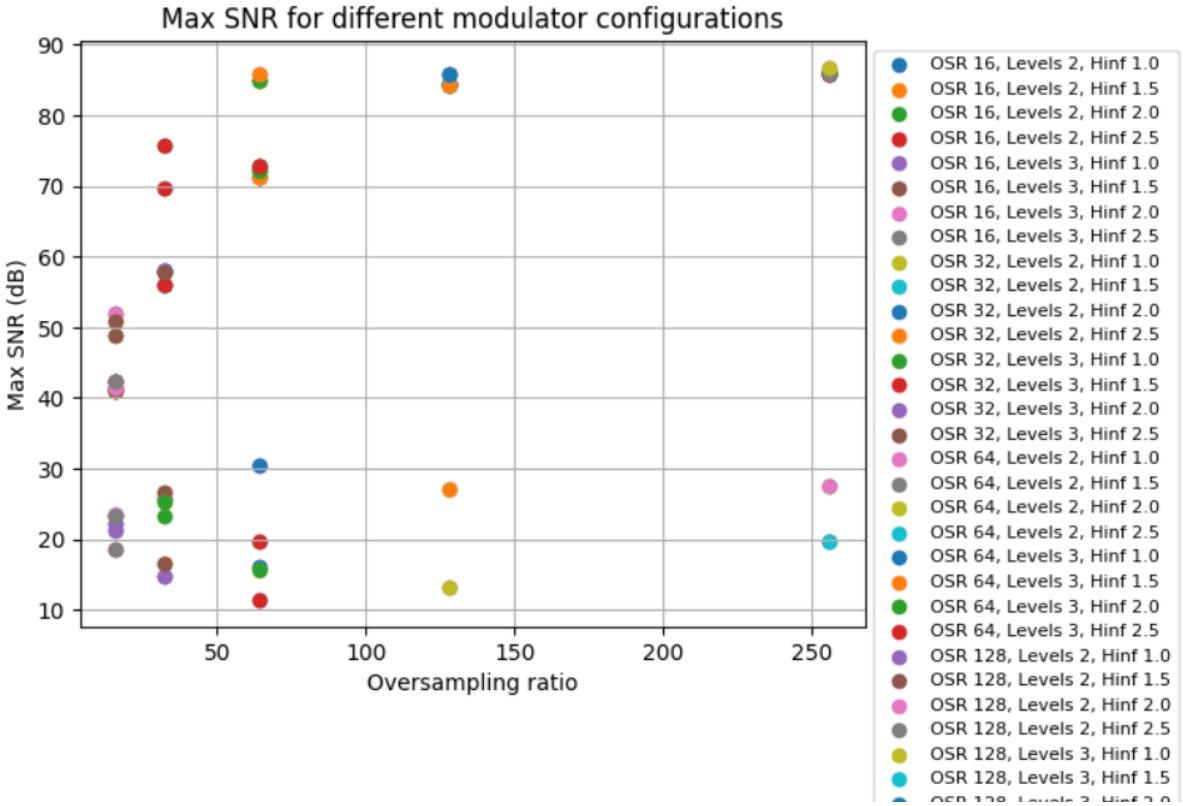
```python
# display the best configuration:
if best_config:
    print("Best configuration found:")
    print(f"\n\tOrder: {best_config[0]}\n\tOSR: {best_config[1]}\n\tDAC Levels:
        {best_config[2]}\n\tHinf: {best_config[3]}\n\tSNR: {best_snr:.2f} dB")
else:
    print("No configuration met the target SNR")
```

```
No configuration met the target SNR
```

We don't get any result. If we look closer to the output of the algorithm, we find that it has stopped before checking all configurations, because one of them (maybe more, but it's the first one) returns failure:

```
###################################
Testing: order=4, osr=128, nlev=2, Hinf=1.0
###################################
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
Cell In[11], line 22
     19 dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)
     21 # run an SNR simulation:
---> 22 snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
     23 max_snr = max(snr)
     25 # store result:
...
```

```python
# plot SNR results:
plt.figure()
for order, osr, nlev, Hinf, max_snr in all_results:
    plt.scatter(osr, max_snr, label=f'OSR {osr}, Levels {nlev}, Hinf {Hinf}')
plt.xlabel('Oversampling ratio')
plt.ylabel('Max SNR (dB)')
plt.title('Max SNR for different modulator configurations')
plt.grid()
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), fontsize=8)
plt.show()
```



Looking at the plot, we see that none of the combinations studied reaches 90 dB of SNR, and that is why we did not get any solution.

2.5.2 Second approach

We need to fix this error. We have two options: try to follow up and fix the error we have encountered, or solve the problem in another way. The first solution is difficult to carry out, since it is an internal error of the repository for specific values. Therefore, we will follow the second one.

We are going to modify the target variable SNR, and set it to 85 dB, since we have seen in the plot that it is achieved. We are going to see what is the best configuration that we obtain for this new problem, and try to follow that way to solve the original one.

```
# target specifications:
target_SNR2 = 85
Bw = 20e3
```

```
# parameter search space:
orders = [2, 3, 4, 5, 6] # modulator order
osr_values = [16, 32, 64, 128, 256] # oversampling ratio
nlev_values = [2, 3] # number of DAC levels
Hinf_values = [1.0, 1.5, 2.0, 2.5] # feedback gain

tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
```

```
# store results:
best_config = None # best parameter configuration (order, osr, nlev, Hinf)
```

```
best_snr = 0 # maximum SNR found
all_results = [] # all tested configurations, and their SNR
```

```
# brute-force search:
for order, osr, nlev, Hinf in product(orders, osr_values, nlev_values, Hinf_values):
    # compute the sampling frequency:
    fs = Bw * osr * 2
    print(f"Sampling frequency: {fs / 1e6} MHz")

    print("####################################")
    print(f"Testing: order={order}, osr={osr}, nlev={nlev}, Hinf={Hinf}")
    print("####################################")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, 0.0)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form="CRFB")
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # Create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    max_snr = max(snr)

    # store result:
    all_results.append((order, osr, nlev, Hinf, max_snr))

    # check if this configuration meets the target SNR:
    if max_snr >= target_SNR2 and (best_config is None or max_snr > best_snr):
        best_snr = max_snr
        best_config = (order, osr, nlev, Hinf)
```

