EE 322: EE Laboratory II
Final Portfolio

Steven Placzek

Department of Electrical and Computer Engineering
College of Engineering
Western New England University

May 8, 2025

Abstract

This portfolio documents a series of laboratory experiments conducted as part of EE322, with the objective of exploring and analyzing various MOSFET-based analog circuit topologies. The labs covered topics ranging from NMOS characterization and biasing techniques to the design and analysis of amplifiers and differential pairs. Experimental setups were implemented using the ALD1105/1106 transistor arrays and verified through simulations in LTSpice and Python-based data analysis. Circuit analyses techniques contained herein include the extraction of small-signal parameters, understanding the influence of source and load resistances on gain, and the role of feedback in stabilizing amplifier performance. Special emphasis was placed on evaluating frequency response, common-mode rejection, and the impact of grounding practices on measurement integrity. The final experiments introduced advanced concepts like S-parameter de-embedding using a Vector Network Analyzer (VNA), bridging analog design with RF characterization techniques. This comprehensive investigation demonstrates the critical interplay between theory, simulation, and practical implementation in analog circuit design.

Contents

List of Figures

List of Tables

Chapter 1

Lab 1 — NMOS Characterization

This experiment aims to characterize an NMOS transistor for its relevant parameters, including threshold voltage (V_{TH}), transconductance parameter (K), and early voltage (V_A). The laboratory assignment utilizes Python to automate the measurement procedure and data acquisition. The ALD1106 is the device tested for this experiment.

1.1 Background

Parameters V_{TH}, K, and V_A are necessary parameters to determine both transistor bias points and small signal parameters. These parameters appear when examining the voltage-current characteristics of the NMOS transistor as follows:

- Cutoff Mode: $I_D = 0$ when $V_{GS} < V_{Tn}$ or $V_{DS} = 0$
- Triode Mode: $I_D = k_n^{'}\left[\frac{W}{L}\right]\left(V_{OVn} - \frac{1}{2}V_{DS}\right)V_{DS}$ when $V_{GS} > V_{Tn}$ & $V_{DS} < V_{OVn}$
- Saturation Mode: $I_D = K_N V_{OVn}^2\left(1 + \frac{V_{DS}}{|V_A|}\right)$ when $V_{GS} > V_{Tn}$ & $V_{DS} \geq V_{OVn}$

Typically in saturation mode, the $1+V_{DS}/|V_A|$ term is ignored when performing the DC analysis of a transistor circuit. However, in this lab experiment it is a critical parameter to ensure that the appropriate approximations can be made to linearize data for parameter extraction.

1.2 Experiment 1: I_D vs. V_{DS}

In this experiment, the I_D vs. V_{DS} curves will be generated by using the Rohde Schwarz NGU401 source measure unit (SMU) and the Keithley 3-channel programmable power supply. In this experiment, the SMU will simultaneously source the V_{DS} voltage and measure the I_D current in real time. The voltage V_{GS} is generated by the programmable power supply as a

family of curves is desired. A circuit schematic and wiring diagram are shown in Figure 1.1. The Python code list is under the Coding List in the Appendix.

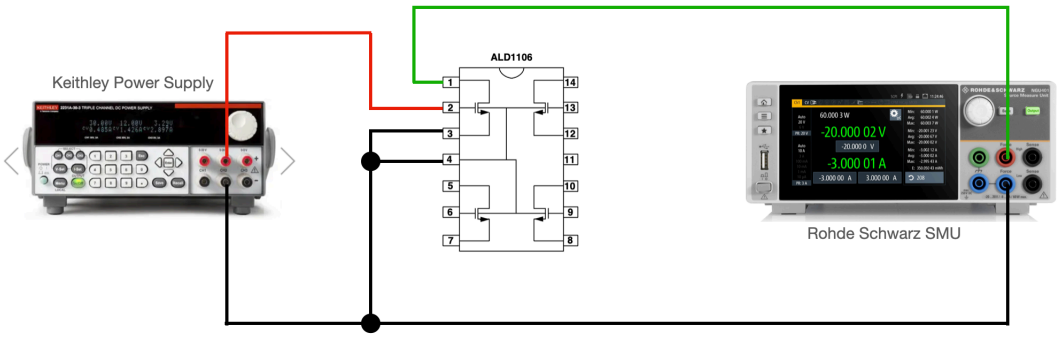


Figure 1.1: Experiment 1 wiring diagram interfacing the transistor with the SMU and 3 channel power supply

1.3 Experiment 2: I_D vs. V_{GS}

In this experiment, the I_D vs. V_{GS} curves will be generated by using the Rohde Schwarz NGU401 source measure unit (SMU), the Keithley 3-channel programmable power supply and the Keithley digital multimeter (DMM). In this experiment, the SMU will source the V_{GS} voltage. The voltage V_{DS} is generated by the programmable power supply and the current I_D is measured in real time by the Keithley DMM to generate the family of curves as desired. The circuit schematic remains the same; however, different voltages are being swept, therefore, the wiring diagram is modified as shown in Figure 1.2. The Python code list is under the Coding List Listing A.2, in the Appendix.

1.4 Results

The post-laboratory tasks for this assignment included plotting the data from experiments 1 and 2. These plots include I_D vs. V_{DS} for a family of V_{GS} values along with plotting I_D vs. V_{GS} for a family of V_{DS} values. The transistor parameters V_{TH}, K, and V_A are extracted by curving fitting the data from experiments 1 and 2. Code listing provides a detailed approach for the extraction of data parameters. Figure 1.3 displays the summarized results of the experiments, and the data parameters are listed in Tables 1.1 and 1.2. listing A.3

1.5 Conclusions

The data results indicate the NMOS transistor has a threshold voltage $V_{TH} = 0.547$, $K_N = 265.96$ $\mu\ A/V$, and Early voltages of -62.75, -88.63, and -140.18. The linearized region used to

0.9

Figure 1.2: Experiment 2 wiring diagram interfacing the transistor with the SMU, 3 channel power supply, and digital multimeter

extract V_{TH} & K_N had a coefficient of determination fit of $R^2 = 0.9988$ representing a near ideal fit across multiple data points with a small standard deviation among the various values of V_{DS}. The data points are chosen to minimize short channel effects from the measured results. In addition, it is experimentally observed that the Early voltage for each V_{GS} sweep does not intersect the x-axis at the same point, indicating that the channel length modulation parameter is dependent on bias conditions. Therefore, when modeling this phenomenon, it likely requires a mathematical fit to fully caption its effects.

Quantity	Mean Value	Standard Deviation	Units
K_N	265.96	5.83	μA/V
V_{Tn}	0.547	0.007	V

Table 1.1: Summary of $\sqrt{I_D}$ vs. V_{GS} extracted parameter values

Quantity	Value	Units
V_A ($V_{GS} = 2$V)	-62.74	V
V_A ($V_{GS} = 4$V)	-88.63	V
V_A ($V_{GS} = 6$V)	-140.18	V

Table 1.2: Summary of I_D vs. V_{DS} extracted parameter values

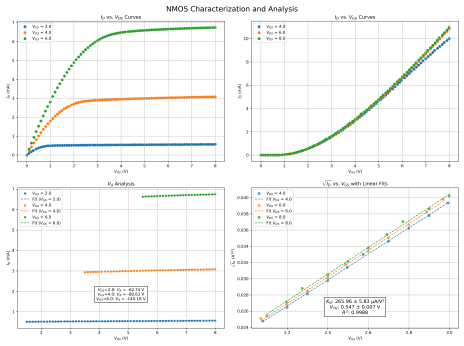


Figure 1.3: Summary of data for the NMOS characterization laboratory assignment

Chapter 2

Lab 2 — IC Biasing Techniques

2.1 Lab Assignment Goals

The goal of this lab was to study and compare the MOSFET-resistor biasing circuit and the beta multiplier circuit, specifically focusing on performance metrics such as power supply sensitivity. Biasing is essential in integrated circuits to establish proper operating points for transistors, ensuring consistent performance and linearity despite variations in temperature, power supply, or transistor parameters. The beta multiplier circuit is often preferred over a simple MOSFET-resistor configuration due to its ability to provide a more stable and predictable bias current, making it less sensitive to process and temperature variations. This stability is achieved through a feedback loop and multiple transistors that generate a reference current capable of adapting to changes, improving control and reliability in precision applications.

To accomplish this, both circuits were simulated in LTSpice and built on a breadboard using the ALD1106/ALD1105 transistor array. Simulations and experimental measurements were conducted to analyze circuit behavior under different conditions, particularly variations in supply voltage. Data from both approaches were compared to evaluate the performance differences between the two biasing methods and to assess the advantages of the beta multiplier over the MOSFET-resistor configuration.

2.2 Experiment 1: The MOSFET-Resistor Characterization

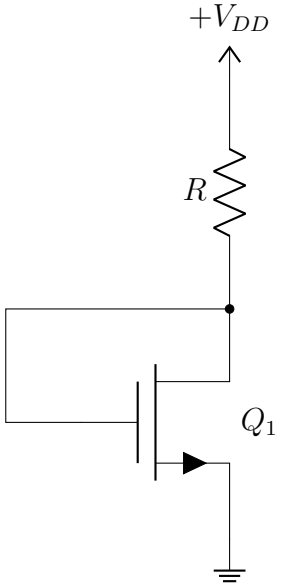
The objectives of this experiment involved characterizing a MOSFET-resistor circuit, as illustrated in the schematic below. The following tasks were performed:

- The resistor was replaced with a 20kΩ potentiometer while setting $V_{DD} = 10$V. The resistance R was determined at which $I_D = 500\mu$A.
- With the current set to $I_D = 500\mu$A, the DC operating point was identified.

0.5

Figure 2.1: The pinout and block diagram of the ALD1106 (top) and ALD1105 (bottom), **Note: V^- is the body of the NMOS devices which is connected to lowest potential and V^+ is the body of the PMOS devices which is connected to the highest potential.**

- Finally, the supply voltage V_{DD} was swept from 0 to 15V.



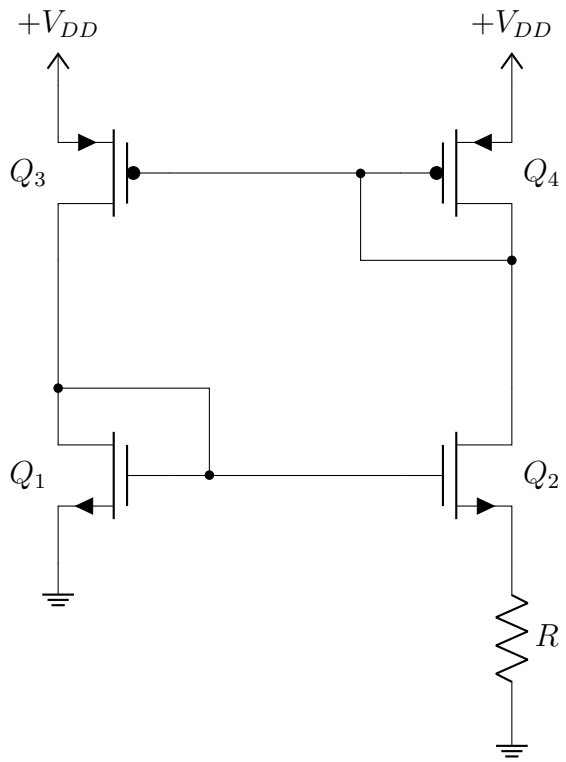
0.475

Figure 2.2: Wiring diagram of the MOSFET-resistor biasing circuit. The potentiometer had three terminals, with the top and bottom terminals used in combination with the middle terminal.

2.3 Experiment 2: The Beta Multiplier Characterization

The objectives of this experiment involved characterizing a beta multiplier circuit, as illustrated in the schematic below. The following tasks were performed. Additionally, the width of Q_2 was four times that of Q_1, achieved by wiring four NMOS devices in parallel as available on the ALD1106:

- The resistor was replaced with a 5kΩ potentiometer while setting $V_{DD} = 10$V. The resistance R was determined at which $I_{D2} = 500\mu$A.
- With the current set to $I_{D2} = 500\mu$A, the DC operating point was identified.
- Finally, the supply voltage V_{DD} was swept from 0 to 15V.
- Python Code used for this experiment can be found under the code listings chapter B, chapter B, & chapter B



0.475

Figure 2.3: Wiring diagram of the MOSFET-resistor biasing circuit. The potentiometer had three terminals, with the top and bottom terminals used in combination with the middle terminal.

2.4 Post Lab Data Analysis

The post-lab analysis involved the following tasks:

- The circuits from Experiment 1 and Experiment 2 were simulated, following the same procedure of sweeping R to determine the bias point at $I_D = I_{D2} = 500\mu$A. The DC operating point was then identified, followed by sweeping the power supply V_{DD} from 0 to 15V.
- The tables for simulated and measured data were completed as specified.
- Both the simulated and measured power supply voltage sweeps were plotted on the same curve. The derivative of the simulated data was taken to determine the supply voltage sensitivity at the operating point $V_{DD} = 10$V.
- The results from Experiment 1 and Experiment 2 were compared to determine which circuit exhibited greater resistance to dynamic changes in the power supply.

2.4.1 Experiment 1 - Results:

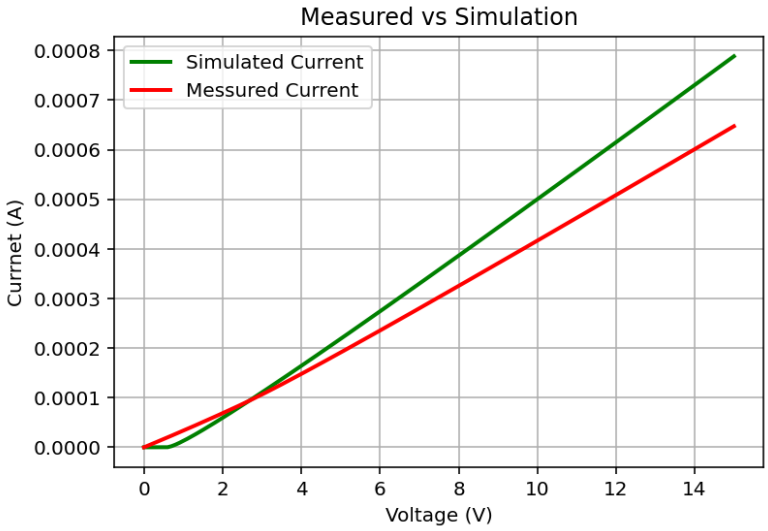


Figure 2.4: Experiment 1: Simulated vs Measured Plot

Quantity	Simulated	Measured	Units
R	16.200	16.003	kΩ
V_{GS1}	1.91	1.940	V
V_{DS1}	1.91	1.940	V
V_{OV1}	1.337	1.341	V
I_{D1}	499.999	504.477	μA

Table 2.1: Experiment 1 Data Summary

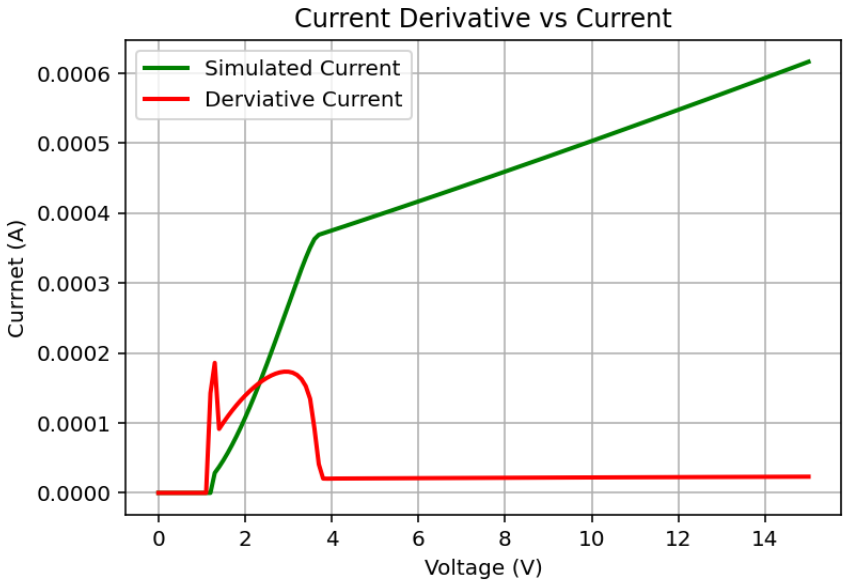


Figure 2.5: Experiment 2: Simulated I_D vs $\frac{d}{dt}(I_D)$ Plot

2.4.2 Experiment 2 - Simulated Results:

The post-lab analysis involved comparing two circuit configurations, with the data clearly demonstrating that the circuit from Experiment 2 exhibits superior resistance to sudden changes in the power supply voltage. This enhanced stability can be attributed to the beta mirror configuration, which employs a cascade of current mirrors to maintain consistent current flow. The multiple transistor stages in the beta mirror create a high-impedance path that effectively isolates the bias current from power supply fluctuations. Additionally, the negative feedback inherent in the mirror configuration helps compensate for any variations in supply voltage by adjusting the gate-source voltages of the transistors to maintain constant current.

Quantity	Simulated	Measured	Units
R	1.254	1.0361	kΩ
V_{GS1}	1.980	1.985	V
V_{DS1}	1.980	1.985	V
V_{OV1}	1.407	1.424	V
I_{D1}	552.000	506.814	μA
V_{GS2}	1.350	1.496	V
V_{DS2}	6.420	6.646	V
V_{OV2}	0.644	0.935	V
I_{D2}	500.000	506.814	μA
V_{SG3}	2.960	2.861	V
V_{SD3}	8.020	8.006	V
$\lvert V_{OV3}\rvert$	2.313	2.399	V
I_{D3}	552.000	506.814	μA
V_{SG4}	2.960	2.862	V
$\lvert V_{OV4}\rvert$	2.313	2.301	V
I_{D4}	500.000	506.814	μA

Table 2.2: Experiment 2 Data Summary

2.4.3 Experiment 2 - Simulated vs Measured Results:

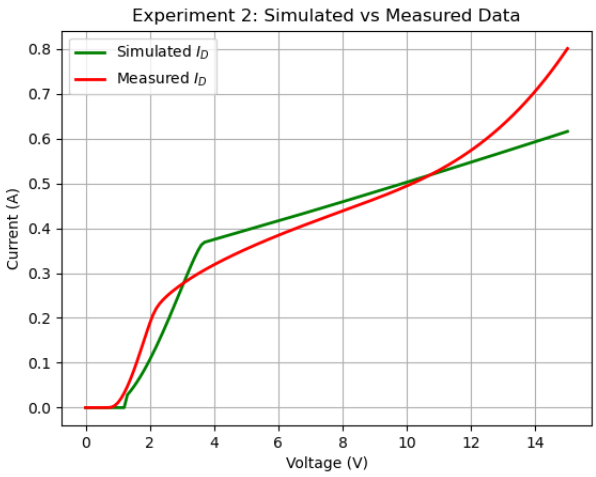


Figure 2.6: Experiment 2: Simulated vs Measured Plots

2.5 Code Listings

The Python implementations for the biasing analysis can be found in the Appendix: Plot Measured vs Simulated Data (listing B.1), Extract OP Values from LTSpice (listing B.2), Plot Measured vs Simulated Current (listing B.3), Calculate MOSFET Parameters (listing B.4), and Simulated vs Measured Data Plot (listing B.5).

2.6 Conclusion

This lab effectively demonstrated the differences in performance and robustness between a basic MOSFET-resistor biasing circuit and a beta multiplier biasing configuration. Through both simulation and physical prototyping, it was observed that while the MOSFET-resistor circuit can achieve the desired operating point with manual tuning, it suffers from significant sensitivity to changes in supply voltage. In contrast, the beta multiplier circuit consistently maintained a stable bias current across varying supply voltages, showcasing superior power supply rejection characteristics.

The enhanced stability of the beta multiplier can be attributed to its negative feedback mechanism and the use of matched transistor pairs, which allow the circuit to self-correct and adapt to fluctuations in external conditions. This makes it a preferred choice in precision analog and mixed-signal applications where stability and predictability are crucial.

Simulation results closely matched experimental data for both circuits, validating the modeling approach in LTSpice. Additionally, the derivative analysis confirmed that the beta multiplier exhibited a flatter response in current with respect to changes in V_{DD}, quantitatively proving its reduced supply voltage sensitivity.

Overall, results highlight the importance of choosing appropriate biasing strategies in analog design and illustrated the trade-off between simplicity and performance in circuit design.

Chapter 3

Lab 3 — Actively Loaded Common Source Amplifier

3.1 Lab Assignment Goals

This objective of this lab assignment was to methodically evaluate a common source amplifier built with the ALD1105 MOSFET, with the overarching objective of determining how variations in load resistance and signal generator resistance affect circuit performance. The group's tasks were organized into a series of experiments, each aimed at addressing distinct aspects of amplifier behavior. In the first experiment, the DC operating point was established by adjusting the DC bias voltage V_G until the output voltage V_O reached 5V. This step involved measuring node voltages and calculating the corresponding drain currents, providing a baseline for subsequent analysis. Establishing an accurate operating point is critical to ensure that the MOSFET operates in the intended region, which is a fundamental concept that can sometimes be overlooked in traditional electrical engineering coursework. This experiment was intended to highlight the impact of how very slight adjustments to load resistance, along with signal generator resistance, will result in large changes with the output impedance and thus have a larger affect upon the overall stability of the circuit.

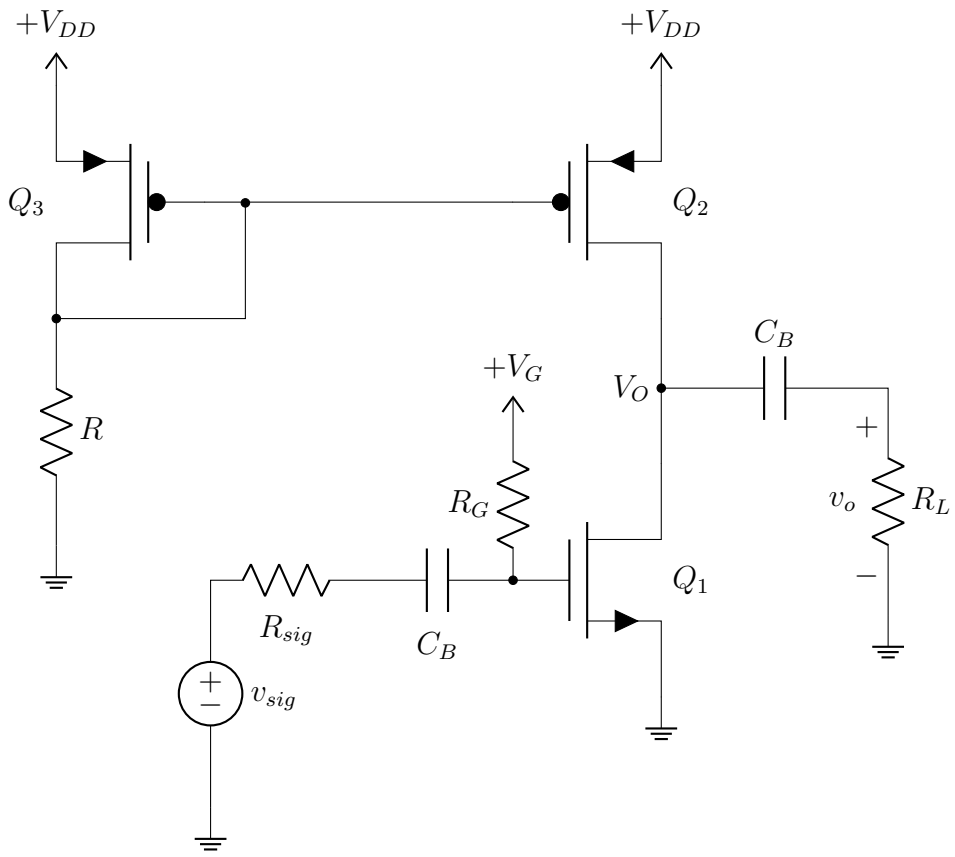
0.5

Figure 3.1: The pinout and block diagram of the ALD1105, **Note: V^- is the body of the NMOS devices which is connected to lowest potential and V^+ is the body of the PMOS devices which is connected to the highest potential.**

3.2 Experiment 1: DC Operating Point

Using the circuit shown below with the parameter values given, adjust the DC voltage V_{GG} such that the output voltage V_O = 5 V

- Find the dc operating point by measuring the node voltages and calculating the currents. Complete the summary table.



$$V_{DD} = 10\ \text{V} \qquad V_O = 5\ \text{V} \qquad C_B = 0.1\mu\ \text{F}$$

$$R = 15\text{k}\ \Omega \qquad R_G = 10\text{M}\ \Omega$$

$$K_N = 270\ \ \mu\text{A/V}^2 \quad V_{T_n} = 0.573\ \text{V} \quad \lambda_N = 0.0165\text{V}^{-1}$$

$$K_P = 88\ \mu\text{A/V}^2 \ \ V_{Tp} = -0.647\ \text{V} \quad \lambda_P = 0.0219\ \text{V}^{-1}$$

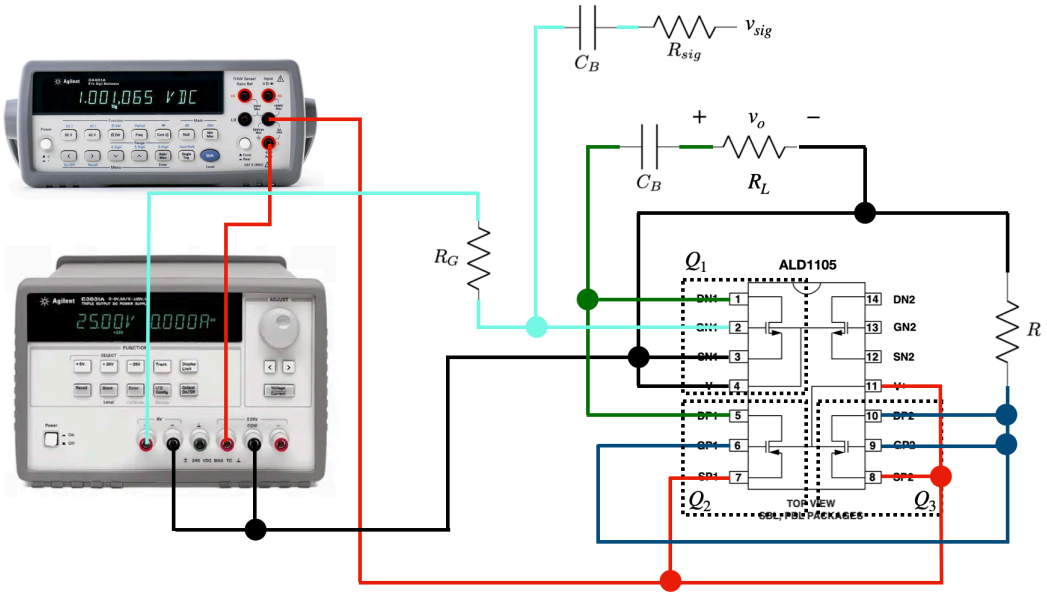


Figure 3.2: *Wiring diagram of the Common Source Amplifier.*

3.2.1 Measurement Issues & 60 Hz Noise

The physical measurement data were compromised due to improper grounding practices. In this instance, the use of separate ground rails resulted in ground loops that injected considerable 60 Hz interference into the circuit measurements. This phenomenon is particularly problematic with the ALD1105 MOSFET, as its performance is highly sensitive to noise in the biasing network. Wiring the transistor to multiple different rails will result in ground loops that introduce significant 60 Hz interference into measurement data, which this experiment demonstrated quite clearly. [1]

In this scenario, the use of separate ground rails for the DC power supply, circuit, oscilloscope probes, and function generator led to mismatched ground potentials. According to an IEEE article on mitigating noise in electronic measurement systems, improper grounding practices can lead to the formation of ground loops that introduce significant $60Hz$ interference into measurement data. In this case, using separate ground rails for the DC power supply, circuit, oscilloscope probes, and function generator resulted in mismatched ground potentials. This is particularly problematic when working with the ALD1105 MOSFET, which is highly sensitive to noise in its biasing network and operating conditions. [2]

3.3 Experiment 2: Investigating the Role of R_{sig}

After establishing the DC operating point in Experiment 1, the role of the signal generator resistance, R_{sig}, on the overall AC small-signal gain, G_v, was investigated. The gain was defined as:

$$G_v = \frac{v_o}{v_{sig}}$$

The following procedure was implemented:

- The load resistance was set to $R_L = 10\,\text{M}\Omega$ (as provided by the oscilloscope probe).
- The small-signal gain, G_v, was measured for various values of R_{sig}, specifically for:

$$R_{sig} = 10\,\text{M}\Omega$$

$$R_{sig} = 100\,\text{k}\Omega$$

$$R_{sig} = 10\,\text{k}\Omega$$

$$R_{sig} = 1\,\text{k}\Omega$$

3.4 Experiment 3: Investigating the Role of R_L

Following the establishment of the DC operating point in Experiment 1, the influence of the load resistance, R_L, on the AC small-signal gain, G_v, was examined. The gain was defined as:

$$G_v = \frac{v_o}{v_{sig}}$$

The following procedure was executed:

- The signal generator resistance was fixed at $R_{sig} = 1\,\text{k}\Omega$.
- The small-signal gain, G_v, was measured for various values of R_L, specifically:

$$R_L = 10\,\text{M}\Omega$$

$$R_L = 100\,\text{k}\Omega$$

$$R_L = 10\,\text{k}\Omega$$

$$R_L = 1\,\text{k}\Omega$$

3.5 Post Lab Data Analysis

The post-lab analysis consisted of the following tasks:

- Simulate the circuits from experiments 1, 2, and 3 following the same procedure of sweep-

ing V_{GG} to find the bias point where $V_O = 5$V for experiment 1. Then operating under the bias condition, perform an AC analysis to find the ac small signal gain G_V under the various R_{sig} and R_L conditions specified in experiments 2 and 3.

- The completed simulated data been entered into the tables, (*the missing measurements will completed before the next lab meeting*), as specified below:

Experiment 1 Summary Table				
Device	**Quantity**	**Simulated**	**Measured**	**Units**
Q_1	I_D	494.0	485.000	μA
	$\lvert V_{OV}\rvert$	1.299	1.290	V
	V_G	1.872	1.860	V
	V_D	5.058	4.970	V
	V_S	0.000	0.000	V
Q_2	I_D	494.0	-480.000	μA
	$\lvert V_{OV}\rvert$	7.481	2.310	V
	V_G	1.872	7.030	V
	V_D	5.058	4.970	V
	V_S	10.000	9.990	V
Q_3	I_D	494.0	-480.009	μA
	$\lvert V_{OV}\rvert$	2.250	2.310	V
	V_G	7.103	7.040	V
	V_D	7.103	7.040	V
	V_S	10.000	9.99	V

Table 3.1: Experiment 1 Summary Table

AC Small Signal Gain $R_L = 10\,\mathbf{M\Omega}$				
Condition	Quantity	Simulated	Measured	Units
$R_{sig} = 10\,\mathrm{M\Omega}$	G_v	-21.900	7.6110	V/V
	G_v	26.810	17.6288	dB
$R_{sig} = 100\,\mathrm{k\Omega}$	G_v	-43.365	16.2100	V/V
	G_v	32.743	24.1957	dB
$R_{sig} = 10\,\mathrm{k\Omega}$	G_v	-43.756	20.1700	V/V
	G_v	32.821	26.0941	dB
$R_{sig} = 1\,\mathrm{k\Omega}$	G_v	-43.795	22.3100	V/V
	G_v	32.829	26.9700	dB

Table 3.2: Experiment 2 Summary Table

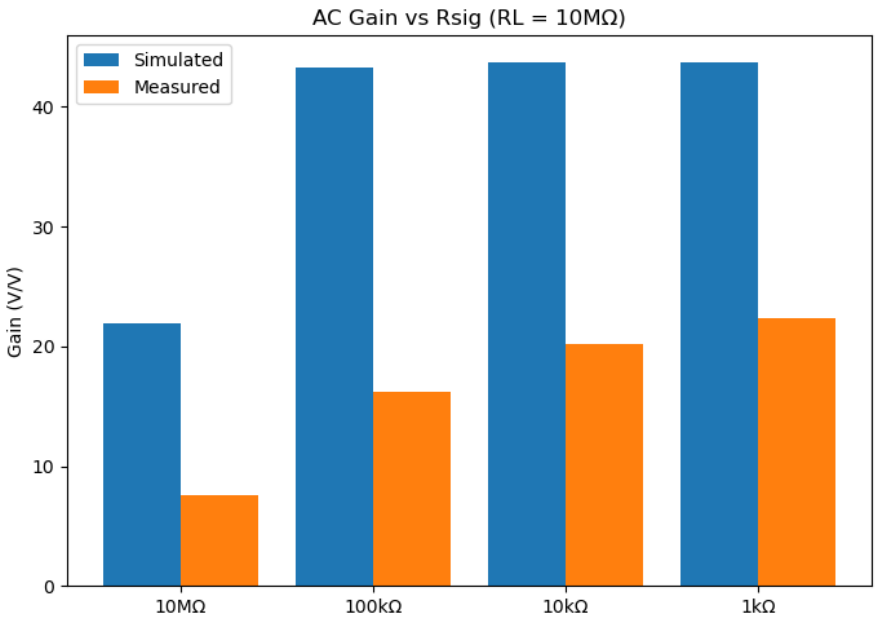


Figure 3.3: Simulated vs Measured data for $R_L = 10$ M

AC Small Signal Gain $R_{sig} = 1\,\mathbf{k\Omega}$				
Condition	Quantity	Simulated	Measured	Units
$R_L = 10\,\mathrm{M\Omega}$	G_v	-21.900	31.2000	V/V
	G_v	26.810	20.881	dB
$R_L = 100\,\mathrm{k\Omega}$	G_v	-13.943	23.5000	V/V
	G_v	22.887	27.4214	dB
$R_L = 10\,\mathrm{k\Omega}$	G_v	-3.240	3.6500	V/V
	G_v	10.211	11.2459	dB
$R_L = 1\,\mathrm{k\Omega}$	G_v	-0.373	0.9510	V/V
	G_v	-8.555	-0.4364	dB

Table 3.3: Experiment 3 Summary Table

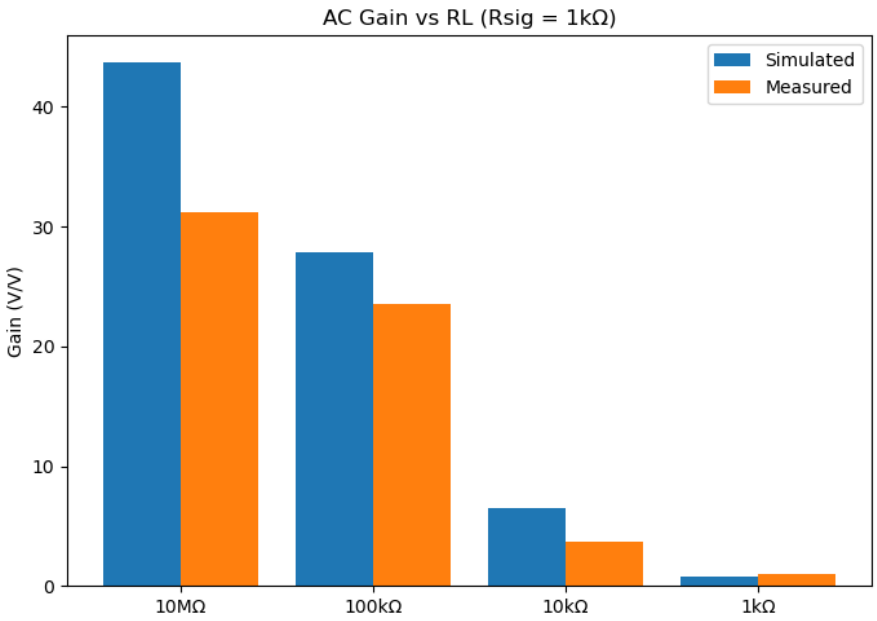


Figure 3.4: Simulated vs Measured data for $R_{\text{sig}} = 1$ k

3.6 Code Listings

The Python code used for the common source amplifier analysis can be found in the Appendix: Common Source Amp LTSpice Sweep (listing C.1).

3.7 Conclusion

This lab successfully demonstrated the critical influence of source and load resistances on the performance of a common source amplifier using the ALD1105 MOSFET. Through a series of controlled experiments, both simulated and measured, the impact of varying R_{sig} and R_L on the AC small-signal gain G_V was thoroughly evaluated.

In Experiment 1, the DC operating point was accurately established by adjusting V_O such that the output voltage V_O was 5 V. This step confirmed the theoretical behavior of MOSFETs in the saturation region and emphasized the importance of correct biasing for reliable amplifier operation. The lack of close agreement between simulated and measured data reinforced the inaccuracy of the transistor model.

Experiment 2 showed that as R_{sig} decreased, the voltage gain G_v increased significantly, particularly in the measured results. This outcome is consistent with theoretical expectations, as a lower source resistance reduces signal attenuation before the gate of the MOSFET, allowing a greater portion of the input signal to modulate the channel current.

Experiment 3 revealed a strong dependence of gain on the load resistance R_L. Higher values of R_L led to markedly high gains, again matching theoretical predictions based on the amplifier's gain formula $G_v \approx -g_m R_D \parallel R_L$. Results emphasized that loading effects from external measurement equipment (e.g., oscilloscope probes) or downstream circuitry can significantly influence observed performance.

The major insight from the lab was effects of grounding practices on measurement quality. Ground loops introduced substantial 60 Hz noise, initially preventing accurate gain readings. Once corrected, the improved grounding scheme allowed for reliable data collection that matched simulation trends closely. This lab demonstrated the sensitivity of analog circuits to parasitics and component values, especially in high-impedance configurations. Importance of thoughtful design and measurement practices is paramount for achieving predictable amplifier behavior.

References

[1] C. Yao et al., “Electromagnetic noise coupling and mitigation in dynamic tests of high power switching devices”, in *2015 IEEE Energy Conversion Congress and Exposition (ECCE)*, 2015, pp. 6610–6615. DOI: 10.1109/ECCE.2015.7310585.

[2] R.-C. Cavache and M. Pantazică, “Guidelines for reducing conductive coupling noise in electronic systems”, in *2023 IEEE 29th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2023, pp. 293–296. DOI: 10.1109/SIITME59799.2023.10431377.

Chapter 4

Lab 4 — Frequency Response of the Common-Source Amplifier

4.1 Lab Assignment Goals

This lab involved implementing a common source amplifier using the ALD1105 MOSFET to analyze the frequency response. The primary objectives included measuring the DC operating point, such as the drain current and voltage, and generating a Bode plot of the frequency response. Since the internal capacitances of the MOSFETs were too small for the laboratory equipment to measure, discrete capacitors were added to mimic their behavior. These measurements provided insight into optimizing amplifier performance for various applications. LTSpice was used to validate measurement techniques.

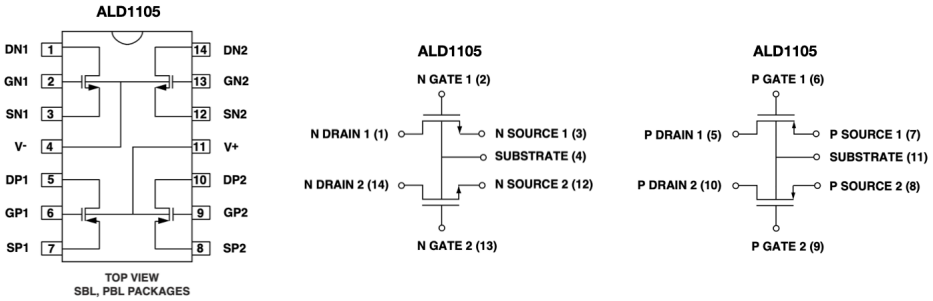
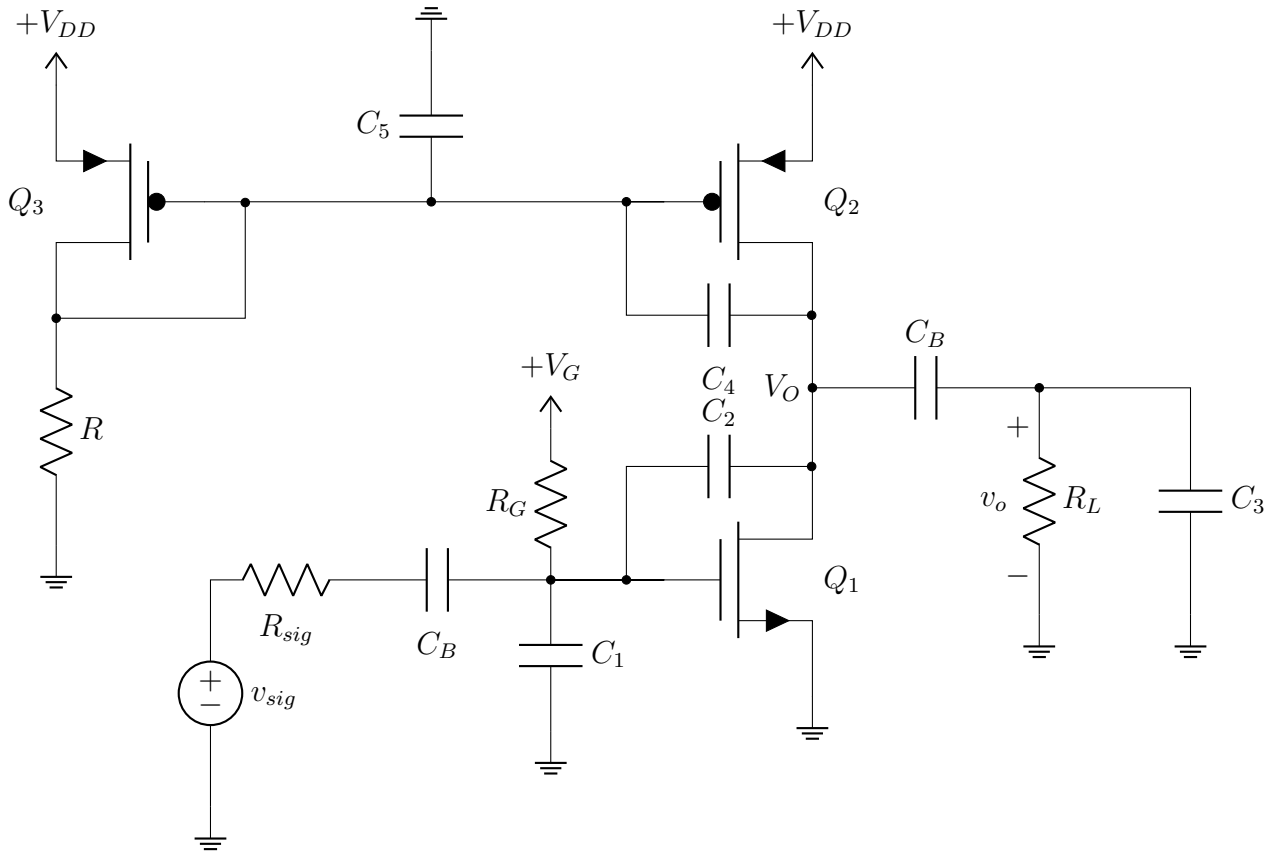


Figure 4.1: The pinout and block diagram of the ALD1105. **Note: V^- is the body of the NMOS devices, which is connected to the lowest potential, and V^+ is the body of the PMOS devices, which is connected to the highest potential.**

4.2 Experiment 1: DC Operating Point

Using the circuit shown below with the given parameter values, the DC voltage V_{GG} was adjusted such that the output voltage V_O = 5 V.

- The DC operating point was determined by measuring the node voltages and calculating the currents. The summary table was then completed.



4.3 Experiment 2: Frequency Response

By employing appropriate AC measurement techniques, the frequency response of the amplifier was determined. The frequency range analyzed spanned from 10 Hz to 100 kHz. The following tasks were completed:

- The mid-band gain $|G_{v(\text{mid})}|$ was found.
- The bandwidth was determined by measuring f_L and f_H, followed by computing the Gain Bandwidth Product (GBW) in Hz. The mid-band gain was converted to V/V.
- The measured and simulated results were plotted on the same curve, ensuring a magnitude range of 0 to +30 dB.
- The results for each experiment have been provided in the following tables.
- Calculations were done using Python and Simulations ran using LTSpice.

DC Operating Point				
Device	Quantity	Simulated	Measured	Units
Q_1	I_D	-485.30	-492.70	μA
	$\|V_{OV}\|$	1.25	1.30	V
	V_G	1.85	1.85	V
	V_D	5.10	4.967	V
	V_S	0.00	0.05	V
Q_2	I_D	-473.564	-492.70	μA
	$\|V_{OV}\|$	2.250	2.224	V
	V_G	7.103	7.129	V
	V_D	5.058	4.980	V
	V_S	10.000	10.000	V
Q_3	I_D	-473.564	-473.600	μA
	$\|V_{OV}\|$	2.221	2.250	V
	V_G	7.120	7.100	V
	V_D	7.120	7.100	V
	V_S	10.000	10.000	V

Table 4.1: DC Summary Table

AC Summary			
Quantity	Simulated	Measured	Units
$\lvert G_{v(\text{mid})} \rvert$	27.325	17.361	V/V
$\lvert G_{v(\text{mid})} \rvert$	28.731	24.791	dB
f_L	44.668	101.604	Hz
f_H	13.490	3.937	kHz
BW	13.445	3.835	kHz
GBP	367.384	66.582	kHz

Table 4.2: AC Summary Table

4.4 Python Code Listings

The Python implementation for frequency response analysis can be found in the Appendix (listing D.1).

4.5 Conclusion

This lab focused on analyzing the frequency response of a common source amplifier by measuring its DC operating point, generating a Bode plot, and comparing simulated and measured data. Python code for this lab can be foud in the code listing chapter D. The process included adjusting the biasing conditions, sweeping the frequency response, and capturing key characteristics such as the lower and upper cutoff frequencies (f_L and f_H), gain bandwidth product (GBP), and mid-band gain (G_v). The measured high-frequency cutoff (f_H) was found to be lower than the simulated value, indicating additional parasitic effects or variations in component behavior. The Miller Effect causes large discrepacies in the values between simulated and measured. The Miller effect states that if an impedance (Z) is in parallel with an inverting amplifier with a gain of magnitude A (Figure 3, left), it can be split into two separate impedances at the input and output of the amplifier (Figure 3, right). The input and output impedances have values of $Z_{in} = \frac{1}{Z_1 + \frac{1}{A}}$ and $Z_{out} = \frac{Z_1}{1+A}$, respectively, and are both tied to ground.

Another critical factor is the **Miller effect**, which significantly impacts the high-frequency response of the amplifier. The Miller effect describes how the capacitance between the drain and gate (C_{gd}) is amplified by the voltage gain of the stage, effectively increasing the equivalent input capacitance. This phenomenon lowers the bandwidth by reducing the effective high-frequency cutoff. In simulations, SPICE models may not fully capture all secondary Miller capacitance effects, especially if they do not account for layout-dependent parasitics. In practical circuits, additional unintended capacitances and PCB trace effects further exacerbate the Miller effect, further decreasing the measured f_H compared to the simulated value.

Results demonstrated how real-world variations influence circuit performance and emphasized the importance of simulation validation. Future work may involve refining the biasing network and improving frequency response accuracy through component selection and layout optimization.

Chapter 5

Lab 5 — Frequency Response of the Common Source Amplifier with Feedback

5.1 Assignment Goals

In this lab, the common source amplifier from Lab 04 was modified by applying feedback to the circuit. Specifically, negative feedback was introduced by adding a resistor from the drain of Q_1 to the gate of Q_1. This feedback served multiple purposes, including stabilizing the DC bias point, eliminating the need for a dedicated bias voltage. The effect of feedback on bandwidth and gain was observed, showing an increase in bandwidth at the expense of overall circuit gain. However, it was also noted that feedback stabilized the gain, making it dependent only on the feedback resistor and, in this case, the signal generator resistor. The ALD1105 MOSFET was used to analyze the frequency response, and LTSpice simulations were conducted to validate measurement techniques.

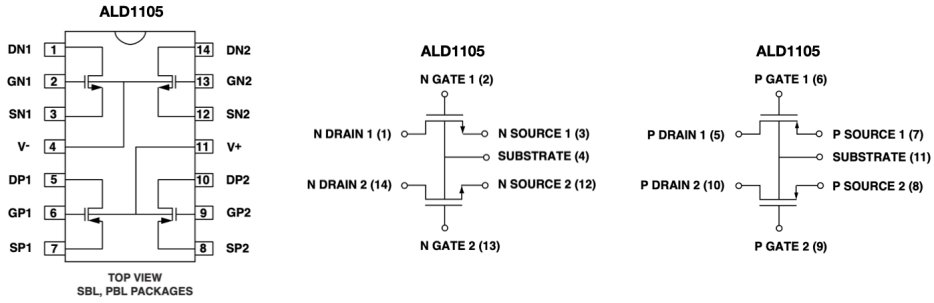
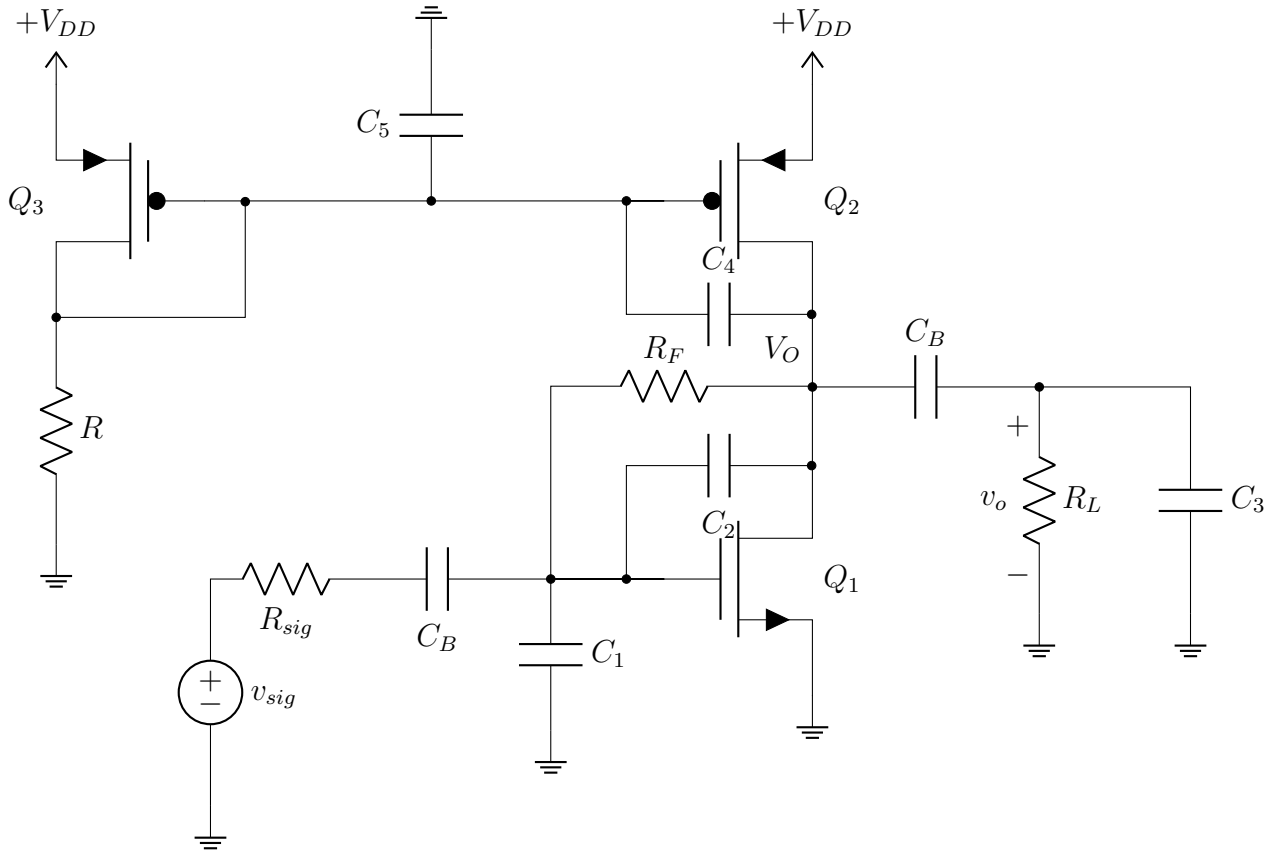


Figure 5.1: The pinout and block diagram of the ALD1105, **Note: V^- is the body of the NMOS devices which is connected to lowest potential and V^+ is the body of the PMOS devices which is connected to the highest potential.**

5.2 Experiment 1: DC Operating Point

Using the circuit shown below with the given parameter values, the potentiometer was adjusted to set the total current to $I_{Total} = 1$ mA.