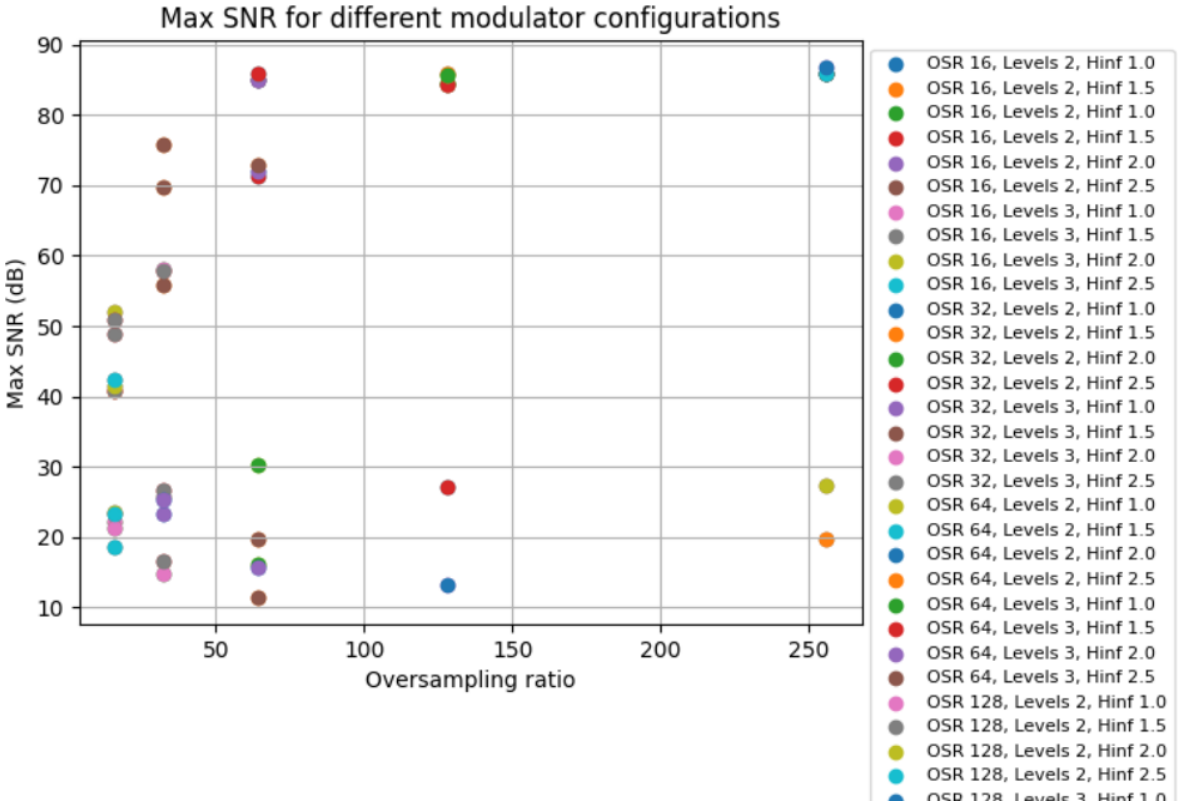
```
# display the best configuration:
if best_config:
    print("Best configuration found:")
    print(f"\n\tOrder: {best_config[0]}\n\tOSR: {best_config[1]}\n\tDAC Levels:
        {best_config[2]}\n\tHinf: {best_config[3]}\n\tSNR: {best_snr:.2f} dB")
else:
    print("No configuration met the target SNR")
```

```
Best configuration found:

        Order: 3
        OSR: 256
        DAC Levels: 2
        Hinf: 2.0
        SNR: 86.74 dB
```

Now, we get an output, although the algorithm returns an error.

```
# plot SNR results for visualization:
plt.figure()
for order, osr, nlev, Hinf, max_snr in all_results:
    plt.scatter(order, max_snr, label=f'OSR {osr}, Levels {nlev}, Hinf {Hinf}')
```

```
plt.xlabel('Modulator order')
plt.ylabel('Max SNR (dB)')
plt.title('Max SNR for different modulator configurations')
plt.grid()
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), fontsize=8)
plt.show()
```



So, as we have obtained previously, and we can also see in the plot, the best configuration is:

- Order of the modulator: 3
- Oversampling ratio: 256
- Number of DAC levels: 2
- Hinf gain: 2.0
- SNR: 86.74

Let's analyze the results, comparing with the parameter search space we defined previously. It seems that as increasing oversampling ratio, SNR also increases. So, it may means that if we increase OSR, SNR will achieve $SNR = 90\text{dB}$.

Then, the next step is to repeat the process, but only varying the oversampling ratio (and Hinf gain, because in previous experiments we obtained different results). Also, we now modify the architecutre of the modulator.

```
# target specifications:
target_SNR = 90
Bw = 20e3
```

```
# parameter search space:
order = 3 # modulator order
osr_values = [128, 256, 512, 1024] # oversampling ratio
nlev = 3 # number of DAC levels
Hinf_values = [1.5, 2.0] # feedback gain
forms = ["CRFB", "CRFF", "CIFB", "CIFF", "CRFBD", "CRFFD"] # modulator architectures
tdac = [0, 1] # DAC sampling times
```

```
# store results:
best_config = None # best parameter configuration (order, osr, nlev, Hinf, form)
best_snr = 0 # maximum SNR found
all_results = [] # all tested configurations, and their SNR
```

```
# brute-force search:
for osr, Hinf, form in product(osr_values, Hinf_values, forms):
    # compute the sampling frequency:
    fs = Bw * osr * 2
    print(f"Sampling frequency: {fs / 1e6} MHz")

    print("####################################")
    print(f"Testing: order={order}, osr={osr}, nlev={nlev}, Hinf={Hinf}, form={form}")
    print("####################################")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, 0.0)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form)
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # Create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    max_snr = max(snr)

    # store result:
    all_results.append((order, osr, nlev, Hinf, form, max_snr))

    # check if this configuration meets the target SNR:
    if max_snr >= target_SNR and (best_config is None or max_snr > best_snr):
        best_snr = max_snr
        best_config = (order, osr, nlev, Hinf, form)
```

```
# brute-force search:
for osr, Hinf, form in product(osr_values, Hinf_values, forms):
    # compute the sampling frequency:
    fs = Bw * osr * 2
    print(f"Sampling frequency: {fs / 1e6} MHz\n\n")

    print("####################################")
    print(f"Testing: order={order}, osr={osr}, nlev={nlev}, Hinf={Hinf}, form={form}")
```

```
    print("###################################")

    # synthesize the Noise Transfer Function (NTF):
    ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, 0.0)

    # realize the NTF in state-space representation:
    a, g, b, c = cb.delsig.realizeNTF(ntf, form)
    ABCD = cb.delsig.stuffABCD(a, g, b, c)

    # create discrete-time analog frontend model:
    dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

    # run an SNR simulation:
    snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
    max_snr = max(snr)

    # store result:
    all_results.append((order, osr, nlev, Hinf, form, max_snr))

    # check if this configuration meets the target SNR:
    if max_snr >= target_SNR and (best_config is None or max_snr > best_snr):
        best_snr = max_snr
        best_config = (order, osr, nlev, Hinf, form)
```

```
# display the best configuration:
if best_config:
    print("Best configuration found:")
    print(f"\n\tOrder: {best_config[0]}\n\tOSR: {best_config[1]}\n\tDAC Levels: {best_config[2]}\n\tHinf: {best_
else:
    print("No configuration met the target SNR")
```