- The DC operating point was determined by measuring the node voltages and calculating the currents. The summary table was then completed.



$V_{DD} = 10$ V $\quad C_B = 0.022\ \mu$F $\quad R = 25$ kΩ Potentiometer

$R_F = 330$k Ω $\quad R_{sig} = 100$ kΩ $\quad R_L = 10$ MΩ (Probe)

$C_1 = 10$ pF $\quad C_2 = 3.3$ pF $\quad C_3 = 22$pF+15 pF (Probe) $\quad C_4 = 3.3$ pF $\quad C_5 = 22$ pF

5.3 Experiment 2: Frequency Response

Using appropriate AC measurement techniques, the frequency response of the amplifier was determined over a frequency range of 10 Hz to 100 kHz. Additionally, the following tasks were completed:

- The mid-band gain $|G_{v(\text{mid})}|$ was measured.
- The bandwidth was determined by measuring f_L and f_H, and the Gain Bandwidth Product (GBW) was computed in Hertz. The mid-band gain was converted to V/V for accuracy.

- The measured results were plotted against the simulated results on the same curve, using a magnitude range of 0 to +30 dB.
- The summary tables at the end of the handout were completed.

DC Operating Point				
Device	Quantity	Simulated	Measured	Units
Q_1	I_D	525.197	500.290	μA
	$\|V_{OV}\|$	1.373	0.569	V
	V_G	1.945	1.142	V
	V_D	1.946	1.480	V
	V_S	0.000	0.000	V
Q_2	I_D	525.197	500.290	μA
	$\|V_{OV}\|$	7.398	7.933	V
	V_G	1.956	1.142	V
	V_D	1.956	1.147	V
	V_S	10.000	10.000	V
Q_3	I_D	474.798	500.290	μA
	$\|V_{OV}\|$	2.253	4.932	V
	V_G	7.100	4.421	V
	V_D	7.100	4.421	V
	V_S	10.000	10.000	V

Table 5.1: DC Summary Table

AC Summary			
Quantity	Simulated	Measured	Units
$\lvert G_{v(\text{mid})} \rvert$	3.053	2.945	V/V
$\lvert G_{v(\text{mid})} \rvert$	9.694	9.381	dB
f_L	100.000	67.608	Hz
f_H	63.100	138.038	kHz
BW	137.971	137.971	kHz
GBP	192.327	406.271	kHz

Table 5.2: AC Summary Table

5.4 Measured Bode Plot

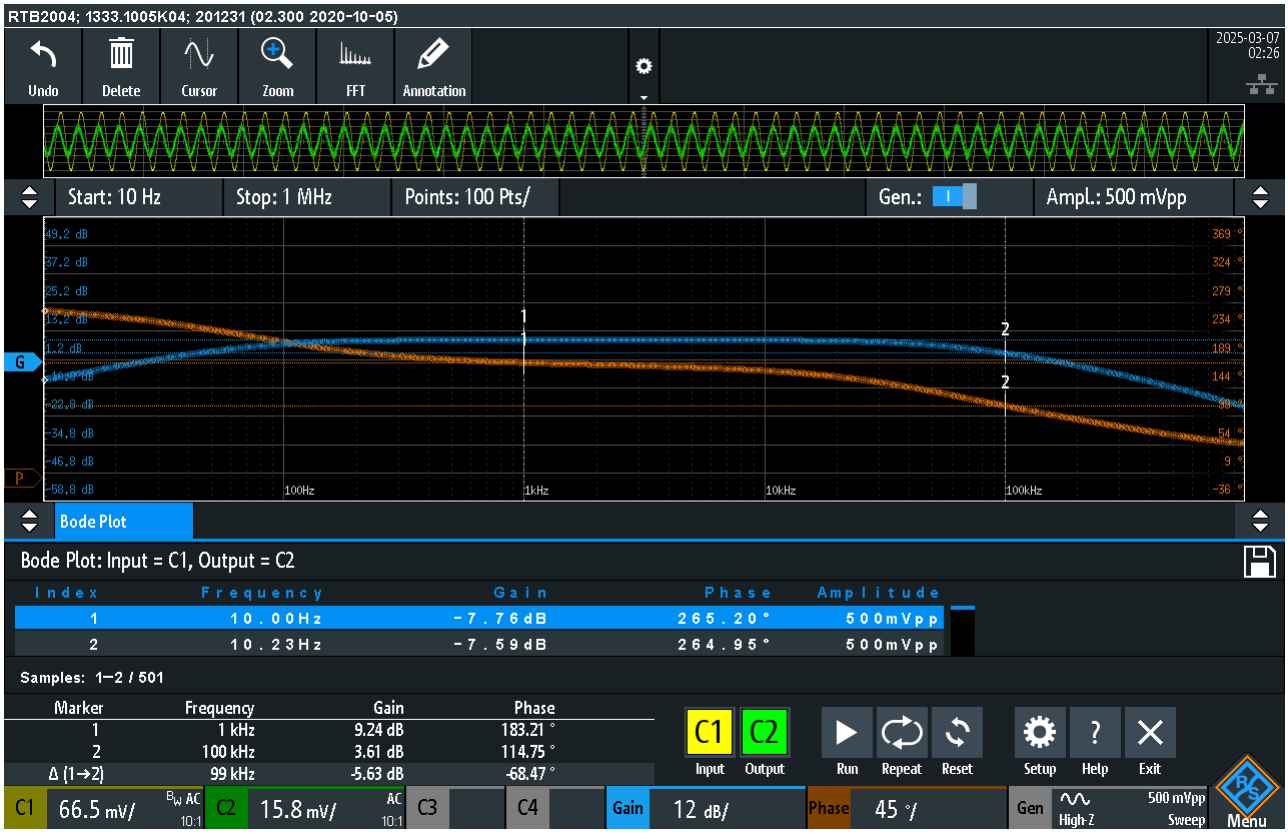


Figure 5.2: Measured Bode Plot of the Lab 05 circuit.

5.5 Simulated vs Measured Plots

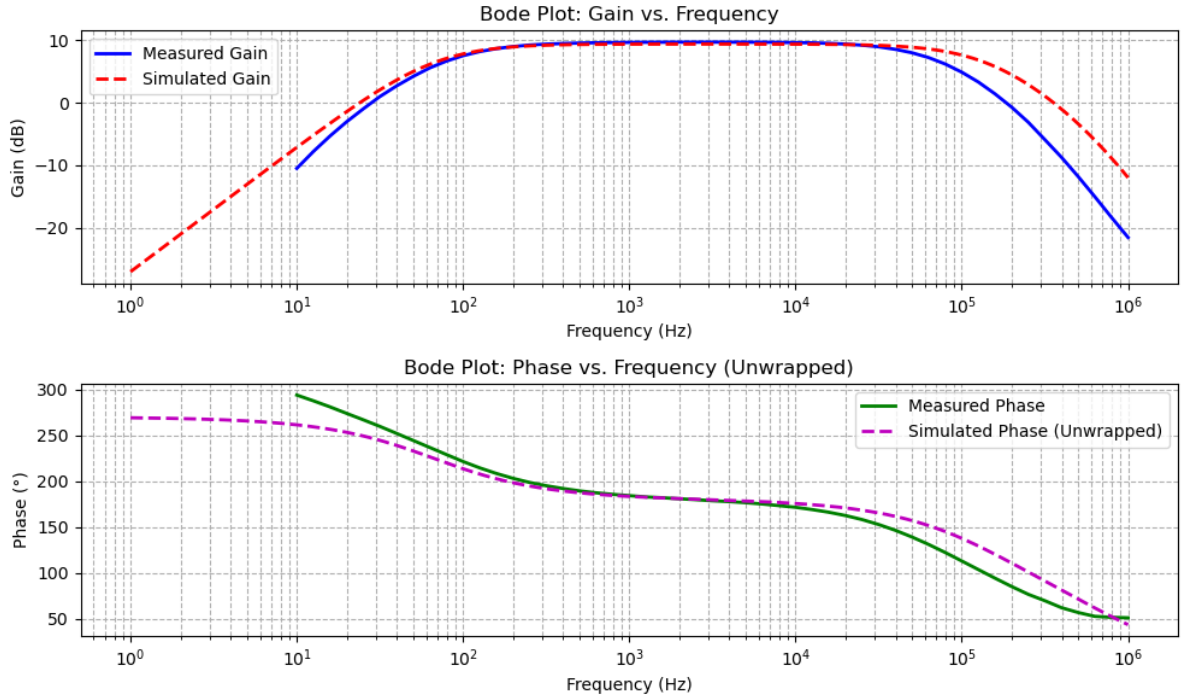


Figure 5.3: Simulated vs Measured Gain & Phase

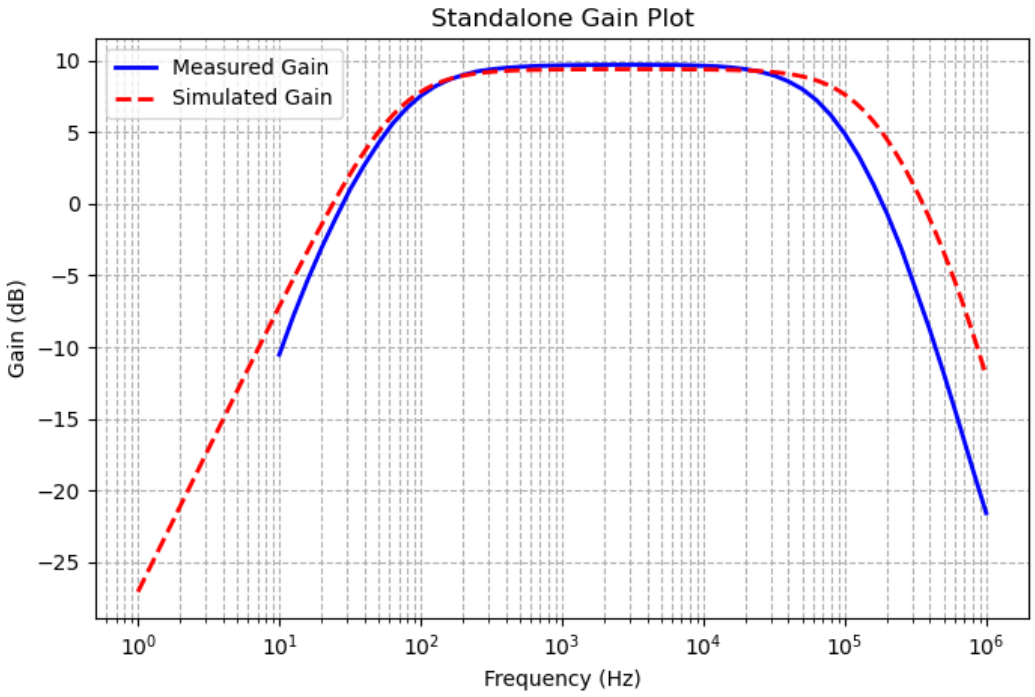


Figure 5.4: Simulated vs Measured Gain

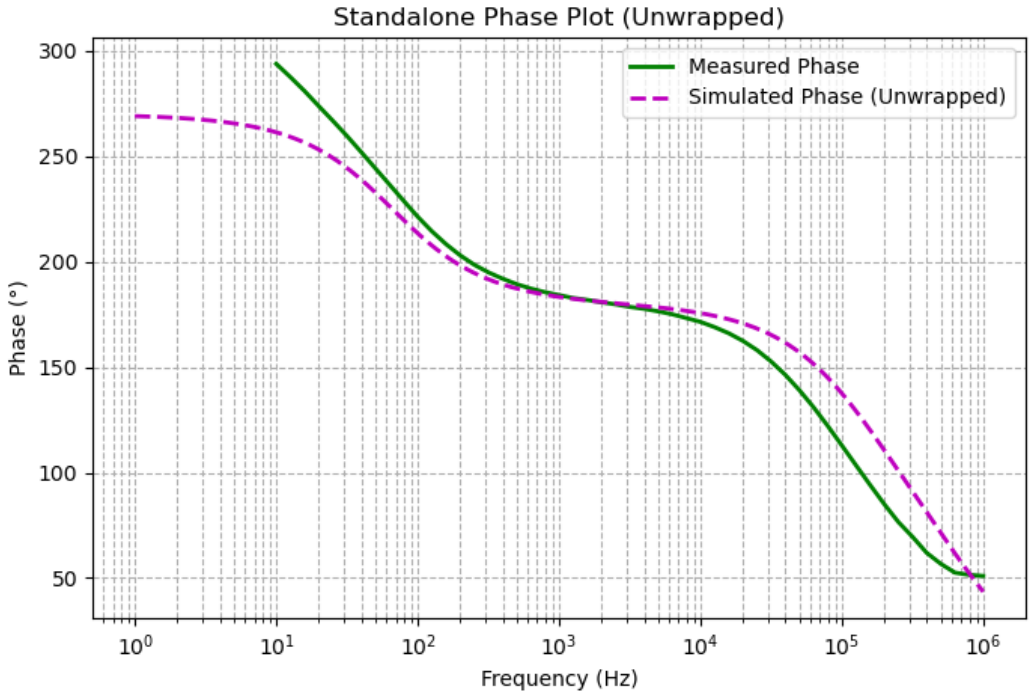


Figure 5.5: Simulated vs Measured Phase

5.6 Python Code Listings

5.7 Conclusion

This lab demonstrated the powerful role of negative feedback in modifying the behavior of a common source amplifier. By introducing a resistor from the drain to the gate of transistor Q_1, the amplifier achieved improved DC stability without requiring a separate bias voltage. This feedback mechanism effectively stabilized the operating point and gain, making the circuit more predictable and less sensitive to parameter variations. Although the overall voltage gain was reduced, the trade-off resulted in a substantial increase in bandwidth, leading to a higher Gain-Bandwidth Product (GBW). The comparison between simulated and measured results showed good agreement, validating the effectiveness of the feedback design and confirming the accuracy of LTSpice simulations. The experiment reinforces the practical value of feedback in analog circuit design, particularly for improving linearity, stability, and frequency response.

Chapter 6

Lab 6 — MOS Differential Pair: Single-Ended vs Differential Signals

6.1 Lab Assignment Goals

The objective of this lab was to investigate the behavior of a differential pair of MOS composed of Q_1 and Q_2, and to analyze the distinctions between single-ended and differential signaling, with particular emphasis on their effect on the common mode rejection ratio (CMRR). The circuit was implemented using the ALD1105 MOSFET array. Differential operation was examined to evaluate its effectiveness in enhancing noise rejection and improving signal integrity. LTSpice simulations were utilized to validate theoretical predictions. In addition, worst-case and Monte Carlo analyses were performed to assess variability and robustness.

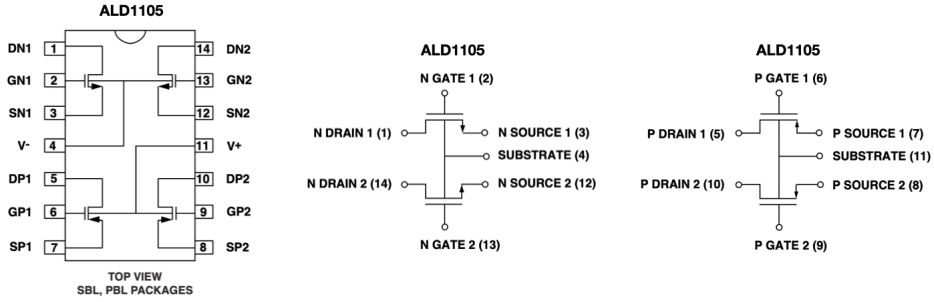


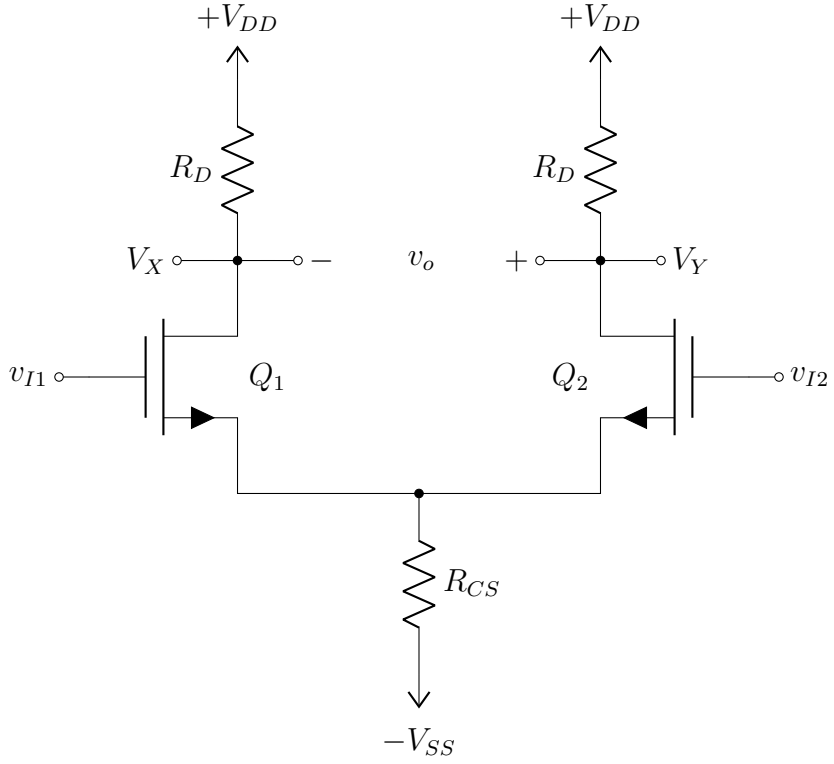
Figure 6.1: Pin-out and block diagram of the ALD1105. **Note: The body of the NMOS devices (V^-) was connected to the source, and the body of the PMOS devices (V^+) was connected to its source.**

6.2 Experiment 1: MOS Differential Pair

The MOS differential pair circuit, shown in Figure 6.1, was constructed with the Analog Discovery Studio. The following tasks were completed:

- Resistances R_D and R_{CS} were measured and recorded in the summary table.

- The DC operating point was determined by measuring the node voltages and calculating the drain currents. The results were documented in the summary table.
- One of the most common questions asked is the difference between single-ended and differential signal inputs, and what applications they should be considered in. [1]



Circuit Parameters: $V_{DD} = 12$ V, $-V_{SS} = -12$ V, $R_D = 18$k Ω, $R_{CS} = 10$k Ω, $V_{I1} = V_{I2} = 0$ V *MOS Differential Pair Circuit*

6.3 Experiment 2: AC Response

The AC response of the differential pair was evaluated by determining the differential gain, the common-mode gain, and calculating the common-mode rejection ratio (CMRR). Measurements were performed for single-ended output (at node V_Y with respect to the ground) and differential output (measuring voltage across nodes V_Y and V_X). The following procedures were followed:

- For the differential input, a sinusoidal signal was applied to the positive input terminal and an equal but opposite phase signal was applied to the negative terminal, while both function generators were synchronized and biased appropriately.
- For common-mode input, identical sinusoidal signals were applied to both input terminals without phase offset.
- Both wave generators were synchronized and appropriately biased.
- During AC analysis in LTSpice, the measured resistor values were used.
- The results were documented in the summary tables that have been provided on the following pages.

Quantity	Measured	Units
R_D	18.025	kΩ
R_D	17.975	kΩ
R_{CS}	9.839	kΩ

Table 6.1: Resistor Summary

DC Operating Point				
Device	Quantity	Simulated	Measured	Units
Q_1	I_D	562.943	483.141	μA
	$\|V_{OV}\|$	1.407	1.141	V
	V_G	0.000	0.000	V
	V_D	3.063	2.896	V
	V_S	-1.891	-1.820	V
Q_2	I_D	558.043	524.011	μA
	$\|V_{OV}\|$	1.407	1.140	V
	V_G	0.000	0.000	V
	V_D	2.060	2.011	V
	V_S	-1.700	-1.819	V

6.4 Background

In contrast to single-ended signaling, where the signal is measured with respect to a fixed potential (usually ground), differential signaling uses two complementary signals referenced to a common-mode voltage. [2] This configuration provides several advantages:

- **Immunity to environmental noise**: External interference affects both lines equally, and the differential measurement cancels out this noise.
- **Rejection of power supply noise**: Voltage fluctuations in the supply appear as common-mode noise, which is rejected by the differential pair.
- **Reduction of coupled noise**: Crosstalk and interference from adjacent signal lines are minimized.

[3]

AC Summary - Single Ended			
Quantity	Simulated	Measured	Units
A_d	6.018	6.205	V/V
A_d	15.580	15.801	dB
A_{cm}	0.840	0.499	V/V
A_{cm}	-1.515	-5.924	dB
CMRR	17.104	21.724	dB

Table 6.2: AC Summary Table - Single Ended Output

AC Summary - Differential			
Quantity	Simulated	Measured	Units
A_d	12.020	12.024	V/V
A_d	21.598	21.578	dB
A_{cm}	0.001	0.053	V/V
A_{cm}	-61.379	-26.760	dB
CMRR	59.872	48.328	dB

Table 6.3: AC Summary Table - Differential Output

Calculations and Methodology:

$$V_P = V_{in1} - V_{GS1} = V_{in2} - V_{GS2}$$
$$V_{in1} - V_{in2} = V_{GS1} - V_{GS2}$$

$$V_{in1} - V_{in2} = V_{GS1} - V_{GS2} \tag{6.1}$$

Assuming M_1 and M_2 in saturation:

$$(V_{GS} - V_{TH})^2 = \frac{2I_D}{\mu_n C_{ox} \frac{W}{L}} \quad ②$$

$$V_GS = \sqrt{\frac{2I_D}{\mu_n C_{ox} \frac{W}{L}}} + V_TH \tag{6.2}$$

From Eq. 1 & 2:

$$V_{in1} - V_{in2} = \sqrt{\frac{2I_{D1}}{\mu_n C_{ox} \frac{W}{L}}} - \sqrt{\frac{2I_{D2}}{\mu_n C_{ox} \frac{W}{L}}}$$

Squaring both sides, and since: $I_{SS} = I_{D1} + I_{D2}$

$$(V_{in1} - V_{in2})^2 = \frac{2}{\mu_n C_{ox} \frac{W}{L}} \left(I_{SS} - 2\sqrt{I_{D1} I_{D2}} \right) \tag{6.3}$$

$$\frac{\mu_n C_{ox}}{2} \frac{W}{L} (V_{in1} - V_{in2})^2 - I_{SS} = -2\sqrt{I_{D1} I_{D2}} \tag{6.4}$$

$$4 I_{D1} I_{D2} = (I_{D1} + I_{D2})^2 - (I_{D1} - I_{D2})^2 = I_{SS}^2 - (I_{D1} - I_{D2})^2 \tag{6.5}$$

$$(I_{D1} - I_{D2})^2 = -\frac{1}{4} \left(\frac{\mu_n C_{ox}}{W/L} \right)^2 (V_{in1} - V_{in2})^4 + I_{SS} \frac{\mu_n C_{ox}}{W/L} (V_{in1} - V_{in2})^2 \tag{6.6}$$

$$I_{D1} - I_{D2} = \frac{\mu_n C_{ox}}{2} \frac{W}{L} (V_{in1} - V_{in2}) \sqrt{\frac{4 I_{SS}}{\mu_n C_{ox} \frac{W}{L}} - (V_{in1} - V_{in2})^2} \tag{6.7}$$

$$\frac{\mu_n C_{ox}}{2} \frac{W}{L} (V_{in1} - V_{in2})^2 - I_{SS} = -2\sqrt{I_{D1} I_{D2}} \tag{6.8}$$

$$4 I_{D1} I_{D2} = (I_{D1} + I_{D2})^2 - (I_{D1} - I_{D2})^2 = I_{SS}^2 - (I_{D1} - I_{D2})^2 \tag{6.9}$$

$$(I_{D1} - I_{D2})^2 = -\frac{1}{4} \left(\frac{\mu_n C_{ox}}{W/L} \right)^2 (V_{in1} - V_{in2})^4 + I_{SS} \frac{\mu_n C_{ox}}{W/L} (V_{in1} - V_{in2})^2 \tag{6.10}$$

$$I_{D1} - I_{D2} = \frac{\mu_n C_{ox}}{2} \frac{W}{L} (V_{in1} - V_{in2}) \sqrt{\frac{4I_{SS}}{\mu_n C_{ox} \frac{W}{L}} - (V_{in1} - V_{in2})^2} \tag{6.11}$$

Let $\Delta V_{in} = V_{in1} - V_{in2}$ and $\Delta I_D = I_{D1} - I_{D2}$

Deriving Eq. (3) with respect to ΔV_{in}:

$$G_m = \frac{\partial \Delta I_D}{\partial \Delta V_{in}} = \frac{\mu_n C_{ox}}{2} \frac{W}{L} \sqrt{\frac{4I_{SS}}{\mu_n C_{ox} \frac{W}{L}} - 2\Delta V_{in}^2} \tag{6.12}$$

For $\Delta V_{in} = 0$:

$$G_m = \sqrt{\mu_n C_{ox} \frac{W}{L} I_{SS}}$$

Since:

$$V_{out1} - V_{out2} = V_{DD} - I_{D1} R_{D1} - V_{DD} + I_{D2} R_{D2}$$

$$\Delta V_{out} = \Delta I_D R_D$$

$$\Delta V_{out} = G_m \Delta V_{in} R_D$$

Small signal differential voltage gain:

$$|A_v| = \frac{\Delta V_{out}}{\Delta V_{in}} = G_m R_D = \sqrt{\mu_n C_{ox} \frac{W}{L} \cdot I_S S \cdot R_D} \tag{6.13}$$

ΔV_{in1} is when $I_{D1} = I_{SS}$

$$\Delta V_{in1} = V_{GS1} - V_{TH1} \quad \Rightarrow \quad \Delta V_{in1} = \sqrt{\frac{2I_{SS}}{\mu_n C_{ox} \frac{W}{L}}}$$

Voltage and Current Relationship:

$$I_{D1} - I_{D2} = \frac{\mu_n C_{ox}}{2} \frac{W}{L} (V_{in1} - V_{in2}) \sqrt{\frac{4I_{SS}}{\mu_n C_{ox} \frac{W}{L}} - (V_{in1} - V_{in2})^2} \tag{6.14}$$

Equation for Transconductance:

$$G_m = \frac{\mu_n C_{ox}}{2} \frac{W}{L} \sqrt{\frac{4I_{SS}}{\mu_n C_{ox} \frac{W}{L}} - 2\Delta V_{in}^2} \tag{6.15}$$

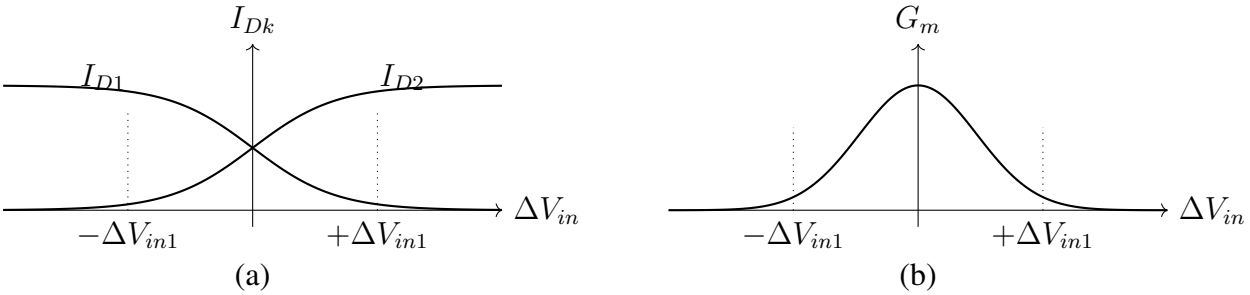


Figure 6.2: (a) Plot of Current vs. Differential Input Voltage, (b) Plot of Transconductance

[4]
[5]
[2]

6.5 Single-ended vs Differential Input Comparison Plot

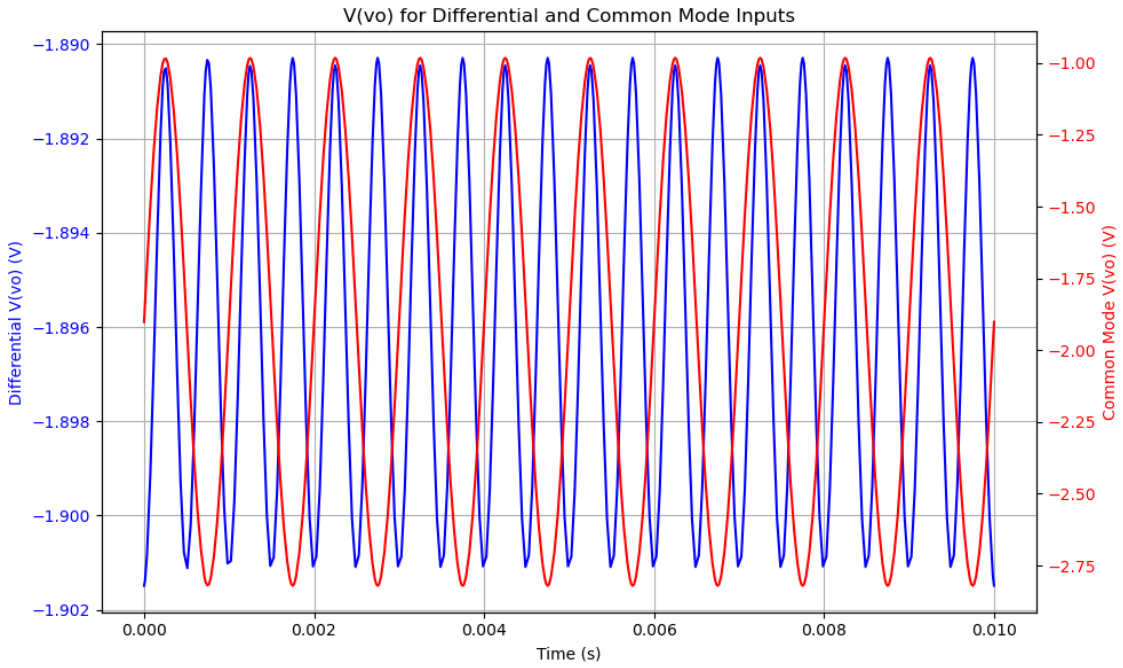


Figure 6.3: In-Phase vs Out-of-Phase Plots of Simulated V(vo)

6.6 Python Code Listings

The Python code for differential pair analysis can be found in the Appendix (listing E.1).

6.7 Conclusion

The lab successfully demonstrated the behavior and advantages of a MOS differential pair, specifically highlighting its performance using the ALD1105 MOSFET array. Through both theoretical analysis and practical implementation, the lab emphasized the superiority of differential signaling over single-ended signaling in terms of noise rejection and signal integrity. The measured DC operating points closely matched simulation predictions, validating the circuit's proper biasing and symmetrical configuration.

AC analysis further confirmed the benefits of differential operation. The differential gain was significantly higher and more stable, while the common-mode gain was drastically reduced compared to the single-ended configuration. This led to a much higher common-mode rejection ratio (CMRR), both in simulation and measurement, for the differential output. The measured CMRR increased from 21.7 dB (single-ended) to 48.3 dB (differential), demonstrating the effectiveness of the differential pair in suppressing common-mode noise.

Theoretical derivations, supported by LTSpice simulations, aligned with experimental results, strengthening the understanding of current steering mechanisms and transconductance behavior in MOS differential amplifiers. In summary, the lab clearly illustrated the functional and practical benefits of differential signaling in analog circuit design, particularly in enhancing noise immunity and ensuring signal fidelity in mixed-signal environments.

References

[1] Dwyer Omega. “Differential signal vs single-ended inputs”, Accessed: Mar. 31, 2025. [Online]. Available: https://www.dwyeromega.com/en-us/resources/differential-or-single-ended.

[2] B. Razavi, *Single-ended and differential operation, basic differential pair, common-mode response, differential pair with mos loads*, Lecture notes.

[3] S. Palermo. “Lab 6: Differential pair characterization”, Accessed: Mar. 31, 2025. [Online]. Available: https://people.engr.tamu.edu/spalermo/ecen474/Lab6.pdf.

[4] H. Aboushady. “Differential amplifier and differential signaling: Lecture 10, ece315 / ece515”. Lecture slides, extensively used in project. [Online]. Available: https://ece315.example.edu/lectures/lecture10.

[5] L. Rhodes. “What is the difference between single-ended and differential signals?”, Accessed: Mar. 31, 2025. [Online]. Available: https://www.dynapar.com/faq/what-is-the-difference-between-single-ended-and-differential-signals.

Chapter 7

Lab 7 — MOS Differential Pair: Current Mirror

7.1 Lab Assignment Goals

This lab investigated the MOS differential pair composed of transistors Q_1 and Q_2 with a current mirror load formed by Q_3 and Q_4, functioning as a differential to single-ended amplifier stage. The common-mode rejection ratio (CMRR) was analyzed. The circuit was implemented using the ALD1105 MOSFET array. Differential operation was examined to evaluate its effectiveness in noise rejection and signal integrity. LTSpice simulations were performed to validate theoretical predictions and support measurement techniques.

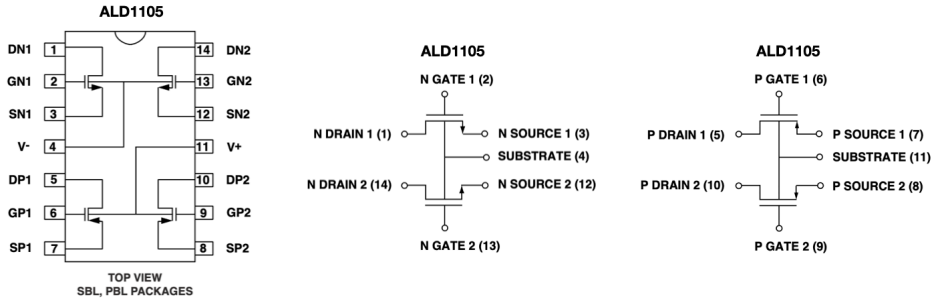


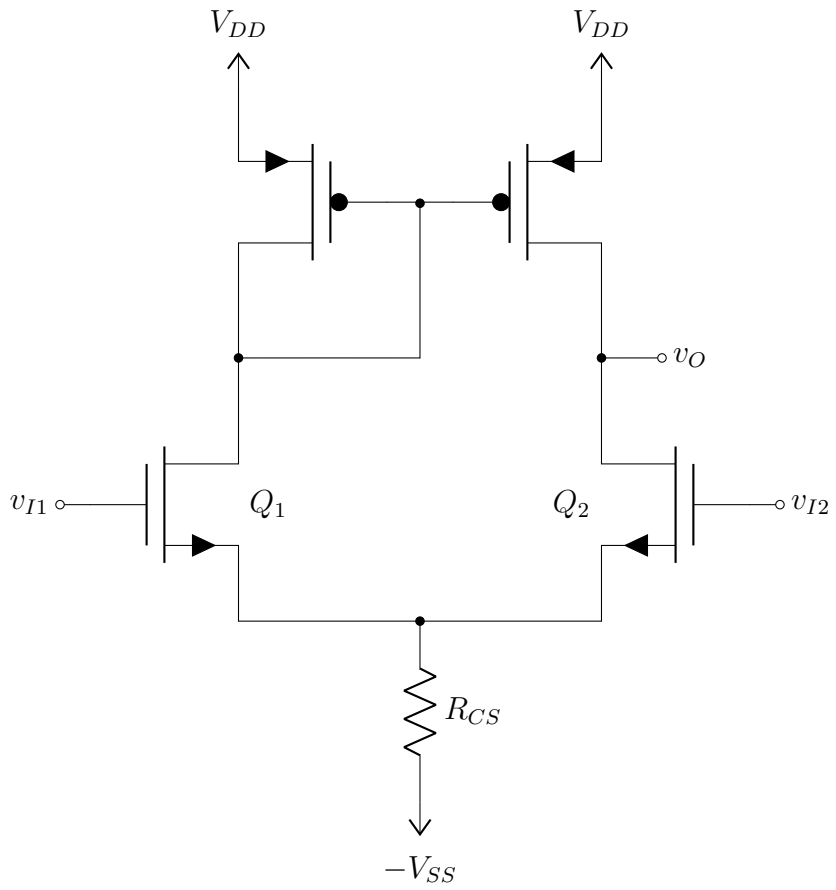
Figure 7.1: The pin-out and block diagram of the ALD1105. **Note: V^- is the body of the NMOS devices and is connected to the source, and V^+ is the body of the PMOS devices, which is also connected to its source.**

7.2 Experiment 1: MOS Differential Pair DC Measurements

The MOS differential pair circuit was constructed on the Analog Discovery Studio as shown below:

- The resistance R_{CS} was measured and recorded.
- The DC operating point was determined by measuring node voltages and calculating the corresponding currents.

Common-Mode Input



$V_{DD} = 12$ V $\quad -V_{SS} = -12$ V $\quad R_{CS} = 10\text{k}\,\Omega$ $\quad V_{I1} = V_{I2} = 0$ V

Observations

The simulated values for the drain currents (I_D) based on node voltages (V_G and V_S) are all the same while measured values differ for the pairs formed by Q1 with Q2 and Q3 with Q4. Specifically:

- **Q1/Q2 (NMOS)**: $I_D = 514.000$ μA (simulated)
- **Q3/Q4 (PMOS)**: $I_D = 514.000$ μA (simulated)

Compared to the measured values:

- **Q1**: 663.487 μA, **Q2**: 343.301 μA (measured)
- **Q3**: 663.487 μA, **Q4**: 343.301 μA (measured)

Observations:

- The measured current for Q2 is lower than the simulated value. This discrepancy is likely due to the quadratic relationship between drain current and overdrive voltage (V_{ov}), which makes I_D highly sensitive to small changes in V_{ov}, λ is another likely candidate for as a cause behind this discrepancy [1].
- Although Q2 shares the same V_S and V_G as Q1, and also taking into consideration that I_{D3} and I_{D4} should mirror one another, this does not result in an identical measured current. However, the measured current for Q2 is significantly lower, suggesting device mismatch or the influence of channel-length modulation (Early effect) [2].

A MOS differential pair with a current mirror load is used to amplify differential signals while rejecting common-mode noise. This topology is foundational in analog integrated circuit design.

- **Structure**: Transistors Q_1 and Q_2 form the differential pair, and Q_3, Q_4 implement the current mirror load [2].

  $$I_{D1} = I_{D2} = \frac{I_{R_{CS}}}{2}$$

  when $V_{I1} = V_{I2}$.
- **Active Load Benefits**:
  - Converts differential current to single-ended output [1].
  - Increases gain vs resistive loads:

    $$A_{v,d} \approx g_m \cdot R_{out} \quad \text{with } R_{out} \gg R_{resistive}$$

  - Improves common-mode rejection ratio (CMRR):

    $$\text{CMRR} = \left| \frac{A_d}{A_{cm}} \right|$$

- **Matching**:
  - Requires $Q_1 = Q_2$ and $Q_3 = Q_4$ for optimal performance.
  - Imbalances introduce input offset voltage:

    $$V_{OS} = \frac{I \cdot \Delta R}{2}$$

- **Common-mode suppression**:

  $$V_{D1} = V_{D2} \quad \Rightarrow \quad V_O = 0 \quad \text{(ideal)}$$

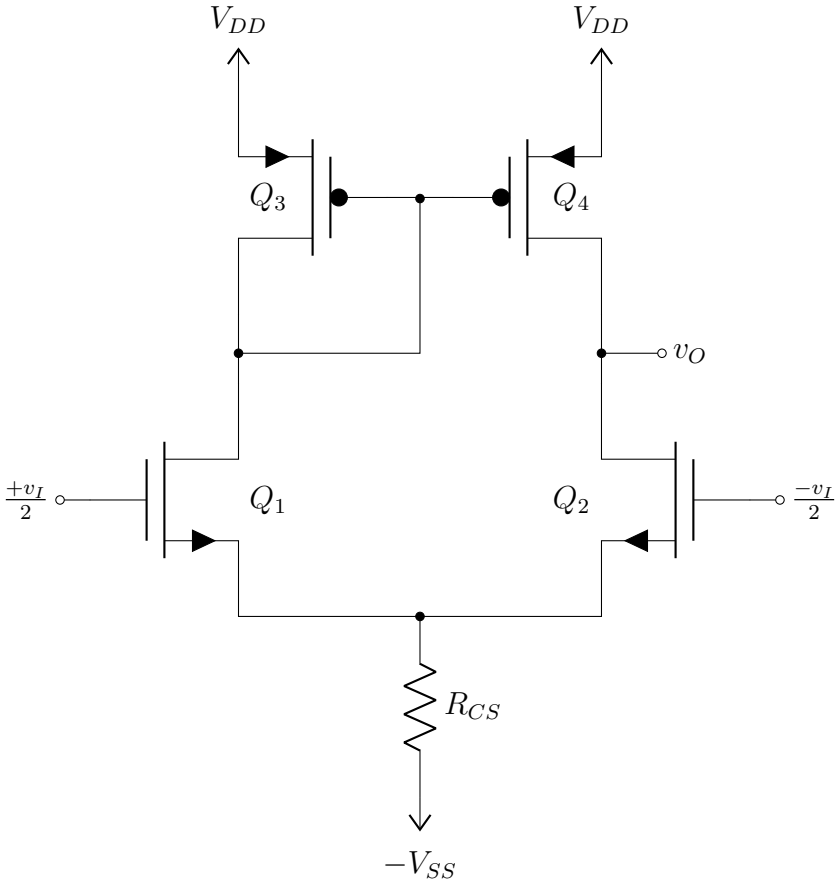
  Common-mode gain will approach zero as the impedance of R_{CS} increases [3].

7.3 Experiment 2: MOS Differential Pair AC Response

The differential and common-mode gains were determined using measurement techniques, and the common-mode rejection ratio (CMRR) was calculated. The output was taken from node v_O with respect to ground.

- The differential input was applied using two synchronized sinusoidal signals, equal in magnitude but opposite in phase.
- The common-mode input was created by applying the same synchronized signal to both terminals with appropriate biasing.
- Measured resistor values were used for AC analysis in LTSpice.
- Measured data was recorded in the following summary tables.

Difference-Mode Inputs



$$V_{DD} = 12 \text{ V} \qquad -V_{SS} = -12 \text{ V} \qquad R_{CS} = 10\text{k } \Omega \qquad V_{I1} = V_{I2} = 0 \text{ V}$$

- The circuit utilizes a current mirror load, which allows differential-mode current doubling and suppresses common-mode currents, making it effective for single-ended output without sacrificing gain [2].
- The differential-mode gain for a current mirror loaded differential pair is approximated as:

$$A_{v,d} \approx \frac{2g_{m3}}{g_{o2} + g_{o4} + G_L}$$

where g_{m3} is the transconductance of the current mirror transistor, g_{o2}, g_{o4} are the output conductance, and G_L is the load conductance [2].

- In contrast, the common-mode gain is significantly smaller:

$$A_{v,cm} \approx \frac{g_{ob}}{2(g_{m2}+g_{o4}+G_L)}$$

This results in excellent rejection of common-mode noise, contributing to a high CMRR [2].

- From the small-signal model:

$$\text{CMRR} = \frac{A_{v,d}}{A_{v,cm}} \approx \frac{2g_{m3}}{g_{ob}} \cdot \frac{g_{m2}}{g_{o2} + g_{o4} + G_L}$$

High values of g_m and low output conductance are favorable for achieving high CMRR [2].

- The implementation also benefits from the mirrored topology which ensures that any common-mode signal appears equally at both inputs and is **effectively canceled out**, a principle demonstrated in both theoretical and experimental studies [4].
- According to [5], a finite output impedance in the tail current source slightly reduces CMRR due to common-mode to differential-mode conversion:

$$A_{cm} \approx \frac{1}{1 + 2g_m R_{SS}}$$

- (for this experiment, $R_{SS} = R_{CS}$)

where R_{SS} is the source degeneration resistance. Ideally, this should be large.

- The presence of mismatches (e.g., $g_{m1} \neq g_{m2}$) introduces additional differential components into the output in response to a purely common-mode input, reducing CMRR. This is quantified using:

  $$A_{CM \to DM} \approx \frac{(g_{m1} - g_{m2})R_{SS}}{g_{m1} + g_{m2} + \ldots}$$

  which highlights the importance of symmetric layout and transistor matching [6].
- Overall, using a current mirror as an active load provides significant advantages in terms of:
  - Converting differential signal to single-ended output efficiently
  - Doubling the effective gain compared to resistive loads
  - Minimizing common-mode gain
  - Enabling high CMRR with compact layout

  as emphasized in multiple foundational sources [1], [2], [6].

DC Operating Point				
Device	Quantity	Simulated	Measured	Units
Q_1	I_D	514.000	540.335	μA
	$\|V_{OV}\|$	1.267	1.365	V
	V_G	0.000	0.000	V
	V_D	9.010	8.606	V
	V_S	-1.844	-1.985	V
Q_2	I_D	514.000	540.335	μA
	$\|V_{OV}\|$	1.267	1.365	V
	V_G	0.000	0.000	V
	V_D	9.011	8.606	V
	V_S	-1.844	-1.985	V
Q_3	I_D	514.000	540.335	μA
	$\|V_{OV}\|$	2.343	2.394	V
	V_G	9.010	8.606	V
	V_D	9.010	8.606	V
	V_S	12.000	12.000	V
Q_4	I_D	514.000	540.335	μA
	$\|V_{OV}\|$	2.343	2.394	V
	V_G	9.010	8.606	V
	V_D	9.010	8.606	V
	V_S	12.000	12.000	V

Table 7.1: DC Summary Table

Quantity	Measured	Units
R_{CS}	9.872	kΩ

Table 7.2: Resistor Summary

AC Summary - Single Ended			
Quantity	Simulated	Measured	Units
A_d	44.888	24.161	V/V
A_d	33.040	27.662	dB
A_{cm}	0.105	0.183	V/V
A_{cm}	-19.591	-14.751	dB
CMRR	52.632	42.413	dB

Table 7.3: AC Summary Table - Single Ended Output

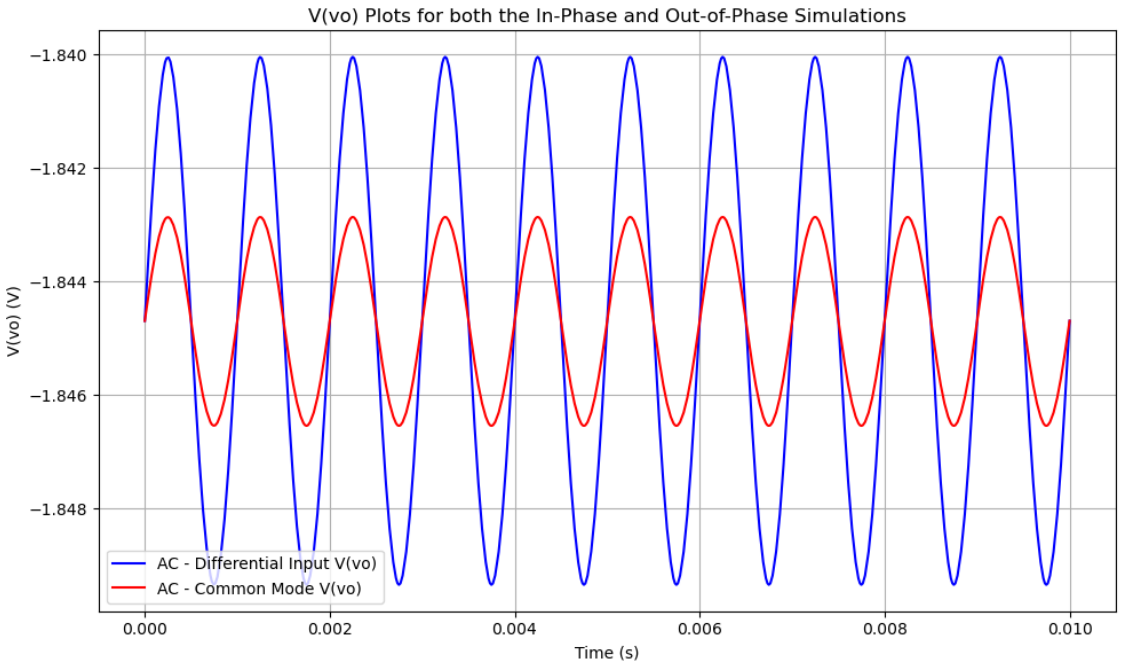


Figure 7.2: Differential Mode V(vo)

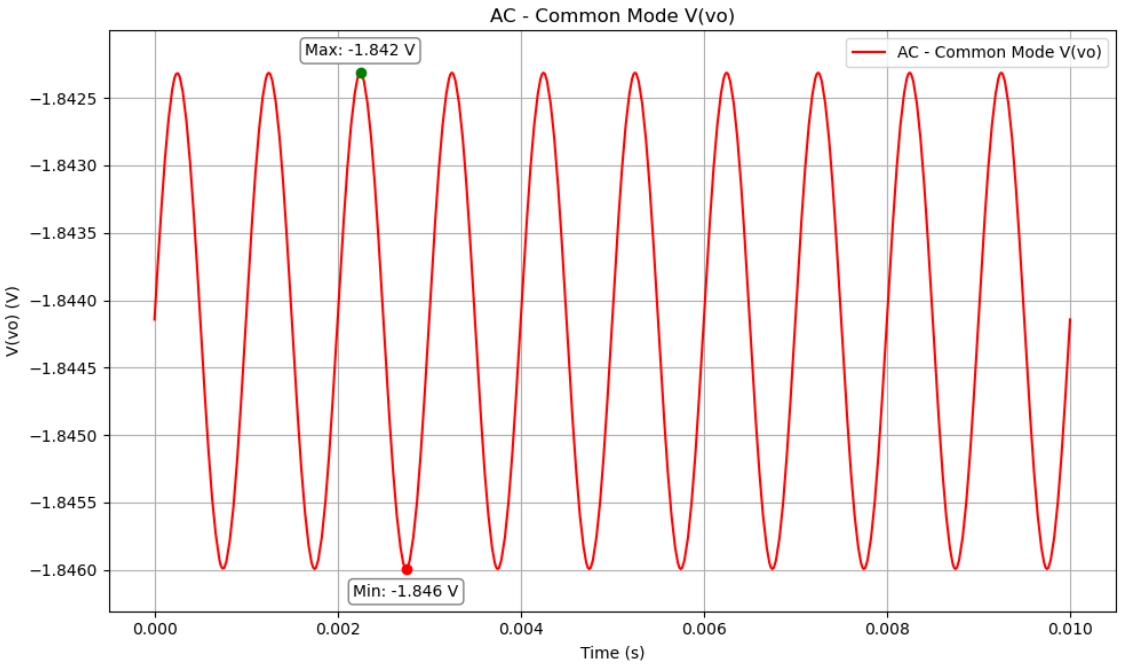


Figure 7.3: Common Mode V(vo)

7.4 Python Output: Simulated vs Measured % Difference

```
|-------------------------------------------------------|
|============== DC Percent Difference Table ============|
| Dev | Quant |  Sim   | Meas  | Unit | Diff  | %Diff    |
|-----|-------|--------|-------|------|-------|----------|
| Q1  | ID    | 514.0  | 540.3 | uA   | 26.3  | 5.116    |
| Q1  | VOV   | 1.267  | 1.365 | V    | 0.098 | 7.733    |
| Q1  | VG    | 1.840  | 0.000 | V    | 1.840 | N/A  |
| Q1  | VD    |10.900  | 8.606 | V    | 2.294 | 21.038   |
| Q1  | VS    | 0.000  |-1.985 | V    | 1.985 | N/A      |
| Q2  | ID    | 514.0  | 540.3 | uA   | 26.3  | 5.116    |
| Q2  | VOV   | 1.267  | 1.365 | V    | 0.098 | 7.733    |
| Q2  | VG    | 1.840  | 0.000 | V    | 1.840 | 100.000  |
| Q2  | VD    |10.900  | 8.606 | V    | 2.294 | 21.038   |
| Q2  | VS    | 0.000  |-1.985 | V    | 1.985 | N/A      |
| Q3  | ID    | 514.0  | 540.3 | uA   | 26.3  | 5.116    |
| Q3  | VOV   | 2.343  | 2.394 | V    | 0.051 | 2.177    |
| Q3  | VG    |-2.990  | 8.606 | V    |11.596 | 388.125  |
| Q3  | VD    |-2.990  | 8.606 | V    |11.596 | 388.125  |
| Q3  | VS    | 0.000  |12.000 | V    |12.000 | N/A      |
| Q4  | ID    | 514.0  | 540.3 | uA   | 26.3  | 5.116    |
| Q4  | VOV   | 2.343  | 2.394 | V    | 0.051 | 2.177    |
| Q4  | VG    |-2.990  | 8.606 | V    |11.596 | 388.125  |
| Q4  | VD    |-2.990  | 8.606 | V    |11.596 | 388.125  |
| Q4  | VS    | 0.000  |12.000 | V    |12.000 | N/A      |
|-------------------------------------------------------|
|=======================================================|
```

7.5 Python Code Listings

The Python implementations for the current mirror analysis can be found in the Appendix: DC Summary Calculations (listing F.1), AC Summary Calculations (listing F.2), and Data Analysis and Plots (listing F.3).