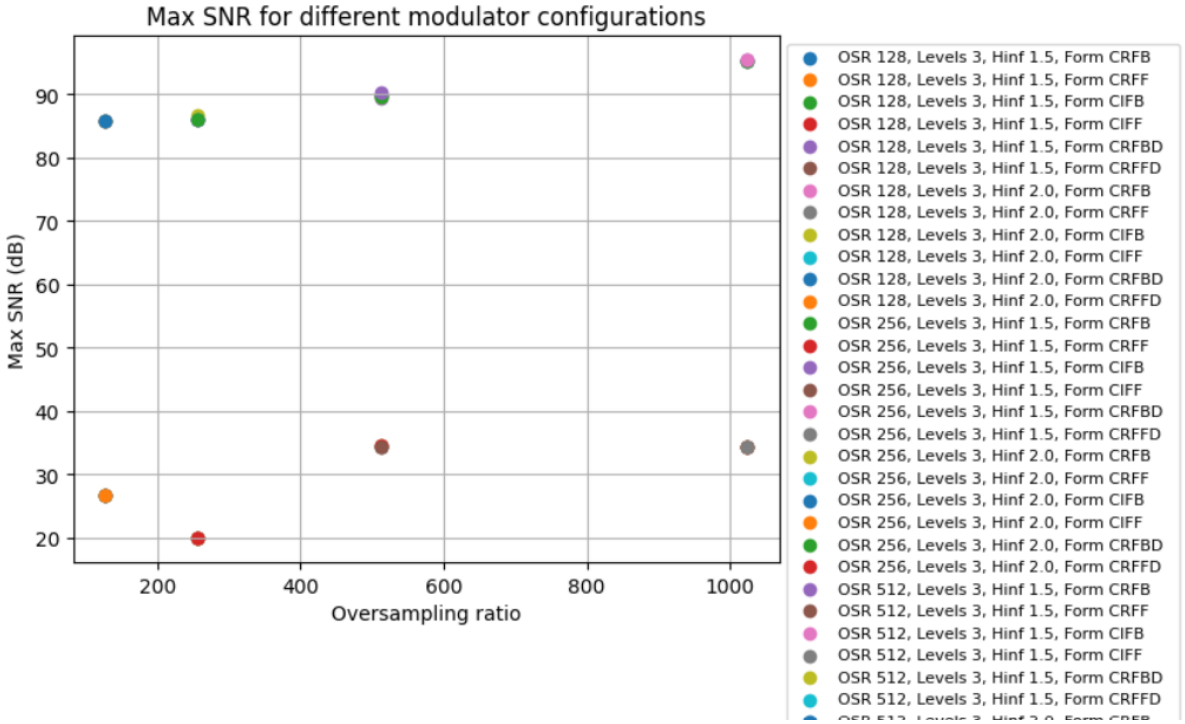
```
Best configuration found:

        Order: 3
        OSR: 1024
        DAC Levels: 3
        Hinf: 2.0
        Form: CRFBD
        SNR: 95.39 dB
```

Now, we can see that we did exceed the SNR target, and the configuration we get is:

- Order of the modulator: 3
- Oversampling ratio: 1024
- Number of DAC levels: 3
- Hinf gain: 2.0
- Architecture of the modulator: CRFBD
- SNR: 95.39

```
# plot SNR results for visualization:
plt.figure()
for order, osr, nlev, Hinf, form, max_snr in all_results:
```

```
    plt.scatter(osr, max_snr, label=f'OSR {osr}, Levels {nlev}, Hinf {Hinf}, Form {form}')
plt.xlabel('Oversampling ratio (OSR)')
plt.ylabel('Max SNR (dB)')
plt.title('Max SNR for different modulator configurations')
plt.grid()
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), fontsize=8)
plt.show()
```



However, this does not guarantee that it is the best configuration, we would have to search in a larger space since the oversampling ratio seems to be able to continue increasing. What is certain is that this way it no longer gives an error as it did before, ensuring that all the options have been explored.

To conclude, we have achieved that these are the **best parameters** for the specifications given:

- **Order** of the modulator: $L = 3$
- **Oversampling ratio**: $OSR = 1024$
- Number of **DAC levels**: 3
- **Hinf gain**: 2.0
- **Architecture** of the modulator: CRFBD
- **Signal-to-Noise Ratio**: $SNR = 95.39$dB
- **Bandwidth**: $B_w = 20$kHz

2.6 Finding the optimal implementation

Date: Febrary 18, 2025

Now we have found the optimal ΣΔM configuration, we need to **determine which circuit implementation (Active-RC or Gm-C) is better** for the design.

For each architecture, we need to sweep through different component values and determine the best-performing configuration:

1. For **Active-RC**:
    - Integration capacitance (Cint)
    - Transconductance (gm)
    - Internal state resistance (Ro)
    - Internal state capacitance (Co)
2. For **Gm-C**:
    - Larger integration capacitor (Cint)
    - Output resistance (Ro)
    - Parasitic capacitance (Cp)
    - Input referred noise density (v_n)
    - Minimum output voltage (v_out_min)
    - Maximum output voltage (v_out_max)
    - Slew rate (slew_rate)

2.6.1 First approach

```python
# optimal modulator parameters from brute-force search:
order = 3 # modulator order
osr = 1024 # oversampling ratio
nlev = 3 # number of DAC levels
f0 = 0. # input frequency (0 for baseband)
Hinf = 2.0 # feedback gain
tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
dt_form = "CRFBD" # time-domain structure
Bw = 20e3 # bandwidth

# compute the sampling frequency:
fs = Bw * osr * 2
print(f"Sampling frequency: {fs / 1e6} MHz")

# rescale to fs:
dt = 1.0 / fs
```

```
Sampling frequency: 40.96 MHz
```

```
# synthesize the optimal NTF:
ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)

# realize the NTF in state-space representation:
a, g, b, c = cb.delsig.realizeNTF(ntf, form)
ABCD = cb.delsig.stuffABCD(a, g, b, c)

# transform the DT NTF into its CT equivalent:
ABCDc, tdac2 = cb.delsig.realizeNTF_ct(ntf, form, tdac)
```

```
analog_frontend = cb.AnalogFrontend.ctsdm(ABCDc, tdac2, dt, nlev)
analog_frontend.dt = 1.0 / fs # probably not necessary
```