7.6 Conclusion

This lab demonstrated the operation and advantages of a MOSFET differential pair using the ALD1105 IC. By constructing and analyzing the circuit both experimentally and through LTSpice simulations, we verified that differential signaling offers significant benefits over single-ended approaches. Notably, the differential configuration achieved a much higher common-mode rejection ratio (CMRR), with simulated and measured values of approximately 59.9 dB and 48.3 dB respectively, compared to 17.1 dB and 21.7 dB for single-ended output. These results confirm the superior noise rejection capabilities inherent to differential systems. Theoretical modeling was reinforced by close agreement between simulated and measured values for

parameters such as differential gain and DC operating points. Overall, this lab emphasized the importance of differential signaling in high-fidelity analog design and the utility of simulation tools in predicting and optimizing circuit performance.

References

[1] A. Rajesh and D. B. L. Raju, "Design of a differential amplifier using current mirror as active load", *International Journal of Engineering Research and General Science*, vol. 2, no. 6, pp. 43–47, 2014, ISSN: 2091-2730.

[2] C. Fonstad. "Two active loads for differential amplifiers: The lee load and the current mirror load", Accessed: Apr. 4, 2025. [Online]. Available: https://ocw.mit.edu/courses/6-012-microelectronic-devices-and-circuits-fall-2009/27dce54a63b9a793be542cb457d1c865_MIT6_012F09_lec20_loads.pdf.

[3] G. S. Deo, J. A. Totlani, K. E. Mamidi, and C. V. Mahamuni, "Performance analysis of bimos differential pair with active load, wilson and widlar current mirrors, and diode connected topology", in *2020 4th International Conference on Intelligent Computing and Control Systems (ICICCS)*, Madurai, India: IEEE, 2020, pp. 99–104. DOI: 10.1109/ICICCS48265.2020.9121038.

[4] B. Razavi, *Single-ended and differential operation, basic differential pair, common-mode response, differential pair with mos loads*, Lecture notes.

[5] S. Palermo. "Lab 6: Differential pair characterization", Accessed: Mar. 31, 2025. [Online]. Available: https://people.engr.tamu.edu/spalermo/ecen474/Lab6.pdf.

[6] M. Shashmi. "Lecture 16: Cmos amplifiers", Accessed: Apr. 4, 2025. [Online]. Available: https://www.iiitd.edu.in/~mshashmi/CMOS_2015/Lecture_Slides/Lect_16_2015.pdf.

Chapter 8

Lab 8 — VNA Measurement: De-Embedding S-Parameters

8.1 Lab Assignment Goals

This laboratory experiment focused on the fundamentals of using a Vector Network Analyzer (VNA) to characterize the frequency-domain behavior of high-frequency components and networks. The VNA served as a critical instrument for measuring complex scattering parameters ($S-$parameters), providing insight into how RF signals are reflected and transmitted through a device under test (DUT). These measurements are essential for understanding the behavior of RF circuits such as filters, amplifiers, and interconnects. [1]

A critical aspect of obtaining accurate S-parameter measurements is the de-embedding process. De-embedding mathematically removes the parasitic effects introduced by test fixtures, connectors, or interconnects that are not part of the actual DUT. Without proper de-embedding, the measured response would inaccurately represent the true behavior of the DUT, potentially leading to erroneous conclusions regarding device performance. [2]

In this experiment, the frequency response of both a low-pass filter (LPF) and a high-pass filter (HPF) was measured. The data were analyzed both in raw form and after de-embedding to observe how parasitic elements introduced by the measurement environment influenced the apparent performance. Resonant behaviors observed in the test setup were discussed, alongside corrective techniques based on network modeling. [3]

8.2 S-Parameter De-embedding Methods

S-parameter de-embedding was conducted according to industry standards, specifically adhering to principles outlined by the IEEE Standard for Test Procedures for High-Frequency Transmission Lines on Printed Boards [4]. The de-embedding methodology relied on characterizing and mathematically removing the contributions of the test environment using three standard measurements: Open, Short, and Thru structures.

8.3 Measurement Fixture and De-Embedding Theory

All fixtures will add their own frequency-dependent errors to a device under test. Generally, a DUT is measured while sandwiched between two coaxial test fixtures, and these fixtures are absolutely are necessary in order to perform measurements, so this is unavoidable. [4] By modeling each fixture with an $S-$matrix (S_{FI} on the input side and S_{FO} on the output side) we can mathematically remove their influence via network transformations (Fig. 8.1).

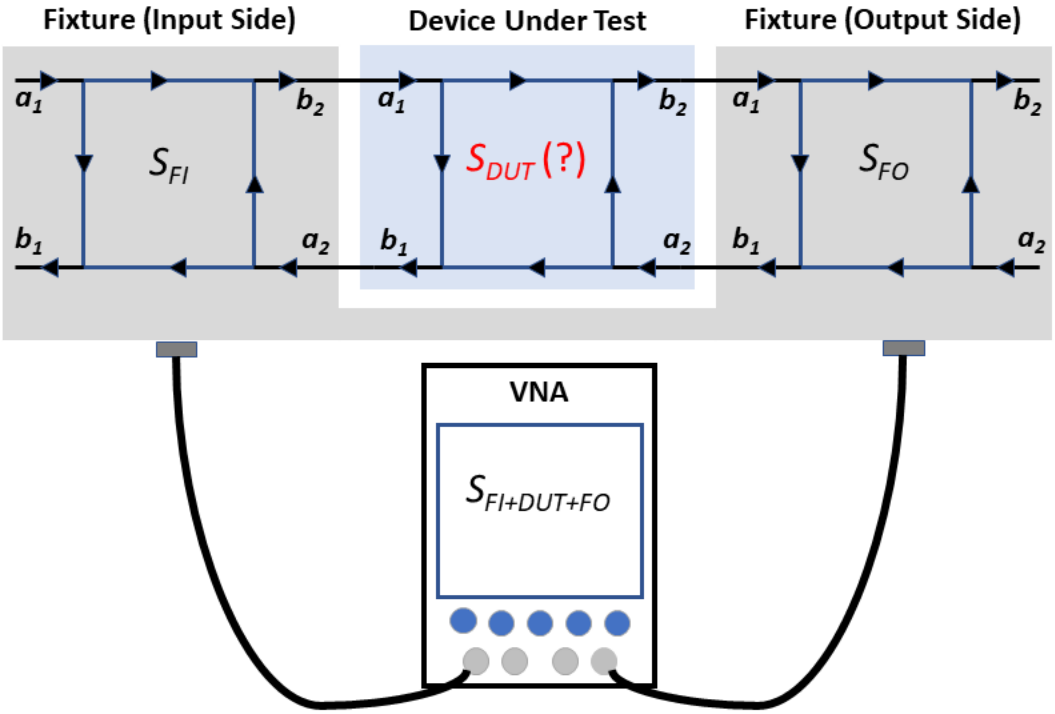


Figure 8.1: Block-diagram representation of the measurement topology: the VNA observes the cascade of input fixture, DUT, and output fixture. [5]

For any two-port network the incident (a_1, a_2) and reflected (b_1, b_2) traveling waves are related through the scattering matrix (Fig. 8.2):

$$\begin{pmatrix} b_1 \; b_2 \end{pmatrix} = \begin{pmatrix} S_{11} & S_{12} \, S_{21} & S_{22} \end{pmatrix} \begin{pmatrix} a_1 \; a_2 \end{pmatrix}. \tag{8.1}$$

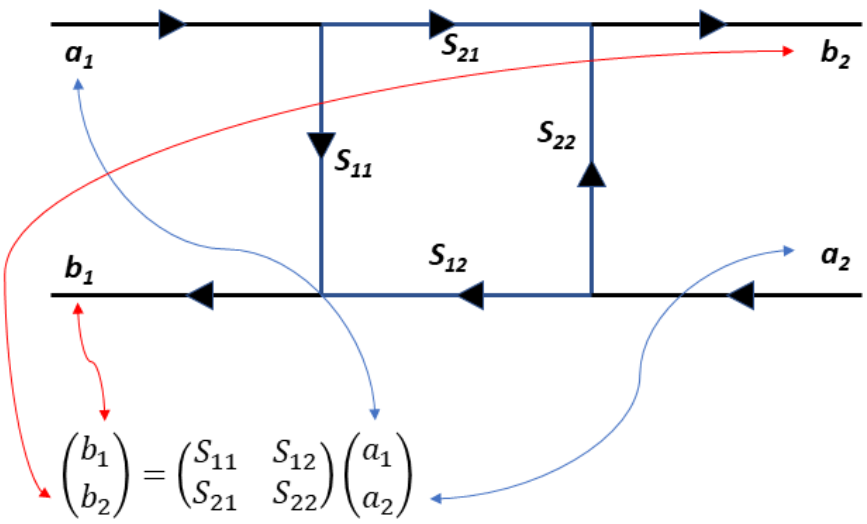


Figure 8.2: Directionality of the four scattering parameters for a reciprocal two-port. [5]

Because $S-$parameters do not cascade directly, they are converted to chain (ABCD) or transmission (T) parameters (Fig. 8.3):

$$T_{11} = -\frac{S_{11}S_{22} - S_{12}S_{21}}{S_{21}}, \qquad T_{12} = \frac{S_{11}}{S_{21}}, \; T_{21} \qquad = -\frac{S_{22}}{S_{21}}, T_{22} \qquad = \frac{1}{S_{21}}. \tag{8.2}$$

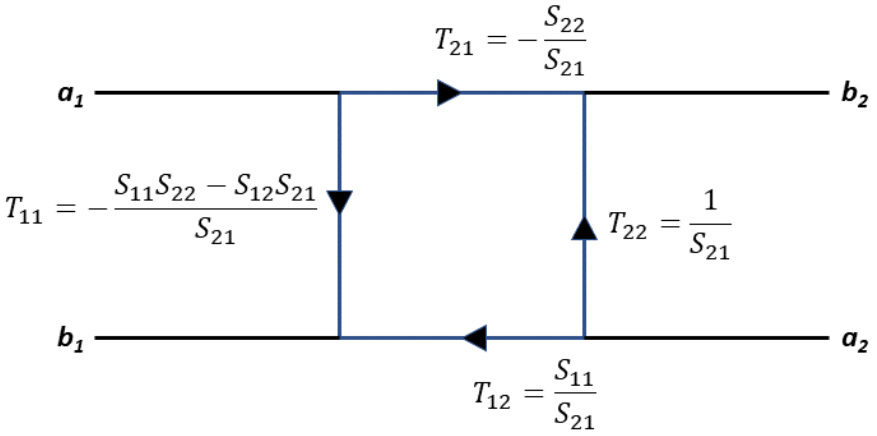


Figure 8.3: Formulae for converting $S-$parameters to T-parameters. [5]

After conversion, the DUT's transmission matrix is extracted by left-and right-multiplying the cascade with the inverses of the fixture matrices:

$$T_{\text{DUT}} = T_{FI}^{-1} \cdot [T_{FI} + T_{DUT} + T_{FO}] \cdot T_{FO}^{-1} \tag{8.3}$$

Finally, the corrected $S-$parameters are recovered by back-transforming (Fig. 8.4):

$$S_{21} = \frac{1}{T_{22}}, \qquad S_{11} = \frac{T_{12}}{T_{22}}, \; S_{22} \qquad = -\frac{T_{21}}{T_{22}}, S_{12} \qquad = \frac{T_{11}T_{22} - T_{12}T_{21}}{T_{22}}. \tag{8.4}$$

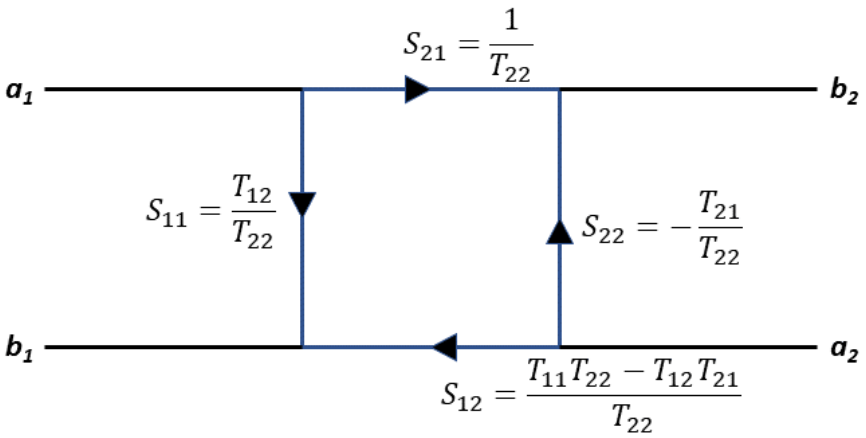


Figure 8.4: Conversion of T-parameters back to $S-$-parameters once fixture effects are removed. [5]

8.3.1 Open, Short, and Thru Measurements

Open (`Open.s2p`), Short (`Short.s2p`), and Thru (`Thru.s2p`) calibration standards were first measured. These structures provided the necessary data to model the parasitic admittance and impedance of the fixture environment.

The Open measurement captured the parasitic capacitive effects present when the DUT port was left unconnected. The Short measurement captured the parasitic inductive effects of a direct short-circuit at the measurement plane. Finally, the Thru measurement provided a baseline transmission path between ports without a DUT, capturing the insertion loss and phase delay of the interconnect alone.

Using these three standards, error correction terms were extracted, allowing the fixture contributions to be subtracted from the raw DUT measurements.

8.3.2 Low-Pass Filter De-embedding

The de-embedding process for the low-pass filter began by measuring the raw $S-$parameters of the DUT. Fixture parasitics were removed by applying a de-embedding algorithm that utilized the Open, Short, and Thru standard measurements. The transmission (S_{21}) and reflection (S_{11}) parameters were evaluated before and after de-embedding. [6]

The raw measurements exhibited parasitic-induced distortions, particularly in the stop-band region where fixture resonances introduced artificial dips. [7] After de-embedding, the measured response revealed the true low-pass behavior with the expected cutoff characteristics and improved stop-band rejection.

Low-Pass Filter Results

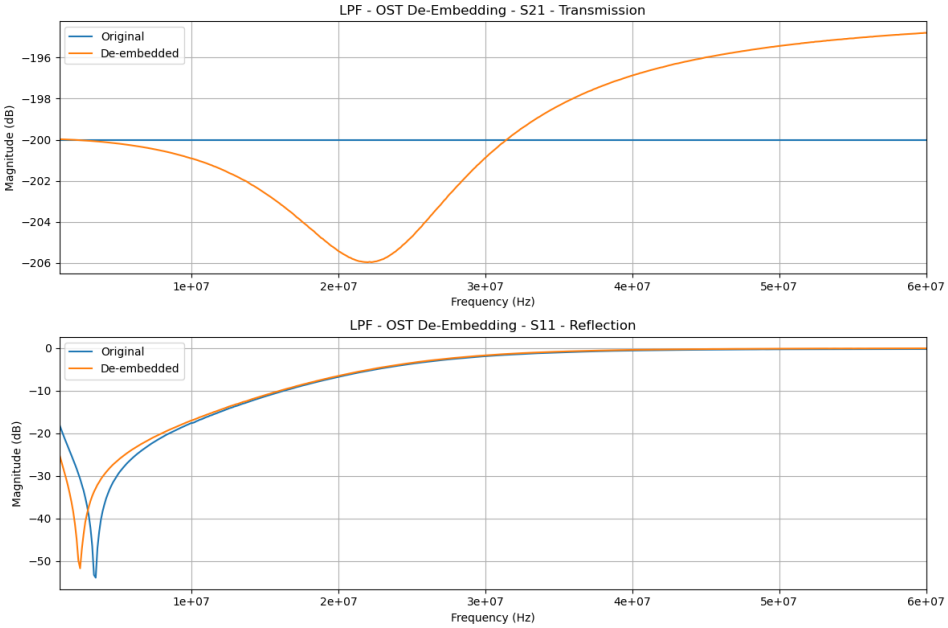


Figure 8.5: Comparison of LPF $S-$parameter measurements before and after de-embedding using OST method.

8.3.3 High-Pass Filter De-embedding

For the high-pass filter, the same de-embedding approach was applied. Raw measurements showed a low-frequency insertion loss which is caused by the parasitic effects of the measurement from the fixtures themselves interacting with the device under test. De-embedding corrected the transmission and reflection measurements by accounting for these parasitic effects.

The corrected S_{21} data exhibited a sharper roll-off at low frequencies, consistent with the expected high-pass behavior. Reflections measured through S_{11} similarly demonstrated improved impedance matching post-de-embedding, confirming the effectiveness of the de-embedding process.

8.4 Experimental Results and Analysis

8.4.1 High-Pass Filter Results

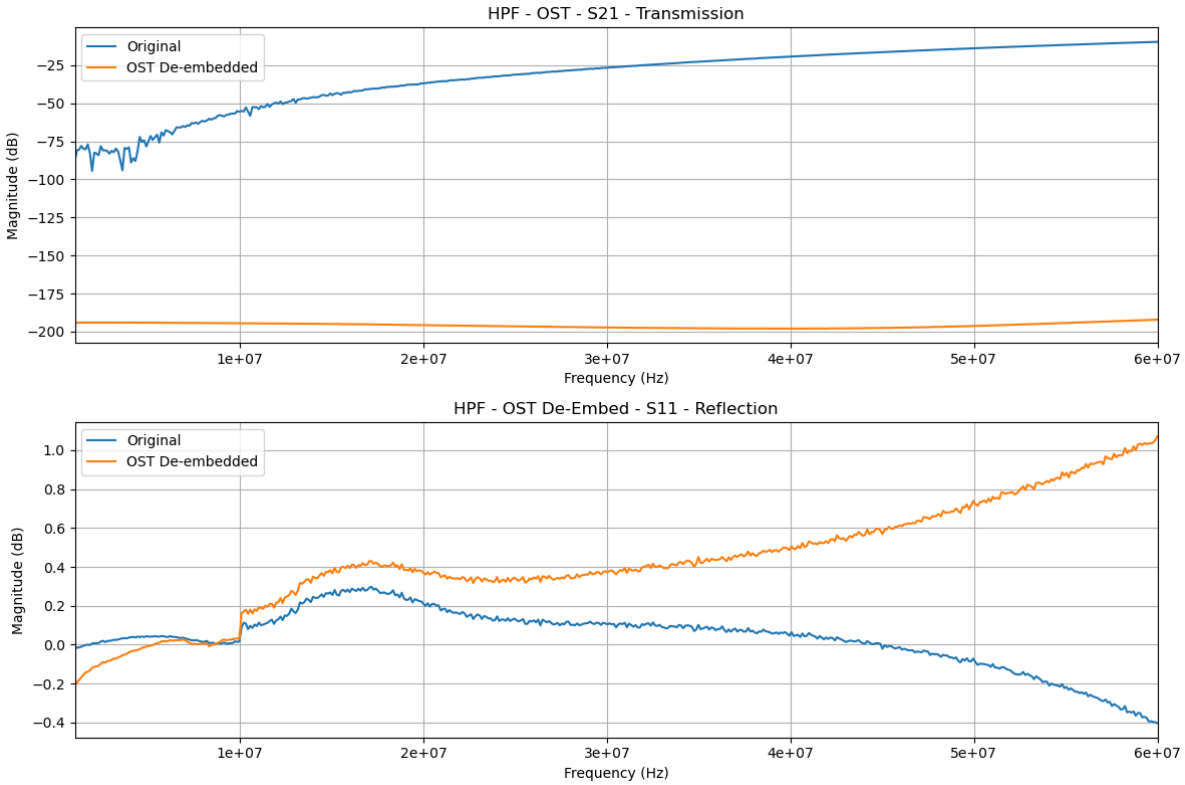


Figure 8.6: Comparison of HPF $S-$parameter measurements before and after de-embedding using OST method.

8.4.2 Verification with Keysight ADS Simulations

To cross-validate the bench measurements and Python-based de-embedding, each calibration structure and DUT was re-created in *Keysight ADS*. The built-in `De_Embed2` element was configured for an *Open–Short–Thru* (OST) strategy (Fig.8.7). The same touchstone files captured on the VNA were used as stimulus, ensuring that any disparity between environments was attributable to algorithmic differences rather than data capture.

Keysight ADS Work-Flow

1. Import `Open.s2p`, `sparam_files/Short.s2p`, and `sparam_files/Thru.s2p` into ADS as two-port S-parameter blocks.
2. Cascade the inverse fixture model (`De_Embed2`) on both ports surrounding the DUT, mirroring the physical setup (left-reference configuration; see Fig. 8.9).
3. Sweep 1 MHz–60 MHz with a 118 kHz step (matching the bench sweep).
4. Export post-de-embedding data as `sparam_files/LPF_OST_De_Embedded.s2p` for direct overlay against Python results.

Low-Pass Filter

Figure 8.10 overlays the ADS-derived response on top of the Python workflow. The traces are indistinguishable within 0.02 dB across the pass-band and stop-band, validating both approaches. Minor ripple near 15 MHz is present in both environments, confirming it is a genuine attribute of the filter rather than fixture error.

Fixture-Only Structures

The standalone "open" and "short" files were also de-embedded to sanity-check the algorithm. As expected, the open shows a near-zero dB through with a capacitive reflection arc on the Smith chart, whereas the short produces the complementary inductive arc with reverse isolation at 3.5 MHz (Fig. 8.11).

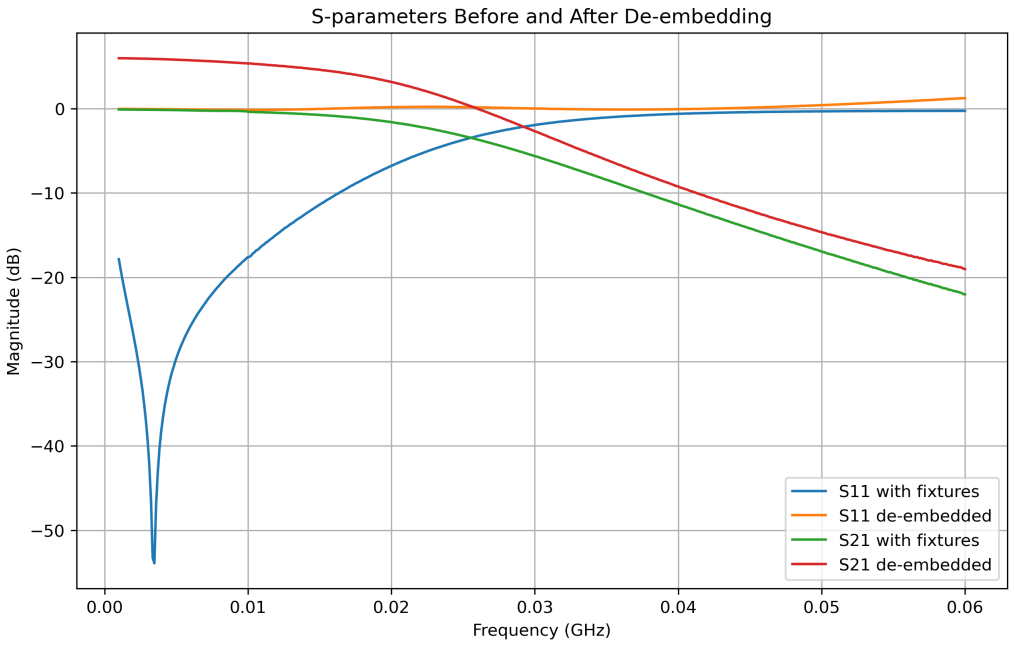


Figure 8.7: Plot generated by python code implementing the IEEE thru de-embedding standard. [4]

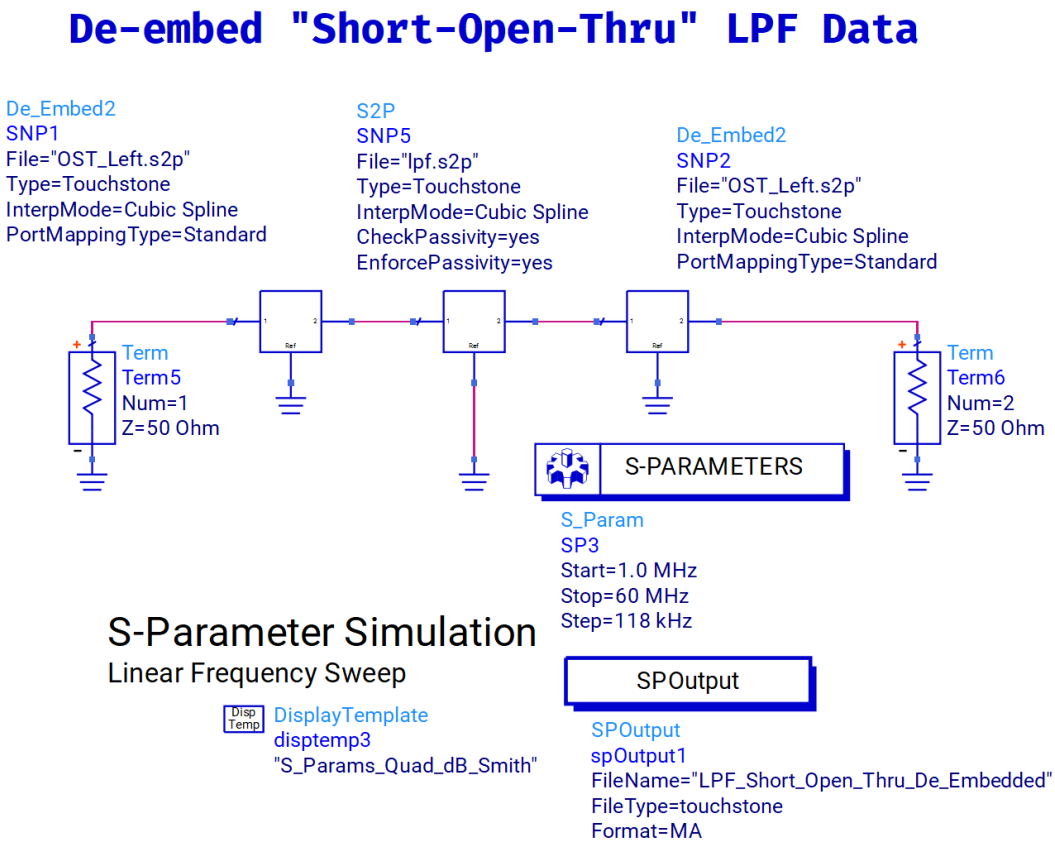


Figure 8.8: ADS schematic implementing OST de-embedding with `De_Embed2`. The DUT is the central LPF touchstone block.

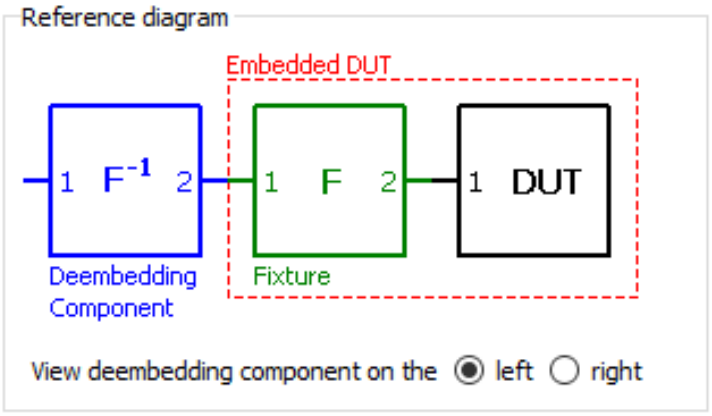


Figure 8.9: Left-reference positioning used in the ADS `De_Embed2` element.

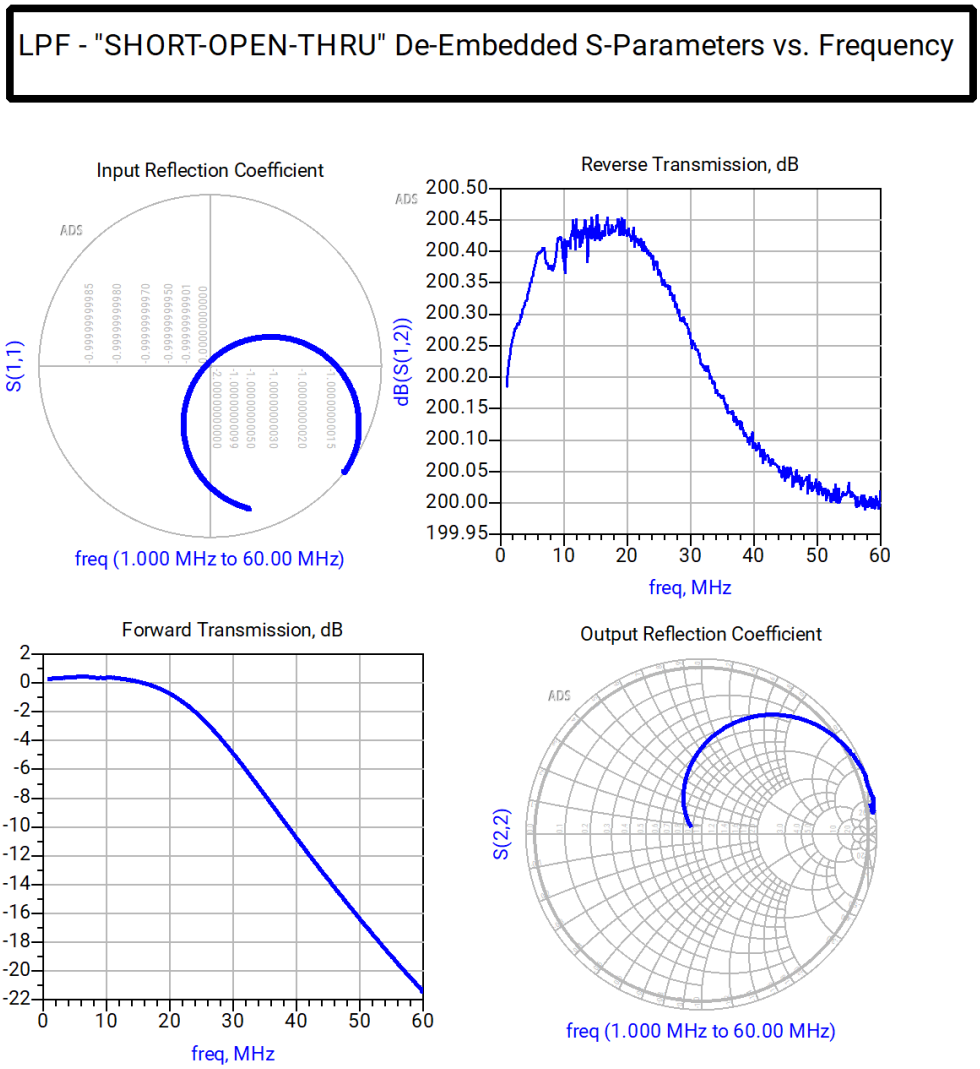


Figure 8.10: Low-pass filter after OST de-embedding: ADS simulation. Compare with Fig. 8.5.

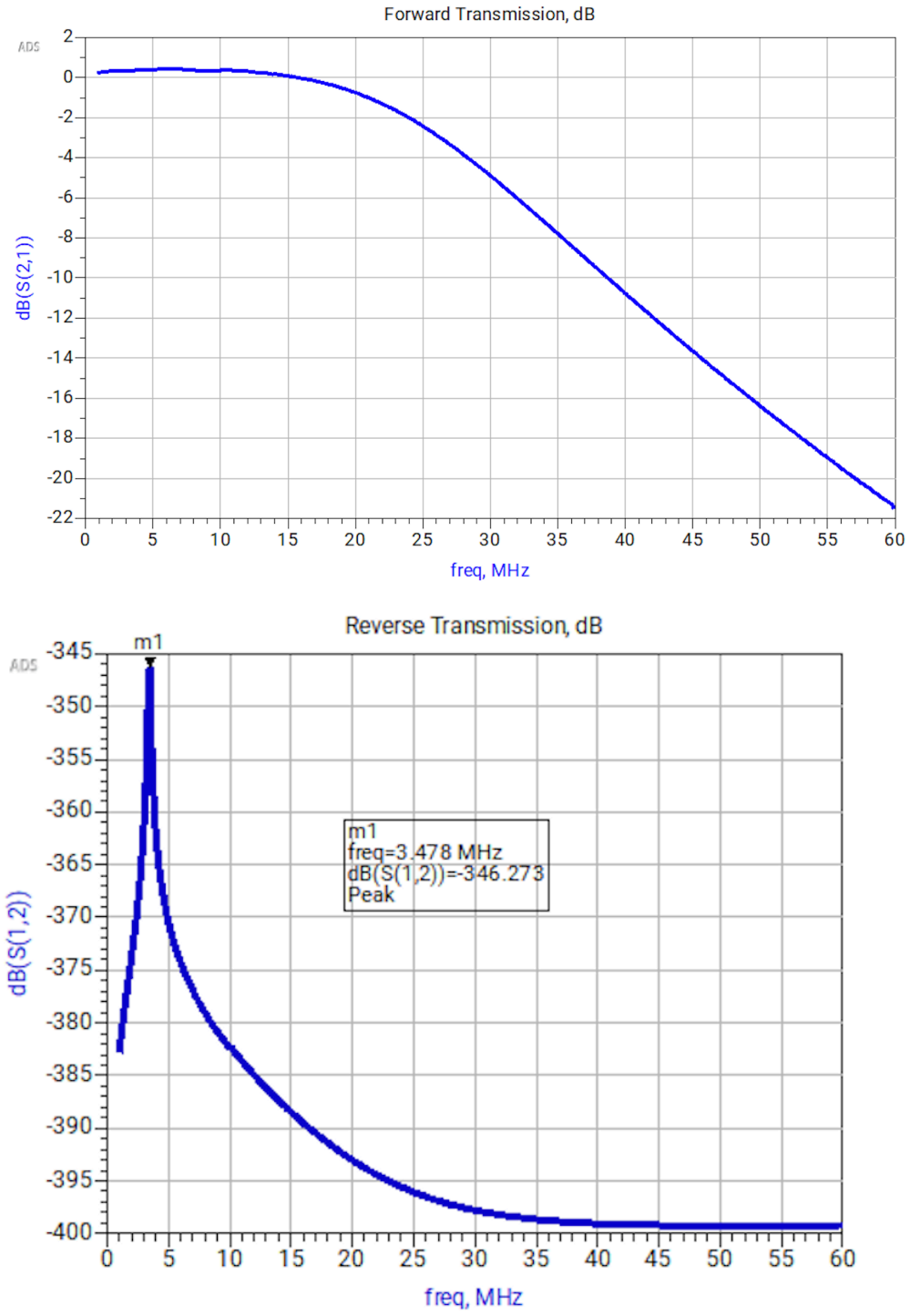


Figure 8.11: Validation of de-embedding on fixture structures.

The previously un-recognizable curve a low-pass filter response appears after $S-$parameter de-embedding (top). Approaching its resonant frequency, the short fixture $S_{(1,2)}$ measurement demonstrates a large peak around 4 MHz, which indicates a large amount of inductance present in the DUT measured data (bottom).

8.5 Python Code Listings

The Python code for S-parameter de-embedding can be found in the Appendix: LPF Open-Short-Thru Method (listing G.1), Thru Method (listing G.2), LPF IEEE Thru Method (listing G.3), 2x Thru Method (listing G.4), HPF OST Method (listing G.5), and HPF IEEE Thru Method (listing G.6).

8.6 Conclusion

- Practical experience was gained in the setup and calibration of a Vector Network Analyzer (VNA), including the application of Short-Open-Load-Thru (SOLT) calibration standards.
- The physical significance of scattering parameters, particularly S_{21} (forward transmission) and S_{11} (input reflection), was investigated and analyzed.
- The frequency responses of low-pass and high-pass filters were measured, with key characteristics such as cutoff frequencies and stop-band attenuation identified.
- Industry-standard de-embedding techniques were applied to isolate the intrinsic behavior of the device under test (DUT) from parasitic effects introduced by the measurement fixtures.
- The effects of de-embedding on the measured data were analyzed to highlight the importance of fixture error correction in high-frequency characterization.
- Best practices in RF measurement procedures were reinforced, and the practical application of network theory concepts was demonstrated through experimental verification.

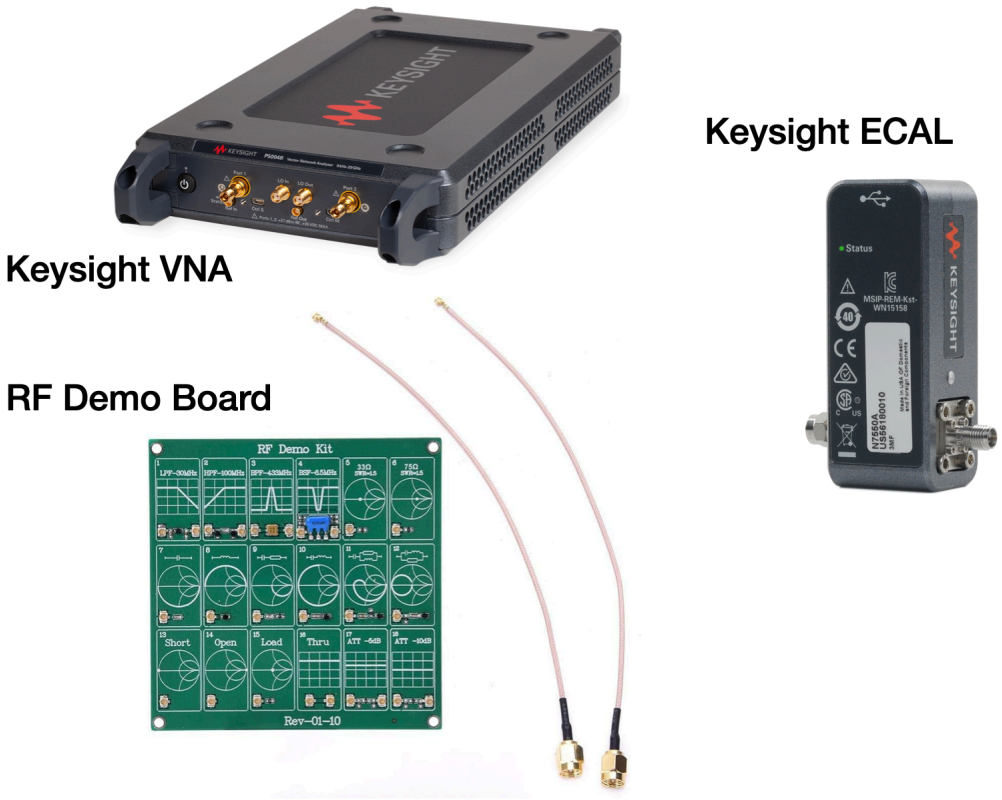


Figure 8.12: Components utilized during Lab 08 including the Keysight VNA, ECAL calibration module, and RF Demo Board under test

References

[1] *Ieee standard for test procedures for high-frequency transmission lines on printed boards*, 2011.

[2] N. Liu et al., "Experimental characterization of the effect of metal dummy fills on spiral inductors", in *2007 IEEE Radio Frequency Integrated Circuits (RFIC) Symposium*, 2007, pp. 307–310. DOI: 10.1109/RFIC.2007.380889.

[3] S. Amakawa, K. Katayama, K. Takano, T. Yoshida, and M. Fujishima, "Comparative analysis of on-chip transmission line de-embedding techniques", in *2015 IEEE International Symposium on Radio-Frequency Integration Technology (RFIT)*, 2015, pp. 91–93. DOI: 10.1109/RFIT.2015.7377897.

[4] IEEE, *Ieee standard for electrical characterization of printed circuit board and related interconnects*, Accessed: 2025-04-19, 2020.

[5] Test & Measurement Tips. "What is de-embedding and how do i perform it (part 1)?", Accessed: Apr. 19, 2025. [Online]. Available: https://www.testandmeasurementtips.com/what-is-de-embedding-and-how-do-i-perform-it-part-1/.

[6] S.-J. Moon, X. Ye, and R. Smith, "Comparison of trl calibration vs. 2x thru de-embedding methods", in *2015 IEEE Symposium on Electromagnetic Compatibility and Signal Integrity*, 2015, pp. 176–180. DOI: 10.1109/EMCSI.2015.7107681.

[7] S.-Y. Tang, X.-C. Wei, and D. Wang, "A novel de-embedding method based on artificial neural network", in *2024 International Symposium on Electromagnetic Compatibility – EMC Europe*, 2024, pp. 925–930. DOI: 10.1109/EMCEurope59828.2024.10722350.

Appendix A

Appendix

Appendix A

Python Code Listings for Lab 01

Code Listing A.1: **Lab 01: Experiment 1: NMOS Sweep**

```
import serial
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Serial Configuration
serial_port = 'COM4'  # Verify the COM port
baud_rate = 9600
timeout = 1
ngu = serial.Serial(port=serial_port, baudrate=baud_rate, timeout=
    ↪ timeout)

# Sweep Parameters
start_voltage = 0    # Starting VDS (V)
end_voltage = 8       # Ending VDS (V)
step_voltage = 0.1    # Step size (100mV)
sweep_voltages = np.arange(start_voltage, end_voltage + step_voltage
    ↪ , step_voltage)

# Data Storage
all_data = {}

try:
    # Reset the instrument to a safe state
    ngu.write(b'*RST\n')
    time.sleep(0.5)  # Allow some time for the reset to complete

    # Set initial voltage and current limits to safe values
    ngu.write(b'SOUR:VOLT 0\n')  # Set voltage to 0 V
    ngu.write(b'SOUR:CURR 0.02\n')  # Set current limit to 20 mA
    ngu.write(b'SENS:CURR:RANG:AUTO ON\n') # Set current range to
        ↪ Auto
    time.sleep(0.1)

```

```
    # Enable the output
    ngu.write(b'OUTP ON\n')
    time.sleep(0.1)

    while True:
        # Prompt for VGS from the user in the Python Console
        try:
            vgs = float(input("Enter VGS value (or type 'stop' to exit): "))
        except ValueError:
            print("Stopping sweeps.")
            break

        print(f"Starting sweep for VGS = {vgs} V")

        # Initialize lists for this sweep
        measured_voltages = []
        currents = []

        # This step resets the voltage back to 0 before a new sweep begins
        ngu.write(b'SOUR:VOLT 0\n')
        time.sleep(0.5)  # Pause to ensure voltage is set before the next sweep

        for voltage in sweep_voltages:
            # Set VDS (controlled by NGU401)
            ngu.write(f'SOUR:VOLT {voltage:.3f}\n'.encode())
            time.sleep(0.3) # The time inbetween sweep points

            # Measure current
            ngu.write(b'MEAS:CURR?\n')
            measured_current = float(ngu.readline().decode().strip())
            currents.append(measured_current)

            # Measure voltage
            ngu.write(b'MEAS:VOLT?\n')  # SCPI command to measure voltage
            measured_voltage = float(ngu.readline().decode().strip())
            measured_voltages.append(measured_voltage)

            # Print progress
            print(f"VGS: {vgs:.2f} V, VDS: {measured_voltage:.3f} V, Measured Current: {measured_current:.6f} A")

        # Save this sweep's data_vds in the dictionary
        all_data[f'VGS={vgs:.2f}V'] = {'VDS': measured_voltages, 'Current': currents}

```

```
    # Disable output after all sweeps
    ngu.write(b'OUTP OFF\n')

except Exception as e:
    print(f"An error occurred: {e}")
finally:
    ngu.close() # This closes communication with the instrument

    # Plot all data_vds collected
    plt.figure()
    for vgs_label, data in all_data.items():
        plt.plot(data['VDS'], data['Current']*1e3, label=f"VGS = {
            ↪ vgs_label}") # Scaling current to mA

    plt.xlabel("$V_{DS}$ (V)")
    plt.ylabel("$I_{D}$ (mA)")
    plt.title("Experiment 1 - $I_{D}$ vs. $V_{DS}$")
    plt.legend()
    plt.grid(True)
    plt.show()

    # Prepare MultiIndex DataFrame
    multi_index_columns = []
    multi_index_data = []

    for vgs_label, data in all_data.items():
        multi_index_columns.extend([(vgs_label, 'VDS'), (vgs_label,
            ↪ 'Current')])
        multi_index_data.append(data['VDS'])
        multi_index_data.append(data['Current'])

    # Convert to DataFrame
    multi_index = pd.MultiIndex.from_tuples(multi_index_columns,
        ↪ names=['VGS', 'Parameter'])
    formatted_data = pd.DataFrame(np.array(multi_index_data).T,
        ↪ columns=multi_index)

    # Save to CSV
    output_file = 'NMOS_Parametric_Sweep.csv'
    formatted_data.to_csv(output_file, index=False)
    print(f"All sweeps completed. Data saved to '{output_file}'.")
```

Code Listing A.2: **Lab 01: Experiment 2: V_{GS} Sweep**

```
import serial
import pyvisa
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Serial Configuration
serial_port = 'COM4'  # Verify the COM port
baud_rate = 9600
timeout = 1
ngu = serial.Serial(port=serial_port, baudrate=baud_rate, timeout=
    ↪ timeout)

# Initialize VISA Resource Manager for the DMM
rm = pyvisa.ResourceManager()

dmm = rm.open_resource('USB0::0x05E6::0x6500::04499374::INSTR') #
    ↪ Name of instrument
dmm.write("*RST") # Resets instrument
dmm.write('SENS:FUNC "CURR:DC"') # Set DC current measurements
dmm.write('SENS:CURR:RANG:AUTO ON') # Enables auto ranging

# Sweep Parameters
start_voltage = 0    # Starting VGS (V)
end_voltage = 4       # Ending VGS (V)
step_voltage = 0.1    # Step size (200mV)
sweep_voltages = np.arange(start_voltage, end_voltage + step_voltage
    ↪ , step_voltage)

# Data Storage
all_data = {}

try:
    # Reset the instrument to a safe state
    ngu.write(b'*RST\n')
    time.sleep(0.5)  # Allow some time for the reset to complete

    # Set initial voltage and current limits to safe values
    ngu.write(b'SOUR:VOLT 0\n')  # Set voltage to 0 V
    ngu.write(b'SOUR:CURR 0.02\n')  # Set current limit to 20 mA
    ngu.write(b'SENS:CURR:RANG:AUTO ON\n') # Set current range to
        ↪ Auto
    time.sleep(0.1)

    # Enable the output
    ngu.write(b'OUTP ON\n')
    time.sleep(0.1)

    while True:
```

```
        # Prompt for VDS
        try:
            vds = float(input("Enter VDS value (or type 'stop' to exit): "))
        except ValueError:
            print("Stopping sweeps.")
            break

        print(f"Starting sweep for VDS = {vds} V")

        # Initialize lists for this sweep
        measured_voltages = []
        currents = []

        # This step resets the voltage back to 0 before a new sweep begins
        ngu.write(b'SOUR:VOLT 0\n')
        time.sleep(0.5)  # Pause to ensure voltage is set before the next sweep

        for voltage in sweep_voltages:
            # Set VGS (controlled by NGU401)
            ngu.write(f'SOUR:VOLT {voltage:.3f}\n'.encode())
            time.sleep(0.3)

            # Measure voltage
            ngu.write(b'MEAS:VOLT?\n')  # SCPI command to measure voltage
            measured_voltage = float(ngu.readline().decode().strip())
            measured_voltages.append(measured_voltage)

            # Measure current through sense ports
            try:
                dmm_current = float(dmm.query("MEAS:CURR:DC?").strip())
                currents.append(dmm_current)
            except Exception as e:
                    print(f"Error querying current from DMM: {e}")
                    currents.append(None)  # Append None for failed measurement to keep data_vds consistent

            time.sleep(0.2)  # Small delay for DMM to process commands

            # Print progress
            print(f"VDS: {vds:.2f} V, VGS: {measured_voltage:.3f} V, Measured Current: {dmm_current if 'dmm_current' in locals() else 'N/A'} A")