Once we have all defined, we can start with each implementation:

```
# parameter search space for Active-RC implementation:
Cint_values = [100e-15, 500e-15, 1e-12, 10e-12] # integration capacitance (F)
gm_values = [1e-6, 1e-5, 1e-4] # transconductance (S)
Ro_values = [1e4, 1e5, 1e6, 1e7] # internal resistance (Ohm)
Co_values = [1e-15, 100e-15, 1e-12] # internal capacitance (F)

# store results:
best_snr_ActiveRC = 0 # maximum SNR found
best_config_ActiveRC = None # best parameter configuration (Cint, gm, Ro, Co)
all_results_ActiveRC = [] # all tested configurations, and their SNR
```

```
# brute-force search for Active-RC:
for Cint, gm, Ro, Co in product(Cint_values, gm_values, Ro_values, Co_values):
    print("####################################")
    print(f"Testing Active-RC implementation: Cint={Cint}, gm={gm}, Ro={Ro}, Co={Co}")
    print("####################################")

    Cint = np.ones(order) * Cint
    gm = np.ones(order) * gm
    Ro = np.ones(order) * Ro
    Co = np.ones(order) * Co

    ActiveRC_analog_frontend = cb.ActiveRC(
        analog_frontend, Cint, gm, Ro, Co
    )

    snr_ActiveRC, amp_ActiveRC, _ = ActiveRC_analog_frontend.simulateSNR(osr)
    max_snr_ActiveRC = max(snr_ActiveRC)

    all_results_ActiveRC.append((Cint, gm, Ro, Co, max_snr_ActiveRC))

    if max_snr_ActiveRC > best_snr_ActiveRC:
        best_snr_ActiveRC = max_snr_ActiveRC
        best_config_ActiveRC = (Cint, gm, Ro, Co)
```

```
# parameter search space for Gm-C implementation:
Cint_values = [100e-15, 500e-15, 1e-12, 10e-12] # integration capacitance (F)
Ro_values = [1e4, 1e5, 1e6, 1e7] # output resistance (Ohm)
Cp_values = [1e-15, 100e-15, 1e-12] # parasitic capacitance (F)
v_n_values = [10e-6, 100e-6, 1e-3] # input noise density (V rms)
slew_rate_values = [1e6, 1e9, 1e12] # slew rate (V/s)
output_swing_values = [0.5, 1.0, 2.0] # output voltage swing (V)
```

```

# store results:
best_snr_GmC = 0 # maximum SNR found
best_config_GmC = None # best parameter configuration (Cint, Ro, Cp, v_n, slew_rate, v_out_max)
all_results_GmC = [] # all tested configurations, and their SNR

# brute-force search for Gm-C:
for Cint, Ro, Cp, v_n, slew_rate, output_swing in product(Cint_values, Ro_values, Cp_values,
        v_n_values, slew_rate_values, output_swing_values):
    print("####################################")
    print(f"Testing Gm-C implementation: Cint={Cint}, Ro={Ro}, Cp={Cp}, v_n={v_n},
        slew_rate={slew_rate}, output_swing={output_swing}")
    print("####################################")

    Cint = np.ones(order) * Cint
    Ro = np.ones(order) * Ro
    Cp = np.ones(order) * Cp
    v_n = np.ones(order) * v_n
    slew_rate = np.ones(order) * slew_rate
    v_out_max = np.ones(order) * output_swing
    v_out_min = -v_out_max

    GmC_analog_frontend = cb.GmC(
        analog_frontend, Cint, Ro, Cp, v_n, v_out_max, v_out_min, slew_rate
    )

    snr_GmC, amp_GmC, _ = GmC_analog_frontend.simulateSNR(osr)
    max_snr_GmC = max(snr_GmC)

    all_results_GmC.append((Cint, Ro, Cp, v_n, slew_rate, output_swing, max_snr_GmC))

    if max_snr_GmC > best_snr_GmC:
        best_snr_GmC = max_snr_GmC
        best_config_GmC = (Cint, Ro, Cp, v_n, slew_rate, v_out_max)

# display best configurations:
print("Best Active-RC configuration:")
print(f"\n\tCint: {best_config_ActiveRC[0][0]} F\n\tgm: {best_config_ActiveRC[1][0]} S\n\t
    Ro: {best_config_ActiveRC[2][0]} Ohm\n\tCo: {best_config_ActiveRC[3][0]} F\n\t
    SNR: {best_snr_ActiveRC:.2f} dB")
print("\nBest Gm-C configuration:")
print(f"\n\tCint: {best_config_GmC[0][0]} F\n\tRo: {best_config_GmC[1][0]} Ohm\n\t
    Cp: {best_config_GmC[2][0]} F\n\tv_n: {best_config_GmC[3][0]} V\n\t
    Slew rate: {best_config_GmC[4][0]} V/s\n\tOutput range: +-{best_config_GmC[5][0]} V\n\t
    SNR: {best_snr_GmC:.2f} dB")

# compare results and determine the best implementation:
if best_snr_ActiveRC > best_snr_GmC:
    print("Active-RC is the optimal implementation")
else:
    print("Gm-C is the optimal implementation")

# plot SNR results:
plt.figure()
plt.plot(amp, snr, 'o-.g', label='Ideal CT implementation')
plt.plot(amp_ActiveRC, snr_ActiveRC, 'o-.r', label='Active-RC')
plt.plot(amp_GmC, snr_GmC, 'o-.b', label='Gm-C')
```

```
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.grid()
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), fontsize=8)
```

```
# plot SNR results:
plt.figure()
plt.bar(['Active-RC', 'Gm-C'], [best_snr_ActiveRC, best_snr_GmC], color=['blue', 'red'])
plt.xlabel('Implementation')
plt.ylabel('Max SNR (dB)')
plt.title('Best SNR for Active-RC vs Gm-C')
plt.grid()
plt.show()
```

I consider this code should be fine for our purpose. However, it does not compile correctly. We might think that it simply takes too long to execute and is due to low computing power, but if we test it leaving only one value for each variable to be optimized, it still does not give an answer and keeps running indefinitely.

We are keeping this code for further analysis as a base code, and try to fix problems in the following sections.

2.6.2 Second approach

Investigating a little in the warnings errors it returns, we see that it is due to very low values, very close to 0. Therefore, although it is not the optimal solution, we will change the oversampling ratio a little and reduce its value by half. In this way, we do get answers. Probably it is not the best solution, but it is the only one I have found.

```
# "optimal" modulator parameters from brute-force search:
order = 3 # modulator order
osr = 512 # oversampling ratio
nlev = 3 # number of DAC levels
f0 = 0. # input frequency (0 for baseband)
Hinf = 2.0 # feedback gain
tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
dt_form = "CRFBD" # time-domain structure
Bw = 20e3 # bandwidth

# compute the sampling frequency:
fs = Bw * osr * 2
print(f"Sampling frequency: {fs / 1e6} MHz")

# rescale to fs:
dt = 1.0 / fs
```

```
Sampling frequency: 20.48 MHz
```

```
# synthesize the optimal NTF:
ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)

# realize the NTF in state-space representation:
a, g, b, c = cb.delsig.realizeNTF(ntf, dt_form)
```

```
ABCD = cb.delsig.stuffABCD(a, g, b, c)

# create discrete-time analog frontend model:
dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

# run an SNR simulation:
snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
max_snr = max(snr)
print(f"SNR = {max_snr} db")
```

```
INFO:root:Simulating discrete-time analog frontend
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
SNR = 90.18351828633857 db
```

We lose in SNR, but it's still greater than 90 db.

```
# transform the DT NTF into its CT equivalent:
ABCDc, tdac2 = cb.delsig.realizeNTF_ct(ntf, form, tdac)

analog_frontend = cb.AnalogFrontend.ctsdm(ABCDc, tdac2, dt, nlev)
analog_frontend.dt = 1.0 / fs # probably not necessary
```