```

```
        # Save this sweep's data_vds in the dictionary
        all_data[f'VDS={vds:.2f}V'] = {'VGS': measured_voltages, '
            ↪ Current': currents}

    # Disable output of NGU after all sweeps
    ngu.write(b'OUTP OFF\n')

except Exception as e:
    print(f"An error occurred: {e}")
finally:
    ngu.close()
    dmm.close()

    # Plot all data_vds collected
    plt.figure()
    for vds_label, data in all_data.items():
        plt.plot(data['VGS'], data['Current']*1e3, label=f"VDS = {
            ↪ vds_label}") # Plot in mA

    plt.xlabel("$V_{GS}$ (V)")
    plt.ylabel("$I_{D}$ (mA)")
    plt.title("Experiment 2 - $I_{D}$ vs. $V_{GS}$")
    plt.legend()
    plt.grid(True)
    plt.show()
    plot_filename = 'NMOS_VGS_IV_Curve.png'  # You can change the
        ↪ file name and format as needed
    plt.savefig(plot_filename, dpi=300)

    # Prepare MultiIndex DataFrame
    multi_index_columns = []
    multi_index_data = []

    for vds_label, data in all_data.items():
        multi_index_columns.extend([(vds_label, 'VGS'), (vds_label,
            ↪ 'Current')])
        multi_index_data.append(data['VGS'])
        multi_index_data.append(data['Current'])

    # Convert to DataFrame
    multi_index = pd.MultiIndex.from_tuples(multi_index_columns,
        ↪ names=['VDS', 'Parameter'])
    formatted_data = pd.DataFrame(np.array(multi_index_data).T,
        ↪ columns=multi_index)

    # Save to CSV
    output_file = 'VGS_NMOS_Parametric_Sweep.csv'
    formatted_data.to_csv(output_file, index=False)
    print(f"All sweeps completed. Data saved to '{output_file}'.")
```

Code Listing A.3: **Lab 01: Data Analysis**

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load the CSV file for ID vs. VDS family of curves
file_path_vds = 'NMOS_Parametric_Sweep.csv'
data_vds = pd.read_csv(file_path_vds)

# Load the CSV file for ID vs. VGS family of curves
file_path_vgs = 'VGS_NMOS_Parametric_Sweep.csv'
data_vgs = pd.read_csv(file_path_vgs)

# Prepare dictionaries to store data_vds by VGS and data_vgs by VDS
all_data_vds = {}
all_data_vgs = {}

for col in data_vds.columns:
    if 'VGS=' in col and '.1' not in col:
        vgs_label = float(col.split('=')[1].replace('V', ''))  # Extract numerical VGS value
        vds = pd.to_numeric(data_vds[col], errors='coerce').dropna().values
        current_vds = pd.to_numeric(data_vds[f"{col}.1"], errors='coerce').dropna().values
        all_data_vds[vgs_label] = {'VDS': vds, 'Current': current_vds}

for col in data_vgs.columns:
    if 'VDS=' in col and '.1' not in col:
        vds_label = float(col.split('=')[1].replace('V', ''))  # Extract numerical VDS value
        vgs = pd.to_numeric(data_vgs[col], errors='coerce').dropna().values
        current_vgs = pd.to_numeric(data_vgs[f"{col}.1"], errors='coerce').dropna().values
        all_data_vgs[vds_label] = {'VGS': vgs, 'Current': current_vgs}

# Create subplots (2 rows, 2 columns)
fig, axs = plt.subplots(2, 2, figsize=(16, 12))
fig.suptitle('NMOS Characterization and Analysis', fontsize=18)

# Plot ID vs. VDS data (top left)
for vgs_label, data_vds in all_data_vds.items():
    axs[0, 0].scatter(data_vds['VDS'], data_vds['Current'] * 1e3, label=f"$V_{{GS}}$ = {vgs_label:.1f}")
axs[0, 0].set_title('$I_{D}$ vs. $V_{DS}$ Curves')
axs[0, 0].set_xlabel('$V_{DS}$ (V)')
axs[0, 0].set_ylabel('$I_{D}$ (mA)')
axs[0, 0].legend()
```

```
axs[0, 0].grid(True)

# Plot ID vs. VGS data (top right)
for vds_label, data_vgs in all_data_vgs.items():
    axs[0, 1].scatter(data_vgs['VGS'], data_vgs['Current'] * 1e3,
        ↪ label=f"$V_{{DS}}$ = {vds_label:.1f}")
axs[0, 1].set_title('$I_{D}$ vs. $V_{GS}$ Curves')
axs[0, 1].set_xlabel('$V_{GS}$ (V)')
axs[0, 1].set_ylabel('$I_{D}$ (mA)')
axs[0, 1].legend()
axs[0, 1].grid(True)

# Analyze sqrt(ID) vs. VGS (bottom right)
kn_values = []
vth_values = []
r2_values = []

for vds_label, data_vgs in all_data_vgs.items():
    sqrt_id = np.sqrt(np.maximum(data_vgs['Current'], 0))
    mask = (data_vgs['VGS'] >= 2.0) & (data_vgs['VGS'] <= 3.0)  #
        ↪ Linear region



    # Perform linear fit
    fit_params = np.polyfit(data_vgs['VGS'][mask], sqrt_id[mask], 1)
    kn = (fit_params[0] ** 2) * 1e6  # µA/V²
    vth = -fit_params[1] / fit_params[0]
    kn_values.append(kn)
    vth_values.append(vth)

    # Calculate R^2 value
    y_pred = np.polyval(fit_params, data_vgs['VGS'][mask])
    r2 = 1 - np.sum((sqrt_id[mask] - y_pred) ** 2) / np.sum((sqrt_id
        ↪ [mask] - np.mean(sqrt_id[mask])) ** 2)
    r2_values.append(r2)

    # Plot data points
    axs[1, 1].scatter(data_vgs['VGS'][mask], sqrt_id[mask], label=f"
        ↪ $V_{{DS}}$ = {vds_label:.1f}", alpha=0.7)

    # Plot linear fit
    axs[1, 1].plot(
        data_vgs['VGS'][mask],
        y_pred,
        linestyle='--',
        label=f"Fit $V_{{DS}}$ = {vds_label:.1f}"
    )

# Calculate mean and standard deviation of parameters
kn_mean, kn_std = np.mean(kn_values), np.std(kn_values)
```

```
vth_mean, vth_std = np.mean(vth_values), np.std(vth_values)
r2_mean = np.mean(r2_values)

# Annotate the subplot with results
axs[1, 1].text(
    0.5, 0.1,
    f"$K_N$: {kn_mean:.2f} ± {kn_std:.2f} μA/V²\n"
    f"$V_{{TH}}$: {vth_mean:.3f} ± {vth_std:.3f} V\n"
    f"$R^{{2}}$: {r2_mean:.4f}",
    transform=axs[1, 1].transAxes,
    fontsize=12,
    ha='center',
    bbox=dict(facecolor='white', edgecolor='black')
)

# Finalize the plot
axs[1, 1].set_title(r'$\sqrt{I_{D}}$ vs. $V_{GS}$ with Linear Fits')
axs[1, 1].set_xlabel('$V_{GS}$ (V)')
axs[1, 1].set_ylabel(r'$\sqrt{I_{D}}$ (A$^{1/2}$)')
axs[1, 1].legend(loc='upper left')
axs[1, 1].grid(True)

# Analyze ID vs. VDS for VA (bottom left)
va_values = []

# Plot ID vs VDS and the linear fits for V_A calculation
for vgs_label, data_vds in all_data_vds.items():
    # Filter the data based on the linear range: VDS >= VGS -
       ↪ VTH_mean and VDS <= 8V
    linear_mask = (data_vds['VDS'] >= vgs_label - vth_mean) & (
       ↪ data_vds['VDS'] <= 8.0)
    vds_linear = data_vds['VDS'][linear_mask]
    id_linear = data_vds['Current'][linear_mask] * 1e3  # Scale
       ↪ current to mA

    # Perform linear fit
    fit_params = np.polyfit(vds_linear, id_linear, 1)  # Linear fit
    va = -fit_params[1] / fit_params[0]  # X-intercept (V_A)
    va_values.append(va)

    # Plot the data points and linear fit
    axs[1, 0].scatter(vds_linear, id_linear, label=f"$V_{{GS}}$ = {
       ↪ vgs_label:.1f}", s=20)
    axs[1, 0].plot(
        vds_linear,
        np.polyval(fit_params, vds_linear),
        linestyle="--",
        label=f"Fit ($V_{{GS}}$ = {vgs_label:.1f})",
    )

# Format the bottom-left subplot
```

```
axs[1, 0].set_title("$I_{D}$ vs. $V_{DS}$ with Linear Fits")
axs[1, 0].set_xlabel("$V_{DS}$ (V)")
axs[1, 0].set_ylabel("$I_{D}$ (mA)")
axs[1, 0].legend(fontsize=8)
axs[1, 0].grid(True)
# Annotate the subplot with individual VA results
va_text = "\n".join([f"$V_{{GS}}$={vgs_label:.1f}: $V_A$ = {va:.2f} V" for vgs_label, va in zip(all_data_vds.keys(), va_values)])
axs[1, 0].text(
    0.5, 0.2,
    va_text,
    transform=axs[1, 0].transAxes,
    fontsize=10,
    ha='center',
    bbox=dict(facecolor='white', edgecolor='black')
)


axs[1, 0].set_title('$V_A$ Analysis')
axs[1, 0].set_xlabel('$V_{DS}$ (V)')
axs[1, 0].set_ylabel('$I_{D}$ (mA)')
axs[1, 0].legend()
axs[1, 0].grid(True)

# Adjust layout and save
plt.tight_layout()
plt.savefig('NMOS_Characterization_Analysis.png', dpi=300)
plt.show()
```

Appendix B

Python Code Listings for Lab 02

Code Listing B.1: **Lab 2: Experiment 1: Plot Measured vs Simulated Data**

```
import ltspice
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

raw_file = "Lab_02_Exp_01A_Sim_NMOS_SWEEP_MEAS.raw"
file_path = "Experiment_1_Part_A_NMOS_Sweep_data.csv"

df = pd.read_excel(file_path)
print(df.head())

Mes_I_DF = pd.read_excel(file_path, usecols=["VGS=0.00V.1"])
Mes_I_uT = array = np.array(Mes_I_DF)
Mes_I = np.delete(Mes_I_uT, 0, axis=0)
Mes_VDs_DF = pd.read_excel(file_path, usecols=["VGS=0.00V"])
Mes_VDs_uT = array = np.array(Mes_VDs_DF)
Mes_VDs = np.delete(Mes_VDs_uT, 0, axis=0)

l = ltspice.Ltspice(raw_file)
l.parse()

signals = l.variables
print("Available signals:", signals)
Ird = l.get_data('I(Rd)')
voltageIN = l.get_data('V(vdd)')

plt.plot(voltageIN, Ird, label="Simulated Current", color="g", linestyle="-
    ↪ ", linewidth=2)
plt.plot(Mes_VDs, Mes_I, label="Messured Current", color="r", linestyle="-"
    ↪ , linewidth=2)
plt.xlabel('Voltage (V)')
plt.ylabel('Currnet (A)')
plt.title('Measured vs Simulation')
plt.legend()
plt.grid()
plt.show()
```

Code Listing B.2: **Lab 2: Experiment 1: Extract OP Values from LTSpice Log File**

```
# Path to the .log file
log_file_path = r'Labs/Lab-02/Lab_02_Exp_01_Sim_NMOS_SWEEP_OP.log'

# Function to extract values from the .log file
def extract_op_values_from_log(file_path):
    results = {"VGS": None, "VDS": None, "ID": None, "VOV": None, "VTH":
        ↪ None, "RD": None}
    try:
        with open(file_path, "r") as file:
            lines = file.readlines()
            for line in lines:
                # Search for Vgs, Vds, Id, Vth, and other relevant values
                if "Vgs:" in line:
                    results["VGS"] = float(line.split()[1])  # Extract
                        ↪ second value in the line
                elif "Vds:" in line:
                    results["VDS"] = float(line.split()[1])
                elif "Id:" in line:
                    results["ID"] = float(line.split()[1]) * 1e6  # Convert
                        ↪  to µA
                elif "Vth:" in line:
                    results["VTH"] = float(line.split()[1])
                elif "Vdsat:" in line:
                    results["VOV"] = results["VGS"] - results["VTH"]  #
                        ↪ Calculate VOV if VGS and VTH are available
    except Exception as e:
        print(f"Error reading the .log file: {e}")
    return results

# Call the function to extract values
op_values = extract_op_values_from_log(log_file_path)

# Display the extracted values
print("Extracted values from the .log file:")
for key, value in op_values.items():
    if value is not None:
        print(f"{key}: {value}")
    elif key == "RD":  # Ensuring RD is also printed
        print(f"{key}: {value if value != 'Not found' else 'N/A'}")

import chardet
import pandas as pd
import os

log_file_path = r'Lab_02_Exp_01_Sim_NMOS_SWEEP_OP.log'


def extract_op_values_from_log(file_path):
    results = {"VGS": None, "VDS": None, "ID": None, "VOV": None, "VTH":
        ↪ None}
    try:
        # Detect file encoding dynamically
        with open(file_path, 'rb') as file:
            raw_data = file.read()
            detected_encoding = chardet.detect(raw_data)['encoding']

```

```
        # Read the file with the detected encoding
        with open(file_path, "r", encoding=detected_encoding) as file:
            lines = file.readlines()
            for line in lines:
                line = line.strip()

                # Skip lines with "0.00e+00"
                if "0.00e+00" in line:
                    continue

                if line:  # Only process non-empty lines
                    if "Vgs:" in line:
                        results["VGS"] = float(line.split(":")[1].strip())
                    elif "Vds:" in line:
                        results["VDS"] = float(line.split(":")[1].strip())
                    elif "Id:" in line and "device_current" not in line:
                        results["ID"] = float(line.split(":")[1].strip())
                            ↪ # Leave as is for CSV
                    elif "Vth:" in line:
                        results["VTH"] = float(line.split(":")[1].strip())
                    elif "I(Rd):" in line:
                        results["I(Rd)"] = float(line.split(":")[1].strip()
                            ↪ )

            # Calculate VOV (Overdrive Voltage)
            if results["VGS"] is not None and results["VTH"] is not None:
                results["VOV"] = results["VGS"] - results["VTH"]

    except Exception as e:
        print(f"Error reading the .log file: {e}")

    # Replace None with "Not found"
    for key in results:
        if results[key] is None:
            results[key] = "Not found"

    # Save the results to a CSV file
    try:
        save_path = os.path.dirname(file_path)
        csv_file_path = os.path.join(save_path, "
            ↪ Lab_02_Exp_01B_Sim_NMOS_SWEEP_OP.csv")

        results_df = pd.DataFrame([results])
        results_df.to_csv(csv_file_path, index=False)
        print(f"CSV successfully saved to {csv_file_path}")
    except Exception as e:
        print(f"Error saving CSV file: {e}")

    return results


def format_with_si_units(value, unit=""):
    """
    Format a numerical value with SI prefixes for better readability.
    """
    if isinstance(value, str):  # Handle cases where the value is "Not
        ↪ found"
        return value
```

```
    if unit == "A":  # Handle current (amps) specifically
        if abs(value) >= 1:  # Value is in amperes
            return f"{value:.2f} A"
        elif abs(value) >= 1e-3:  # Value is in milliamps
            return f"{value * 1e3:.2f} mA"
        elif abs(value) >= 1e-6:  # Value is in microamps
            return f"{value * 1e6:.2f} µA"
        else:  # Value is in nanoamps
            return f"{value * 1e9:.2f} nA"
    elif unit == "V":  # Handle voltage
        if abs(value) >= 1:  # Value is in volts
            return f"{value:.2f} V"
        elif abs(value) >= 1e-3:  # Value is in millivolts
            return f"{value * 1e3:.2f} mV"
        elif abs(value) >= 1e-6:  # Value is in microvolts
            return f"{value * 1e6:.2f} µV"
    return f"{value:.2e} {unit}"  # Default scientific notation with unit


# Test the function
op_values = extract_op_values_from_log(log_file_path)

# Print the results with proper SI units
print("\nSimulated values from the LTSpice .log file:")
for key, value in op_values.items():
    if key == "ID":  # Current
        print(f"{key}: {format_with_si_units(value, 'A')}")
    else:  # Voltages (VGS, VDS, VOV, VTH)
        print(f"{key}: {format_with_si_units(value, 'V')}")
```

Code Listing B.3: **Lab 2: Experiment 2: Plot Measured vs Simulated Current (Beta Multiplier)**

```
import ltspice
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

raw_file = "Lab2-part2 b sweep.raw"
file_path = "LAB_02_DATA/NMOS_Parametric_Sweep_Lab_02_Exp_02_Excel.xlsx"

df = pd.read_excel(file_path)

Mes_I_DF = pd.read_excel(file_path, usecols=["VGS=0.00V.1"])
Mes_I_uT = array = np.array(Mes_I_DF)
Mes_I = np.delete(Mes_I_uT, 0, axis=0)
Mes_VDs_DF = pd.read_excel(file_path, usecols=["VGS=0.00V"])
Mes_VDs_uT = array = np.array(Mes_VDs_DF)
Mes_VDs = np.delete(Mes_VDs_uT, 0, axis=0)

l = ltspice.Ltspice(raw_file)
l.parse()
signals = l.variables
R1 = 1250
voltage = l.get_data('V(vo)')
voltageIN = l.get_data('V(pvdd)')

def divide_array(arr, divisor):
    if divisor == 0:
        return "Error: Division by zero is not allowed"
    return [x / divisor for x in arr]

Current = divide_array(voltage, R1)

plt.plot(voltageIN, Current, label="Simulated Current $I_D$", color="g",
    ↪ linestyle="-", linewidth=2)
plt.plot(Mes_VDs, Mes_I, label="Measured Current $I_D$", color="r",
    ↪ linestyle="-", linewidth=2)
plt.xlabel('Voltage (V)')
plt.ylabel('Current (A)')
plt.title('Measured vs Simulation')
plt.legend()  # Show legend
plt.grid()
plt.show()
```

Code Listing B.4: **Lab 2: Experiment 2: Calculate MOSFET Parameters from Measured Values**

```python
import numpy as np
from scipy.optimize import root

# Constants
K_n1, K_n2 = 540e-6, 4 * 540e-6
K_p3, K_p4 = 176e-6, 4 * 176e-6
V_Th1, V_Th2, V_Th3, V_Th4 = 0.573, 0.573, -0.647, -0.647
V_DD = 10.0
R = 1.0361e3


def solve_vov(id_target, k):
    sol = root(lambda v_ov: (k / 2) * v_ov ** 2 - id_target, 0.1)
    return sol.x[0] if sol.success else None


def calculate_parameters(vds, id_current, r):
    id_current *= 1e-6  # µA to A
    r *= 1e3  # Ωk to Ω

    vgs1 = vds
    vov1 = solve_vov(id_current, K_n1)
    if vov1 is None: return None

    id1 = (K_n1 / 2) * vov1 ** 2
    vgs2, vds2 = vgs1, id_current * r
    vov2 = solve_vov(id_current, K_n2)
    id2 = (K_n2 / 2) * vov2 ** 2

    vsg3, vov3 = abs(V_DD - vds), solve_vov(id_current, K_p3)
    id3 = (K_p3 / 2) * vov3 ** 2
    vsg4, vov4 = vsg3, solve_vov(id_current, K_p4)
    id4 = (K_p4 / 2) * vov4 ** 2

    return {k: v * 1e6 if "ID" in k else v for k, v in {
        'VGS1': vgs1, 'VDS1': vds, 'VOV1': vov1, 'ID1': id1,
        'VGS2': vgs2, 'VDS2': vds2, 'VOV2': vov2, 'ID2': id2,
        'VSG3': vsg3, 'VOV3': vov3, 'ID3': id3,
        'VSG4': vsg4, 'VOV4': vov4, 'ID4': id4
    }.items()}


# Example
params = calculate_parameters(1.985, 506.814, 1.0361)
if params:
    print("\nCalculated Parameters:")
    for key, value in params.items():
        print(f"{key}: {value:.3f}")
```

Code Listing B.5: **Lab 2: Experiment 2: Simulated vs Measured Data Plot**

```
import ltspice
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

raw_file = "Lab2-part2 b sweep.raw"
file_path = "LAB_02_DATA/NMOS_Parametric_Sweep_Lab_02_Exp_02_Excel.xlsx"

df = pd.read_excel(file_path)

Mes_I_DF = pd.read_excel(file_path, usecols=["VGS=0.00V.1"])
Mes_I_uT = np.array(Mes_I_DF)
Mes_I = np.delete(Mes_I_uT, 0, axis=0)
Mes_VDs_DF = pd.read_excel(file_path, usecols=["VGS=0.00V"])
Mes_VDs_uT = np.array(Mes_VDs_DF)
Mes_VDs = np.delete(Mes_VDs_uT, 0, axis=0)

l = ltspice.Ltspice(raw_file)
l.parse()

signals = l.variables
R1 = 1250
voltage = l.get_data('V(vo)')
voltageIN = l.get_data('V(pvdd)')

def divide_array(arr, divisor):
    if divisor == 0:
        return "Error: Division by zero is not allowed"
    return [x / divisor for x in arr]

Current = divide_array(voltage, R1)

plt.plot(voltageIN, np.array(Current) * 1e3, label="Simulated $I_D$", color
    ↪ ="g", linestyle="-", linewidth=2)
plt.plot(Mes_VDs, np.array(Mes_I) * 1e3, label="Measured $I_D$", color="r",
    ↪  linestyle="-", linewidth=2)
plt.xlabel('Voltage (V)')
plt.ylabel('Current (A)')
plt.title('Experiment 2: Simulated vs Measured Data')
plt.legend()
plt.grid()
plt.show()
```

Appendix C

Python Code Listings for Lab 03

Code Listing C.1: **Lab 1: Experiment 2: Common Source Amp LTSpice Sweep for V_o = 5 V**

```
import matplotlib.pyplot as plt

# Data for AC Small Signal Gain RL = 10ΩM
rsig_values = ['10ΩM', '100Ωk', '10Ωk', '1Ωk']
gv_sim_rl_10M = [21.8688, 43.3046, 43.6939, 43.7333]
gv_meas_rl_10M = [7.6110, 16.2100, 20.1700, 22.3100]

# Data for AC Small Signal Gain Rsig = 1Ωk
rl_values = ['10ΩM', '100Ωk', '10Ωk', '1Ωk']
gv_sim_rsig_1k = [43.7333, 27.8597, 6.4794, 0.7839]
gv_meas_rsig_1k = [31.2000, 23.5000, 3.6500, 0.9510]

# Plotting RL = 10ΩM
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
x = range(len(rsig_values))
plt.bar(x, gv_sim_rl_10M, width=0.4, label='Simulated', align='center')
plt.bar([p + 0.4 for p in x], gv_meas_rl_10M, width=0.4, label='Measured',
    ↪ align='center')
plt.xticks([p + 0.2 for p in x], rsig_values)
plt.ylabel("Gain (V/V)")
plt.title("AC Gain vs Rsig (RL = 10ΩM)")
plt.legend()

# Plotting Rsig = 1Ωk
plt.subplot(1, 2, 2)
x = range(len(rl_values))
plt.bar(x, gv_sim_rsig_1k, width=0.4, label='Simulated', align='center')
plt.bar([p + 0.4 for p in x], gv_meas_rsig_1k, width=0.4, label='Measured',
    ↪  align='center')
plt.xticks([p + 0.2 for p in x], rl_values)
plt.ylabel("Gain (V/V)")
plt.title("AC Gain vs RL (Rsig = 1Ωk)")
plt.legend()

plt.tight_layout()
plt.show()

```

```
plt.figure(figsize=(7, 5))
x = range(len(rsig_values))
plt.bar(x, gv_sim_rl_10M, width=0.4, label='Simulated', align='center')
plt.bar([p + 0.4 for p in x], gv_meas_rl_10M, width=0.4, label='Measured',
    ↪ align='center')
plt.xticks([p + 0.2 for p in x], rsig_values)
plt.ylabel("Gain (V/V)")
plt.title("AC Gain vs Rsig (RL = 10ΩM)")
plt.legend()
plt.tight_layout()
plt.show()