Now, we can start with the implementations. In order to fix bugs, we are starting with **only one value per parameter**, and then we will apply brute-force search.

```
# parameter search space for Active-RC implementation:
Cint_values = [1e-12] # integration capacitance (F)
gm_values = [1e-4] # transconductance (S)
Ro_values = [1e4] # internal resistance (Ohm)
Co_values = [1e-12] # internal capacitance (F)

# store results:
best_snr_ActiveRC = 0 # maximum SNR found
best_config_ActiveRC = None # best parameter configuration (Cint, gm, Ro, Co)
all_results_ActiveRC = [] # all tested configurations, and their SNR
```

```
# brute-force search for Active-RC (only one value):
for Cint, gm, Ro, Co in product(Cint_values, gm_values, Ro_values, Co_values):
    print("####################################")
    print(f"Testing Active-RC implementation: Cint={Cint}, gm={gm}, Ro={Ro}, Co={Co}")
    print("####################################")

    Cint = np.ones(order) * Cint
    gm = np.ones(order) * gm
    Ro = np.ones(order) * Ro
    Co = np.ones(order) * Co

    ActiveRC_analog_frontend = cb.ActiveRC(
        analog_frontend, Cint, gm, Ro, Co
    )

    snr_ActiveRC, amp_ActiveRC, _ = ActiveRC_analog_frontend.simulateSNR(osr)
    max_snr_ActiveRC = max(snr_ActiveRC)

    all_results_ActiveRC.append((Cint, gm, Ro, Co, max_snr_ActiveRC))
```

```
20
21      if max_snr_ActiveRC > best_snr_ActiveRC:
22          best_snr_ActiveRC = max_snr_ActiveRC
23          best_config_ActiveRC = (Cint, gm, Ro, Co)
```

```
INFO:root:Simulating continuous-time analog frontend for sinusoidal input
####################################
Testing Active-RC implementation: Cint=1e-12, gm=0.0001, Ro=10000.0, Co=1e-12
####################################
```

```
1  # parameter search space for Gm-C implementation:
2  Cint_values = [1e-12] # integration capacitance (F)
3  Ro_values = [1e4] # output resistance (Ohm)
4  Cp_values = [1e-12] # parasitic capacitance (F)
5  v_n_values = [1e-3] # input noise density (V rms)
6  slew_rate_values = [1e-3] # slew rate (V/s)
7  output_swing_values = [1.0] # output voltage swing (V)
8
9  # store results:
10 best_snr_GmC = 0 # maximum SNR found
11 best_config_GmC = None # best parameter configuration (Cint, Ro, Cp, v_n, slew_rate, v_out_max)
12 all_results_GmC = [] # all tested configurations, and their SNR
```

```
1  # brute-force search for Gm-C (only one value):
2  for Cint, Ro, Cp, v_n, slew_rate, output_swing in product(Cint_values, Ro_values, Cp_values,
3          v_n_values, slew_rate_values, output_swing_values):
4      print("####################################")
5      print(f"Testing Gm-C implementation: Cint={Cint}, Ro={Ro}, Cp={Cp}, v_n={v_n},
6          slew_rate={slew_rate}, output_swing={output_swing}")
7      print("####################################")
8
9      Cint = np.ones(order) * Cint
10     Ro = np.ones(order) * Ro
11     Cp = np.ones(order) * Cp
12     v_n = np.ones(order) * v_n
13     slew_rate = np.ones(order) * slew_rate
14     v_out_max = np.ones(order) * output_swing
15     v_out_min = -v_out_max
16
17     GmC_analog_frontend = cb.GmC(
18         analog_frontend, Cint, Ro, Cp, v_n, v_out_max, v_out_min, slew_rate
19     )
20
21     snr_GmC, amp_GmC, _ = GmC_analog_frontend.simulateSNR(osr)
22     max_snr_GmC = max(snr_GmC)
23
24     all_results_GmC.append((Cint, Ro, Cp, v_n, slew_rate, output_swing, max_snr_GmC))
25
26     if max_snr_GmC > best_snr_GmC:
27         best_snr_GmC = max_snr_GmC
28         best_config_GmC = (Cint, Ro, Cp, v_n, slew_rate, v_out_max)
```

```
INFO:root:Simulating continuous-time analog frontend for sinusoidal input
####################################
```

```
Testing Gm-C implementation: Cint=1e-12, Ro=10000.0, Cp=1e-12, v_n=0.001, slew_rate=0.001,
    output_swing=1.0
###################################
```

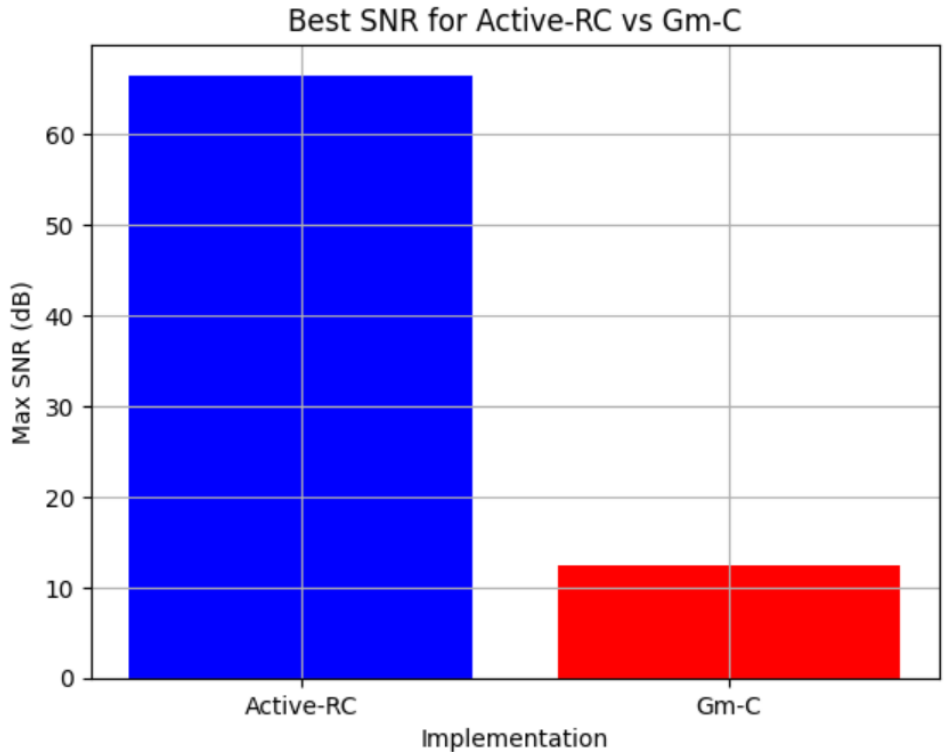
```python
# display best configurations:
print("Best Active-RC configuration:")
print(f"\n\tCint: {best_config_ActiveRC[0][0]} F\n\tgm: {best_config_ActiveRC[1][0]} S\n\t
    Ro: {best_config_ActiveRC[2][0]} Ohm\n\tCo: {best_config_ActiveRC[3][0]} F\n\t
    SNR: {best_snr_ActiveRC:.2f} dB")
print("\nBest Gm-C configuration:")
print(f"\n\tCint: {best_config_GmC[0][0]} F\n\tRo: {best_config_GmC[1][0]} Ohm\n\t
    Cp: {best_config_GmC[2][0]} F\n\tv_n: {best_config_GmC[3][0]} V\n\t
    Slew rate: {best_config_GmC[4][0]} V/s\n\tOutput range: +-{best_config_GmC[5][0]} V\n\t
    SNR: {best_snr_GmC:.2f} dB")
```

```
Best Active-RC configuration:

        Cint: 1e-12 F
        gm: 0.0001 S
        Ro: 10000.0 Ohm
        Co: 1e-12 F
        SNR: 66.52 dB

Best Gm-C configuration:

        Cint: 1e-12 F
        Ro: 10000.0 Ohm
        Cp: 1e-12 F
        v_n: 0.001 V
        Slew rate: 0.001 V/s
        Output range: +-1.0 V
        SNR: 12.33 dB
```

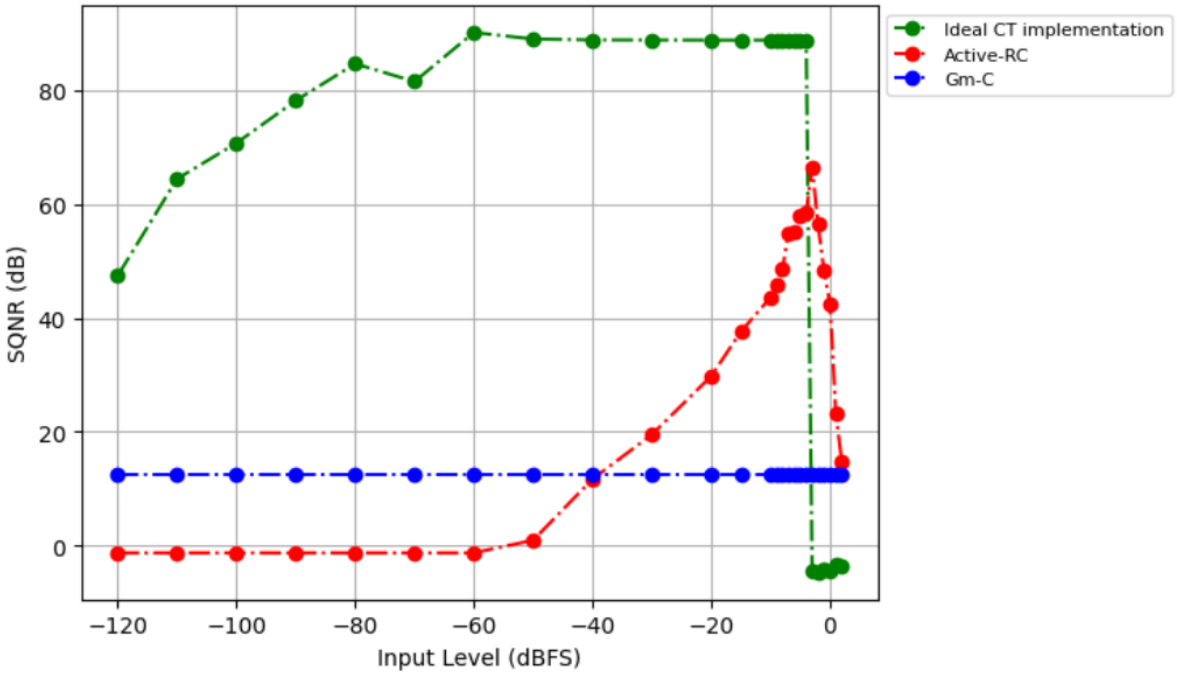
```python
# compare results and determine the best implementation:
if best_snr_ActiveRC > best_snr_GmC:
    print("Active-RC is the optimal implementation")
else:
    print("Gm-C is the optimal implementation")
```

```
Active-RC is the optimal implementation
```

```python
# plot SNR results:
plt.figure()
plt.bar(['Active-RC', 'Gm-C'], [best_snr_ActiveRC, best_snr_GmC], color=['blue', 'red'])
plt.xlabel('Implementation')
plt.ylabel('Max SNR (dB)')
plt.title('Best SNR for Active-RC vs Gm-C')
plt.grid()
plt.show()
```




```
# plot SNR results:
plt.figure()
plt.plot(amp, snr, 'o-.g', label='Ideal CT implementation')
plt.plot(amp_ActiveRC, snr_ActiveRC, 'o-.r', label='Active-RC')
plt.plot(amp_GmC, snr_GmC, 'o-.b', label='Gm-C')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.grid()
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), fontsize=8)
```



It works! :)

Of course, both implementations are much worse than the ideal continuous-time modulator, since we have chosen some values randomly. Therefore, the next step is to apply exactly the same code, but varying the parameter values for each Active-RC and Gm-C implementation.

2.6.3 Third approach

If we start trying with all values, we will find a lot of numerical errors. To solve this, we will modify a bit the code, and add a **try-except block**, so we can skip those configurations with troubles. Also, I find problems with NaN values of SNR, so I will add an if condition in the algorithm.

Finally, the solution to our problem is the one shown bellow.

```
from numpy.linalg import LinAlgError
```

```
# "optimal" modulator parameters from brute-force search:
order = 3 # modulator order
osr = 512 # oversampling ratio
nlev = 3 # number of DAC levels
f0 = 0. # input frequency (0 for baseband)
Hinf = 2.0 # feedback gain
tdac = [0, 1] # DAC sampling times
form = "FB" # modulator architecture (feedback)
dt_form = "CRFBD" # time-domain structure
Bw = 20e3 # bandwidth

# compute the sampling frequency:
fs = Bw * osr * 2
print(f"Sampling frequency: {fs / 1e6} MHz")

```

```
# rescale to fs:
dt = 1.0 / fs
```

```
Sampling frequency: 20.48 MHz
```

```
# synthesize the optimal NTF:
ntf = cb.delsig.synthesizeNTF(order, osr, 2, Hinf, f0)

# realize the NTF in state-space representation:
a, g, b, c = cb.delsig.realizeNTF(ntf, dt_form)
ABCD = cb.delsig.stuffABCD(a, g, b, c)

# create discrete-time analog frontend model:
dt_analog_frontend = cb.AnalogFrontend.dtsdm(ABCD, nlev)

# run an SNR simulation:
snr, amp, _ = dt_analog_frontend.simulateSNR(osr)
max_snr = max(snr)
print(f"SNR = {max_snr} db")
```

```
INFO:root:Simulating discrete-time analog frontend
WARNING:cbadc.digital_backend:Discrete time Wiener filter not properly implemented. Results
    may be incorrect.
SNR = 90.18351828633857 db
```

```
# transform the DT NTF into its CT equivalent:
ABCDc, tdac2 = cb.delsig.realizeNTF_ct(ntf, form, tdac)

analog_frontend = cb.AnalogFrontend.ctsdm(ABCDc, tdac2, dt, nlev)
analog_frontend.dt = 1.0 / fs # probably not necessary
```

```
# parameter search space for Active-RC implementation:
Cint_values = [1e-12, 5e-12, 1e-11, 2e-11] # integration capacitance (F)
gm_values = [0.001, 0.002, 0.005, 0.01] # transconductance (S)
Ro_values = [1e4, 5e4, 1e5, 2e5] # internal resistance (Ohm)
Co_values = [1e-12, 5e-12, 1e-11, 2e-11] # internal capacitance (F)

# store results:
best_snr_ActiveRC = -np.inf # maximum SNR found
best_config_ActiveRC = None # best parameter configuration (Cint, gm, Ro, Co)
all_results_ActiveRC = [] # all tested configurations, and their SNR
best_snr_values_ActiveRC = None # SNR values of the best configuration
best_amp_values_ActiveRC = None # amplitude values of the best configuration
```

```
# brute-force search for Active-RC:
for Cint, gm, Ro, Co in product(Cint_values, gm_values, Ro_values, Co_values):
    print("####################################")
    print(f"Testing Active-RC implementation: Cint={Cint}, gm={gm}, Ro={Ro}, Co={Co}")
    print("####################################")

    Cint = np.ones(order) * Cint
    gm = np.ones(order) * gm
    Ro = np.ones(order) * Ro
    Co = np.ones(order) * Co
```

```

    try:
        ActiveRC_analog_frontend = cb.ActiveRC(
            analog_frontend, Cint, gm, Ro, Co
        )

        snr_ActiveRC, amp_ActiveRC, _ = ActiveRC_analog_frontend.simulateSNR(osr)
        max_snr_ActiveRC = max(snr_ActiveRC)

        if not np.isnan(max_snr_ActiveRC):
            all_results_ActiveRC.append((Cint, gm, Ro, Co, max_snr_ActiveRC))

            if max_snr_ActiveRC > best_snr_ActiveRC:
                best_snr_ActiveRC = max_snr_ActiveRC
                best_config_ActiveRC = (Cint, gm, Ro, Co)
                best_snr_values_ActiveRC = snr_ActiveRC
                best_amp_values_ActiveRC = amp_ActiveRC
                # print(snr_GmC, amp_GmC) # show the best configuration until now, to see how
                    # to vary parameters
        else:
            print(f"Skipping NaN result for Cint={Cint}, gm={gm}, Ro={Ro}, Co={Co}")

    except (LinAlgError, ValueError) as e:
        print(f"Error encountered with Cint={Cint}, gm={gm}, Ro={Ro}, Co={Co}. Skipping this
            configuration")
```

```
INFO:root:Simulating continuous-time analog frontend for sinusoidal input
###################################
Testing Active-RC implementation: Cint=1e-12, gm=0.001, Ro=10000.0, Co=1e-12
###################################
INFO:root:Simulating continuous-time analog frontend for sinusoidal input
###################################
Testing Active-RC implementation: Cint=1e-12, gm=0.001, Ro=10000.0, Co=5e-12
###################################
...
INFO:root:Simulating continuous-time analog frontend for sinusoidal input
Skipping NaN result for Cint=[1.e-12 1.e-12 1.e-12], gm=[0.001 0.001 0.001],
    Ro=[100000. 100000. 100000.], Co=[1.e-12 1.e-12 1.e-12]
###################################
Testing Active-RC implementation: Cint=1e-12, gm=0.001, Ro=100000.0, Co=5e-12
###################################
...
```

```
# parameter search space for Gm-C implementation:
Cint_values = [1e-11, 2e-11] # integration capacitance (F)
Ro_values = [10e3, 20e3] # output resistance (Ohm)
Cp_values = [1e-11, 1e-10] # parasitic capacitance (F)
v_n_values = [1e-3, 5e-3] # input noise density (V rms)
slew_rate_values = [5e-4, 5e-5]  # slew rate (V/s)
output_swing_values = [1.0] # output voltage swing (V)

# store results:
best_snr_GmC = -np.inf # maximum SNR found
best_config_GmC = None # best parameter configuration (Cint, Ro, Cp, v_n, slew_rate, v_out_max)
all_results_GmC = [] # all tested configurations, and their SNR
```

```
best_snr_values_GmC = None # SNR values of the best configuration
best_amp_values_GmC = None # amplitude values of the best configuration
```

```
# brute-force search for Gm-C:
for Cint, Ro, Cp, v_n, slew_rate, output_swing in product(Cint_values, Ro_values, Cp_values,
        v_n_values, slew_rate_values, output_swing_values):
    print("####################################")
    print(f"Testing Gm-C implementation: Cint={Cint}, Ro={Ro}, Cp={Cp}, v_n={v_n},
        slew_rate={slew_rate}, output_swing={output_swing}")
    print("####################################")

    Cint = np.ones(order) * Cint
    Ro = np.ones(order) * Ro
    Cp = np.ones(order) * Cp
    v_n = np.ones(order) * v_n
    slew_rate = np.ones(order) * slew_rate
    v_out_max = np.ones(order) * output_swing
    v_out_min = -v_out_max

    try:
        GmC_analog_frontend = cb.GmC(
            analog_frontend, Cint, Ro, Cp, v_n, v_out_max, v_out_min, slew_rate
        )

        snr_GmC, amp_GmC, _ = GmC_analog_frontend.simulateSNR(osr)
        max_snr_GmC = max(snr_GmC)

        if not np.isnan(max_snr_GmC):
            all_results_GmC.append((Cint, Ro, Cp, v_n, slew_rate, output_swing, max_snr_GmC))

            if max_snr_GmC > best_snr_GmC:
                best_snr_GmC = max_snr_GmC
                best_config_GmC = (Cint, Ro, Cp, v_n, slew_rate, v_out_max)
                best_snr_values_GmC = snr_GmC
                best_amp_values_GmC = amp_GmC
                # print(snr_GmC, amp_GmC) # show the best configuration until now, to see how
                    # to vary parameters
        else:
            print(f"Skipping NaN result for Cint={Cint}, Ro={Ro}, Cp={Cp}, v_n={v_n},
                slew_rate={slew_rate}, output_swing={output_swing}")

    except (LinAlgError, ValueError) as e:
        print(f"Error encountered with Cint={Cint}, Ro={Ro}, Cp={Cp}, v_n={v_n},
            slew_rate={slew_rate}, output_swing={output_swing}. Skipping this configuration")
```

```
INFO:root:Simulating continuous-time analog frontend for sinusoidal input
####################################
Testing Gm-C implementation: Cint=1e-11, Ro=10000.0, Cp=1e-11, v_n=0.001, slew_rate=0.0005,
    output_swing=1.0
####################################
INFO:root:Simulating continuous-time analog frontend for sinusoidal input
####################################
Testing Gm-C implementation: Cint=1e-11, Ro=10000.0, Cp=1e-11, v_n=0.001, slew_rate=5e-05,
    output_swing=1.0
####################################
...
```

```python
# display best configurations:
print("Best Active-RC configuration:")
print(f"\n\tCint: {best_config_ActiveRC[0][0]} F\n\tgm: {best_config_ActiveRC[1][0]} S\n\t
    Ro: {best_config_ActiveRC[2][0]} Ohm\n\tCo: {best_config_ActiveRC[3][0]} F\n\t
    SNR: {best_snr_ActiveRC:.2f} dB")
print("\nBest Gm-C configuration:")
print(f"\n\tCint: {best_config_GmC[0][0]} F\n\tRo: {best_config_GmC[1][0]} Ohm\n\t
    Cp: {best_config_GmC[2][0]} F\n\tv_n: {best_config_GmC[3][0]} V\n\t
    Slew rate: {best_config_GmC[4][0]} V/s\n\tOutput range: +-{best_config_GmC[5][0]} V\n\t
    SNR: {best_snr_GmC:.2f} dB")
```

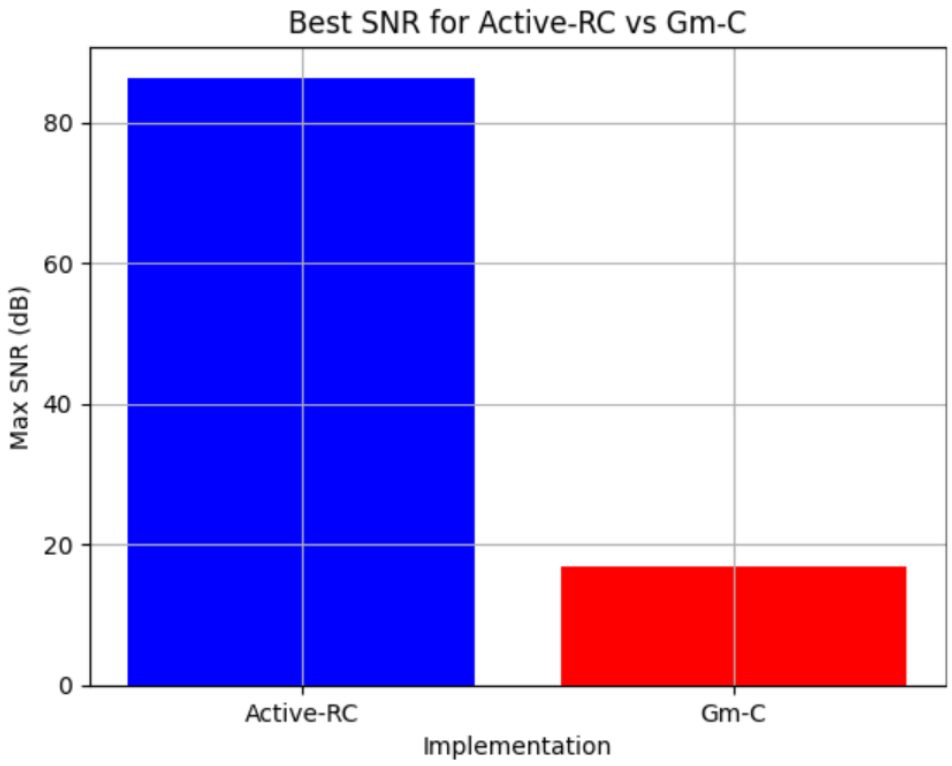
```
Best Active-RC configuration:

        Cint: 1e-12 F
        gm: 0.001 S
        Ro: 100000.0 Ohm
        Co: 1e-11 F
        SNR: 86.34 dB

Best Gm-C configuration:

        Cint: 2e-11 F
        Ro: 20000.0 Ohm
        Cp: 1e-10 F
        v_n: 0.001 V
        Slew rate: 0.0005 V/s
        Output range: +-1.0 V
        SNR: 16.92 dB
```

I have tried several configurations for Gm-C implementations and I can not achieve a better SNR.

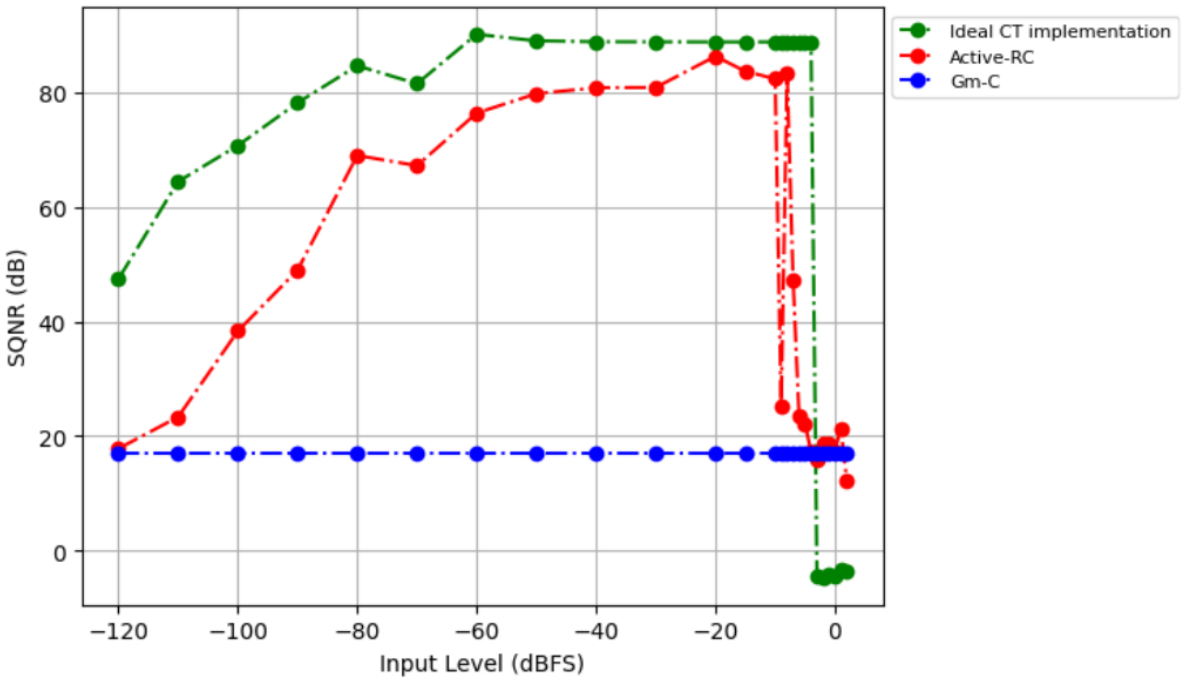
```python
# compare results and determine the best implementation:
if best_snr_ActiveRC > best_snr_GmC:
    print("Active-RC is the optimal implementation")
else:
    print("Gm-C is the optimal implementation")
```

```
Active-RC is the optimal implementation
```

```python
# plot SNR results:
plt.figure()
plt.bar(['Active-RC', 'Gm-C'], [best_snr_ActiveRC, best_snr_GmC], color=['blue', 'red'])
plt.xlabel('Implementation')
plt.ylabel('Max SNR (dB)')
plt.title('Best SNR for Active-RC vs Gm-C')
plt.grid()
plt.show()
```




```
# plot SNR results:
plt.figure()
plt.plot(amp, snr, 'o-.g', label='Ideal CT implementation')
plt.plot(best_amp_values_ActiveRC, best_snr_values_ActiveRC, 'o-.r', label='Active-RC')
plt.plot(best_amp_values_GmC, best_snr_values_GmC, 'o-.b', label='Gm-C')
plt.xlabel('Input Level (dBFS)')
plt.ylabel('SQNR (dB)')
plt.grid()
plt.legend(loc='upper left', bbox_to_anchor=(1, 1), fontsize=8)
```



The Active-RC implementation closely follows the ideal implementation up to around 85 dB of SQNR, indicating that it can efficiently approximate the ideal modulator behavior. On the other hand, the Gm-C implementation shows significantly worse performance, with SQNR remaining at approximately 16 dB across all input levels.

In conclusion, the best implementation achieved is:

Active-RC:

- **Integration capacitance**: $C_{\text{int}} = 10^{-12}\text{F}$
- **Transconductance**: $g_m = 0.001\text{S}$
- **Internal resistance**: $\rho = 10^5\Omega$
- **Internal capacitance**: $C_o = 10^{-11}\text{F}$
- **Signal-to-Noise Ratio**: $SNR = 86.34\text{dB}$

Finally, we couldn't reach the target SNR, but this implementation is not too far from it. Maybe, if we expand the parameters search-space with more values, we would be able to find a better solution, but computing time increase exponentially due to the kind of algorithm.

In any case, there must be errors, because in Gm-C implementation, the SNR keeps constant in every configuration. To see that, we can show the output in brute-force search, and also in the final plot. I have been trying to solve this mistake, but I couldn't get a solution :(

3 Conclusions and recommendations

Date: Febrary 19, 2025

In this final section, I would like to make some conclusions of this work, and give some recommendations.

One of the key takeaways from this work is that working directly with the repository and **running the code oneself is essential**. Merely reviewing the existing implementations is not enough, as executing the functions with different parameters often reveals **unexpected issues**. Many errors and inconsistencies arise when modifying variables or using different setups, highlighting the importance of hands-on experimentation.

During this study, numerous challenges were encountered when attempting to run the algorithms correctly. Each attempt often led to new, distinct errors, requiring incremental fixes to ensure proper execution. While many of these issues were patched to make the code functional, it is likely that **more underlying problems remain undetected**. Therefore, further testing and debugging are recommended for anyone working with this repository.

To improve robustness and reduce potential runtime failures, I strongly recommend to **integrate try-catch blocks into the code**. These would help handle exceptions more gracefully, particularly when dealing with invalid configurations or unexpected parameter values. Implementing structured error handling will make debugging more efficient and improve overall code stability.

Additionally, the use of brute-force search for optimization in this report has proven to be **inefficient**. While this method served as a means to familiarize oneself with the environment and algorithms, it is far from an optimal solution. Future work should explore more sophisticated optimization techniques to improve efficiency and reduce computational time.

Given the challenges encountered and the insights gained from working with this repository, several potential **improvements** can be proposed for future work:

- As previously mentioned, incorporating **more robust error-handling mechanisms**, such as structured exception handling (try-except blocks), would significantly improve code stability.
- The results obtained during this study were not as good as expected, mainly due to the errors found in the Gm-C implementations. Future work should focus on identifying the root causes of these issues, **debugging the transconductance-based designs**, and ensuring that the circuit behavior matches theoretical expectations.
- Explore more **advanced optimization techniques**, such as genetic algorithms, gradient-based optimization, or neural networks.
- Adding support for **more $\Sigma\Delta$M architectures**.

Finally, **any feedback, contributions, or suggestions from others working on similar implementations are highly encouraged :)**. Collaboration can help identify additional errors, propose alternative approaches, and enhance the overall functionality of the repository.