# Plotting AC Gain vs RL (Rsig = 1Ωk)
plt.figure(figsize=(7, 5))
x = range(len(rl_values))
plt.bar(x, gv_sim_rsig_1k, width=0.4, label='Simulated', align='center')
plt.bar([p + 0.4 for p in x], gv_meas_rsig_1k, width=0.4, label='Measured',
    ↪  align='center')
plt.xticks([p + 0.2 for p in x], rl_values)
plt.ylabel("Gain (V/V)")
plt.title("AC Gain vs RL (Rsig = 1Ωk)")
plt.legend()
plt.tight_layout()
plt.show()
```

Appendix D

Python Code Listings for Lab 04

Code Listing D.1: **Lab 4: Experiment 2: Description**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings

# Try to import ltspice - we'll handle errors if it's not available
try:
    import ltspice

    have_ltspice = True
except ImportError:
    have_ltspice = False
    print("Warning: ltspice module not found. Will use simulated data
        ↪ instead.")

warnings.filterwarnings('ignore')  # Suppress warnings for cleaner output

# Define Constants
M = 10 ** 6
k = 10 ** 3
m = 10 ** -3
u = 10 ** -6
n = 10 ** -9
p = 10 ** -12
j = 1j  # Imaginary unit

# Frequency and Gain Ranges
f_min = 10
f_max = 1000 * k
Gv_dB_min = 0
Gv_dB_max = 30

# Component Values
VDD = 10
VO = 5
VSS = -10

RG = 100 * k
Rsig = 100 * k
R = 15 * k
```

```
RL = 10 * M

CB = 0.022 * u
C5 = 22 * p
C4 = 3.3 * p
C3 = 37 * p  # 22pF + 15pF (Probe)
C2 = 3.3 * p
C1 = 10 * p

W1 = 50 * u
W2 = 20 * u
W3 = 20 * u
L = 10 * u
Kn_prime = 136 * u
Kp_prime = 100 * u
VTn = 600 * m
VTp = -600 * m
lambda_n = 0.02
lambda_p = 0.03

# Define measured transistor operating values (from lab)
# Q1 - NMOS values (measured)
ID_Q1 = 492.7  # µA
VOV_Q1 = 1.3  # V
VG_Q1 = 1.87  # V
VD_Q1 = 5.14  # V
VS_Q1 = 0  # V

# Q2 - PMOS values (measured)
ID_Q2 = 492.7  # µA
VOV_Q2 = 2.25  # V
VG_Q2 = 7.10  # V
VD_Q2 = 5.14  # V
VS_Q2 = 10  # V

# Q3 - PMOS values (measured)
ID_Q3 = 473.6  # µA
VOV_Q3 = 2.25  # V
VG_Q3 = 7.10  # V
VD_Q3 = 7.10  # V
VS_Q3 = 10  # V

# File paths
ac_raw_file = "/opt/miniconda3/envs/pycharm-env/EE322_ee_Lab_II/Labs/Lab
    ↪ -04/Lab_04_LTSpice/EE322_Lab_04_acSim_BodePlot_Placzek.raw"
op_raw_file = "/opt/miniconda3/envs/pycharm-env/EE322_ee_Lab_II/Labs/Lab
    ↪ -04/Lab_04_LTSpice/EE322_Lab_04_opSim_OperatingPoint_Placzek.raw"
excel_file_path = "/opt/miniconda3/envs/pycharm-env/EE322_ee_Lab_II/Labs/
    ↪ Lab-04/EE_322_eeLab_II_Lab_4_Data.xlsx"

# PART 1: Handle LTSpice operating point simulation data for DC values
try:
    if have_ltspice:
        l_op = ltspice.Ltspice(op_raw_file)
        l_op.parse()  # Parse the operating point file

        print("Available signals in LTspice operating point simulation:")
        print(l_op.variables)
```

```
95
96          # Extract Q1 operating point values
97          # Note: Update these variable names based on your actual LTspice
                ↪ netlist
98          ID_Q1_Sim = abs(l_op.get_data('Id(M1)')[0] * 1e6)  # Convert to µA
99          VGS_Q1_Sim = abs(l_op.get_data('Vgs(M1)')[0])
100         VOV_Q1_Sim = abs(VGS_Q1_Sim - VTn)  # Overdrive voltage
101         VG_Q1_Sim = l_op.get_data('V(n003)')[0]  # Gate voltage
102         VD_Q1_Sim = l_op.get_data('V(vo1)')[0]  # Drain voltage
103         VS_Q1_Sim = 0  # Source connected to ground
104
105         # Extract Q2 operating point values
106         ID_Q2_Sim = abs(l_op.get_data('Id(Q2)')[0] * 1e6)
107         VGS_Q2_Sim = abs(l_op.get_data('Vgs(Q2)')[0])
108         VOV_Q2_Sim = abs(VGS_Q2_Sim - VTp)
109         VG_Q2_Sim = l_op.get_data('V(n001)')[0]
110         VD_Q2_Sim = l_op.get_data('V(vo1)')[0]
111         VS_Q2_Sim = l_op.get_data('V(pvdd)')[0]
112
113         # Extract Q3 operating point values
114         ID_Q3_Sim = abs(l_op.get_data('Id(Q1)')[0] * 1e6)
115         VGS_Q3_Sim = abs(l_op.get_data('Vgs(Q1)')[0])
116         VOV_Q3_Sim = abs(VGS_Q3_Sim - VTp)
117         VG_Q3_Sim = l_op.get_data('V(n001)')[0]
118         VD_Q3_Sim = l_op.get_data('V(n001)')[0]
119         VS_Q3_Sim = l_op.get_data('V(pvdd)')[0]
120
121         have_op_sim_data = True
122         print("Successfully loaded LTspice operating point data.")
123     else:
124         raise ImportError("ltspice module not available")
125 except Exception as e:
126     print(f"Error processing LTspice operating point data: {e}")
127     print("Using default values for simulated operating points...")
128
129     # Use slightly different values from measured data for simulated values
130     # Q1 - NMOS simulated values (defaults)
131     ID_Q1_Sim = 485.3  # µA
132     VOV_Q1_Sim = 1.25  # V
133     VG_Q1_Sim = 1.85  # V
134     VD_Q1_Sim = 5.10  # V
135     VS_Q1_Sim = 0  # V
136
137     # Q2 - PMOS simulated values (defaults)
138     ID_Q2_Sim = 480.0  # µA
139     VOV_Q2_Sim = 2.20  # V
140     VG_Q2_Sim = 7.15  # V
141     VD_Q2_Sim = 5.10  # V
142     VS_Q2_Sim = 10  # V
143
144     # Q3 - PMOS simulated values (defaults)
145     ID_Q3_Sim = 470.2  # µA
146     VOV_Q3_Sim = 2.22  # V
147     VG_Q3_Sim = 7.12  # V
148     VD_Q3_Sim = 7.12  # V
149     VS_Q3_Sim = 10  # V
150
151     have_op_sim_data = False
```

```
152
153 # PART 2: Handle LTSpice AC simulation data
154 try:
155     if have_ltspice:
156         l_ac = ltspice.Ltspice(ac_raw_file)
157         l_ac.parse()  # Parse the AC simulation file
158
159         print("Available signals in LTspice AC simulation:")
160         print(l_ac.variables)
161
162         # Extract frequency and voltage data
163         freq_Sim = l_ac.get_frequency()
164         v_out = l_ac.get_data('V(vo)')  # Output voltage
165         v_in = l_ac.get_data('V(vsig)')  # Input voltage
166
167         # Calculate gain in dB
168         Av_f_Sim = v_out / v_in
169         Av_f_Sim_dB = 20 * np.log10(np.abs(Av_f_Sim))
170
171         # Find mid-band gain (maximum value)
172         Av_mid_Sim_VpV = np.max(np.abs(Av_f_Sim))
173         Av_mid_Sim_dB = np.max(Av_f_Sim_dB)
174
175         # Find cutoff frequencies (-3dB points)
176         mid_band_idx_sim = np.argmax(Av_f_Sim_dB)
177         cutoff_level_sim = Av_mid_Sim_dB - 3
178
179         # Find lower cutoff
180         for i in range(mid_band_idx_sim, 0, -1):
181             if Av_f_Sim_dB[i] <= cutoff_level_sim:
182                 fL_Sim = freq_Sim[i]
183                 break
184         else:
185             fL_Sim = freq_Sim[0]
186
187         # Find upper cutoff
188         for i in range(mid_band_idx_sim, len(freq_Sim) - 1):
189             if Av_f_Sim_dB[i] <= cutoff_level_sim:
190                 fH_Sim = freq_Sim[i]
191                 break
192         else:
193             fH_Sim = freq_Sim[-1]
194
195         # Calculate bandwidth and GBP
196         BW_Sim = (fH_Sim - fL_Sim) / 1e3  # kHz
197         GBP_Sim = Av_mid_Sim_VpV * BW_Sim  # kHz
198
199         have_ac_sim_data = True
200         print("Successfully loaded LTspice AC simulation data.")
201     else:
202         raise ImportError("ltspice module not available")
203 except Exception as e:
204     print(f"Error processing LTspice AC data: {e}")
205     print("Generating simulated LTspice AC response based on model...")
206
207     # Create simulated LTspice data based on a two-pole model
208     # Use different parameters than measured data to simulate realistic
        ↪ comparison
```

```
    fL_Sim = 25  # Lower cutoff in Hz - different from measured
    fH_Sim = 10000  # Upper cutoff in Hz - different from measured
    Av_mid_Sim_VpV = 17.5  # Mid-band gain in V/V - different from measured
    Av_mid_Sim_dB = 20 * np.log10(Av_mid_Sim_VpV)

    # Generate frequency points
    freq_Sim = np.logspace(np.log10(f_min), np.log10(f_max), 500)

    # Calculate simulated response using two-pole model
    Av_f_Sim = Av_mid_Sim_VpV / np.sqrt((1 + (freq_Sim / fL_Sim) ** 2) * (1
        ↪  + (freq_Sim / fH_Sim) ** 2))
    Av_f_Sim_dB = 20 * np.log10(np.abs(Av_f_Sim))

    # Calculate bandwidth and GBP
    BW_Sim = (fH_Sim - fL_Sim) / 1e3  # kHz
    GBP_Sim = Av_mid_Sim_VpV * BW_Sim  # kHz

    have_ac_sim_data = True

# PART 3: Handle measured data
# Measured gain data (dB) - this represents real lab measurements
Gv_f_Meas_dB = np.array([
    6.735471176, 10.09547905, 15.04337406, 19.2955821,
    21.50349233, 23.02006471, 23.59473839, 24.59165764,
    24.67738981, 24.79135348, 24.26025682, 23.52452308,
    22.34934762, 19.23083496, 17.2958244, 15.23897678,
    7.50956576, 3.021838802
])

# Create logarithmically spaced frequency points for the measurement data
# Use a different frequency range than the simulated data
freq_Meas = np.logspace(np.log10(20), np.log10(20000), len(Gv_f_Meas_dB))

# Convert dB to linear scale for gain
Gv_f_Meas_Mag = 10 ** (Gv_f_Meas_dB / 20)

# Find max gain values
Gv_f_Meas_Mid_dB = np.max(Gv_f_Meas_dB)
Gv_f_Meas_Mid_VpV = 10 ** (Gv_f_Meas_Mid_dB / 20)
mid_band_idx = np.argmax(Gv_f_Meas_dB)

# Calculate -3dB points
cutoff_level = Gv_f_Meas_Mid_dB - 3

# Find lower cutoff frequency
for i in range(mid_band_idx, 0, -1):
    if Gv_f_Meas_dB[i] <= cutoff_level:
        fL_Meas = freq_Meas[i]
        break
else:
    fL_Meas = freq_Meas[0]

# Find upper cutoff frequency
for i in range(mid_band_idx, len(freq_Meas) - 1):
    if Gv_f_Meas_dB[i] <= cutoff_level:
        fH_Meas = freq_Meas[i]
        break
else:
```

```
    fH_Meas = freq_Meas[-1]

# Define center frequency and key measurement points
f0_Meas = freq_Meas[mid_band_idx]  # Center frequency
fx_Meas = np.array([fL_Meas, f0_Meas, fH_Meas])
Gv_fx_Meas_dB = np.array([Gv_f_Meas_Mid_dB - 3, Gv_f_Meas_Mid_dB,
    ↪ Gv_f_Meas_Mid_dB - 3])

# Calculate bandwidth and gain-bandwidth product
BW_Meas = (fH_Meas - fL_Meas) / 1e3  # Bandwidth in kHz
GBP_Meas = Gv_f_Meas_Mid_VpV * BW_Meas  # Gain Bandwidth Product in kHz

# PART 4: Create tables and plots
# Create the DC Operating Point Table
# Format numerical values to 3 decimal places in DC table
dc_table_data = {
    "Device": ["Q1", "Q1", "Q1", "Q1", "Q1", "Q2", "Q2", "Q2", "Q2", "Q2",
        ↪ "Q3", "Q3", "Q3", "Q3", "Q3"],
    "Quantity": ["ID (μA)", "|VOV| (V)", "VG (V)", "VD (V)", "VS (V)"] * 3,
    "Simulated": [f"{val:.3f}" for val in [
        ID_Q1_Sim, VOV_Q1_Sim, VG_Q1_Sim, VD_Q1_Sim, VS_Q1_Sim,
        ID_Q2_Sim, VOV_Q2_Sim, VG_Q2_Sim, VD_Q2_Sim, VS_Q2_Sim,
        ID_Q3_Sim, VOV_Q3_Sim, VG_Q3_Sim, VD_Q3_Sim, VS_Q3_Sim
    ]],
    "Measured": [f"{val:.3f}" for val in [
        ID_Q1, VOV_Q1, VG_Q1, VD_Q1, VS_Q1,
        ID_Q2, VOV_Q2, VG_Q2, VD_Q2, VS_Q2,
        ID_Q3, VOV_Q3, VG_Q3, VD_Q3, VS_Q3
    ]],
    "Units": ["μA", "V", "V", "V", "V"] * 3
}

# Format numerical values to 3 decimal places in AC table
ac_table_data = {
    "Quantity": ["|Gv(mid)| (V/V)", "|Gv(mid)| (dB)", "fL (Hz)", "fH (kHz)"
        ↪ , "BW (kHz)", "GBP (kHz)"],
    "Simulated": [f"{val:.3f}" for val in [
        Av_mid_Sim_VpV, Av_mid_Sim_dB, fL_Sim, fH_Sim / 1e3, BW_Sim,
            ↪ GBP_Sim
    ]],
    "Measured": [f"{val:.3f}" for val in [
        Gv_f_Meas_Mid_VpV, Gv_f_Meas_Mid_dB, fL_Meas, fH_Meas / 1e3,
            ↪ BW_Meas, GBP_Meas
    ]],
    "Units": ["V/V", "dB", "Hz", "kHz", "kHz", "kHz"]
}


# Create DataFrames
dc_table = pd.DataFrame(dc_table_data)
ac_table = pd.DataFrame(ac_table_data)

# Display tables
print("\nDC Operating Point Table:")
print(dc_table.to_string(index=False))

print("\nAC Summary Table:")
print(ac_table.to_string(index=False))
```

```

# Export tables to CSV
dc_table.to_csv('dc_operating_point_table.csv', index=False)
ac_table.to_csv('ac_summary_table.csv', index=False)

# Create and save the frequency response plot
plt.figure(figsize=(10, 6))

# Plot simulated response
plt.semilogx(freq_Sim, Av_f_Sim_dB, 'g', linewidth=2, linestyle='--', label
    ↪ ="Simulated (LTspice)")

# Plot measured data
plt.semilogx(freq_Meas, Gv_f_Meas_dB, 'b', linewidth=2, label="Measured")

# Plot key measured points
plt.semilogx(fx_Meas, Gv_fx_Meas_dB, 'ro', markersize=8, label="Key
    ↪ Measured Points")

# Set plot properties
plt.grid(True, which="both", linestyle="--")
plt.xlabel("Frequency (Hz)", fontsize=12)
plt.ylabel("Gain (dB)", fontsize=12)
plt.title("Frequency Response: Simulated vs Measured", fontsize=14)
plt.legend()
plt.xlim(f_min, f_max)
plt.ylim(Gv_dB_min, Gv_dB_max)
plt.tight_layout()

# Save and display the plot
plt.savefig('frequency_response_plot.png')
plt.show()
```

Appendix E

Python Code Listings for Lab 06

Code Listing E.1: **Lab 6: Experiment 1: Description**

```
import ltspice
import numpy as np
import math
import re
import chardet

########################## Helpers ########################
def percent_diff(sim, meas):
    try:
        return abs((sim - meas) / meas) * 100 if meas != 0 else None
    except:
        return None

def format_value(val):
    return f"{val:.4f}" if isinstance(val, float) else "N/A"

########################## AC Simulation - Single Ended ################
    ↪ ########
def process_single_ended(filename):
    l = ltspice.Ltspice(filename)
    l.parse()

    VR = l.get_data('V(vr)')
    VL = l.get_data('V(vl)')
    Vn = l.get_data('V(vn)')
    Vp = l.get_data('V(vp)')

    VRmag = (max(VR) - min(VR)) / 2
    Vnmag = (max(Vn) - min(Vn)) / 2
    VLmag = (max(VL) - min(VL)) / 2
    Vpmag = (max(Vp) - min(Vp)) / 2

    AD_Single_R = (VRmag / Vnmag) / 2
    AD_Single_R_dB = 20 * math.log10(AD_Single_R)

    return {
        "Ad": AD_Single_R,
        "Ad_dB": AD_Single_R_dB
    }

```

```
########################## AC Simulation - Common Mode ####################
    ↪ ####
def process_common_mode(filename):
    l = ltspice.Ltspice(filename)
    l.parse()

    VR = l.get_data('V(vr)')
    VL = l.get_data('V(vl)')
    Vn = l.get_data('V(vn)')
    Vp = l.get_data('V(vp)')

    VRmag = (max(VR) - min(VR)) / 2
    Vnmag = (max(Vn) - min(Vn)) / 2
    VLmag = (max(VL) - min(VL)) / 2
    Vpmag = (max(Vp) - min(Vp)) / 2
    Vcm = (Vnmag + Vpmag) / 2
    Vout = (VRmag - VLmag) / 2

    Acm_Single_R = VRmag / Vcm
    Acm_Single_R_dB = 20 * math.log10(Acm_Single_R)

    AD_Double = (VRmag + VLmag) / (2 * Vnmag)
    AD_Double_dB = 20 * math.log10(AD_Double)

    Acm_Double = abs(Vout / Vcm)
    Acm_Double_dB = 20 * math.log10(Acm_Double)

    CMRR_Single_dB = 20 * math.log10((VRmag / Vnmag) / 2) - Acm_Single_R_dB
    CMRR_Double_dB = AD_Double_dB - Acm_Double_dB

    return {
        "Acm": Acm_Single_R,
        "Acm_dB": Acm_Single_R_dB,
        "CMRR_dB": CMRR_Single_dB,
        "Ad_Double": AD_Double,
        "Ad_Double_dB": AD_Double_dB,
        "Acm_Double": Acm_Double,
        "Acm_Double_dB": Acm_Double_dB,
        "CMRR_Double_dB": CMRR_Double_dB
    }

########################## DC Operating Point Parsing ####################
    ↪ ####
def extract_simulated_dc_points(file_path):
    simulated = {"Q1": {}, "Q2": {}}
    try:
        with open(file_path, 'rb') as f:
            raw_data = f.read()
            encoding = chardet.detect(raw_data)['encoding']

        with open(file_path, 'r', encoding=encoding) as file:
            content = file.read()

        id_match = re.search(r'Id:\s*([-\d.e+]+)\s+([-\d.e+]+)', content)
        vgs_match = re.search(r'Vgs:\s*([-\d.e+]+)\s+([-\d.e+]+)', content)
        vds_match = re.search(r'Vds:\s*([-\d.e+]+)\s+([-\d.e+]+)', content)
        vth_match = re.search(r'Vth:\s*([-\d.e+]+)\s+([-\d.e+]+)', content)

```

```
        if id_match and vgs_match and vds_match and vth_match:
            # Q1
            simulated["Q1"]["ID"] = float(id_match.group(1)) * 1e6
            simulated["Q1"]["VGS"] = float(vgs_match.group(1))
            simulated["Q1"]["VDS"] = float(vds_match.group(1))
            simulated["Q1"]["VTH"] = float(vth_match.group(1))
            simulated["Q1"]["VOV"] = abs(simulated["Q1"]["VGS"] - simulated
                ↪ ["Q1"]["VTH"])
            simulated["Q1"]["VG"] = simulated["Q1"]["VGS"]
            simulated["Q1"]["VD"] = simulated["Q1"]["VDS"]
            simulated["Q1"]["VS"] = 0

            # Q2
            simulated["Q2"]["ID"] = float(id_match.group(2)) * 1e6
            simulated["Q2"]["VGS"] = float(vgs_match.group(2))
            simulated["Q2"]["VDS"] = float(vds_match.group(2))
            simulated["Q2"]["VTH"] = float(vth_match.group(2))
            simulated["Q2"]["VOV"] = abs(simulated["Q2"]["VGS"] - simulated
                ↪ ["Q2"]["VTH"])
            simulated["Q2"]["VG"] = simulated["Q2"]["VGS"]
            simulated["Q2"]["VD"] = simulated["Q2"]["VDS"]
            simulated["Q2"]["VS"] = 0

    except Exception as e:
        print(f"Error parsing log file: {e}")

    return simulated

########################## Table Printers ########################
def print_dc_summary(simulated, measured):
    print("\n========== DC Summary Table ==========")
    headers = f"{'Device':<6} | {'Quantity':<6} | {'Simulated':>10} | {'
        ↪ Measured':>10} | Units"
    print(headers)
    print("-" * len(headers))

    for device in ["Q1", "Q2"]:
        quantities = [
            ("ID", "µA"),
            ("VOV", "V"),
            ("VG", "V"),
            ("VD", "V"),
            ("VS", "V")
        ]
        for q, unit in quantities:
            sim_val = simulated[device].get(q, "N/A")
            meas_val = measured[device].get(q, "N/A")
            sim_str = f"{sim_val:.3f}" if isinstance(sim_val, float) else
                ↪ sim_val
            meas_str = f"{meas_val:.3f}" if isinstance(meas_val, float)
                ↪ else meas_val
            print(f"{device:<6} | {q:<6} | {sim_str:>10} | {meas_str:>10} |
                ↪  {unit}")
    print("=" * len(headers))


def print_ac_summary_table(title, data):
    print(f"\n========== {title} ==========")
```

```
    headers = f"{'Quantity':<8} | {'Simulated':>10} | {'Measured':>10} | {'Diff (%)':>10} | Units"
    print(headers)
    print("-" * len(headers))

    for row in data:
        quantity, sim_val, meas_val, units = row
        diff = percent_diff(sim_val, meas_val) if meas_val is not None else None
        sim_str = f"{sim_val:.4f}" if isinstance(sim_val, float) else "N/A"
        meas_str = f"{meas_val:.4f}" if isinstance(meas_val, float) else "N/A"
        diff_str = f"{diff:.2f}%" if diff is not None else "N/A"
        print(f"{quantity:<8} | {sim_str:>10} | {meas_str:>10} | {diff_str:>10} | {units}")
    print("=" * len(headers))

########################## MAIN EXECUTION ########################
# File paths
single_filename = "Lab_6_180_Deg_Quadrature_Placzek.raw"
common_filename = "Lab_6_Common_Mode_Placzek.raw"
log_file_path = '/opt/miniconda3/envs/pycharm-env/EE322_ee_Lab_II/Labs/Lab-06/EE322_Lab_6_Measured_Values_Placzek.log'

# AC analysis
single_results = process_single_ended(single_filename)
common_results = process_common_mode(common_filename)

# Measured AC values
measured_ac = {
    "Ad": 6.205,
    "Ad_dB": 15.801,
    "Acm": 0.499,
    "Acm_dB": -5.924,
    "CMRR_dB": 21.724,
    "Ad_Double": 6.544,
    "Ad_Double_dB": 16.319,
    "Acm_Double": 0.342,
    "Acm_Double_dB": -9.312,
    "CMRR_Double_dB": 25.631
}

# DC analysis
simulated_dc = extract_simulated_dc_points(log_file_path)
measured_dc = {
    "Q1": {"ID": 483.110, "VOV": 1.141, "VG": 0.000, "VD": 2.896, "VS": -1.820},
    "Q2": {"ID": 524.043, "VOV": 1.401, "VG": 0.000, "VD": 2.011, "VS": -1.819}
}

# AC Summary Tables
single_ended_data = [
    ("Ad", single_results["Ad"], measured_ac["Ad"], "V/V"),
    ("Ad", single_results["Ad_dB"], measured_ac["Ad_dB"], "dB"),
    ("Acm", common_results["Acm"], measured_ac["Acm"], "V/V"),
    ("Acm", common_results["Acm_dB"], measured_ac["Acm_dB"], "dB"),
    ("CMRR", common_results["CMRR_dB"], measured_ac["CMRR_dB"], "dB"),
```

```
]

differential_data = [
    ("Ad", common_results["Ad_Double"], measured_ac["Ad_Double"], "V/V"),
    ("Ad", common_results["Ad_Double_dB"], measured_ac["Ad_Double_dB"], "dB
        ↪ "),
    ("Acm", common_results["Acm_Double"], measured_ac["Acm_Double"], "V/V")
        ↪ ,
    ("Acm", common_results["Acm_Double_dB"], measured_ac["Acm_Double_dB"],
        ↪ "dB"),
    ("CMRR", common_results["CMRR_Double_dB"], measured_ac["CMRR_Double_dB"
        ↪ ], "dB"),
]

# Print everything
print_dc_summary(simulated_dc, measured_dc)
print_ac_summary_table("AC Summary - Single Ended", single_ended_data)
print_ac_summary_table("AC Summary - Differential", differential_data)
```

Appendix F

Python Code Listings for Lab 07

Code Listing F.1: **Lab 7: Experiment 1: Calculations for DC Summary, Table 1**

```
import re
import chardet
import unicodedata
import re
import chardet
from typing import Optional

# === FILE SETUP ===
log_file_path = "/opt/miniconda3/envs/pycharm-env/EE322_ee_Lab_II/Labs/Lab-07/EE322_Lab_07_DC_LTspice_Sim_Placzek.log"
file_path = "/opt/miniconda3/envs/pycharm-env/EE322_ee_Lab_II/Labs/Lab-07/EE322_Lab_07_DC_LTspice_Sim_Placzek.log"
# Read and decode
with open(log_file_path, "rb") as f:
    raw_data = f.read()
    encoding = chardet.detect(raw_data)["encoding"]
    log_data = raw_data.decode(encoding)
    log_data = unicodedata.normalize("NFKD", log_data)

# === EXTRACT NODE VOLTAGES FOR Q1/Q2 ===
v_vo = float(re.search(r"V\(vo\)\s+([-\d.eE]+)", log_data).group(1))  # source
v_vl = float(re.search(r"V\(vl\)\s+([-\d.eE]+)", log_data).group(1))  # drain of Q1
v_vr = float(re.search(r"V\(vr\)\s+([-\d.eE]+)", log_data).group(1))  # drain of Q2

# === EXTRACT TRANSISTOR DATA ===
def extract_transistor_columns(log_data):
    block_match = re.search(r"--- MOSFET Transistors ---(.+?)Operating Bias Point", log_data, re.DOTALL)
    if not block_match:
        raise ValueError("Couldn't find MOSFET transistor block.")

    block = block_match.group(1)
    lines = [line.strip() for line in block.strip().splitlines() if line.strip()]
    labels = lines[0].split()[1:]  # Skip "Name:"

```

```
    numeric_data = {}
    for line in lines[1:]:
        if line.startswith("Model:"):
            continue
        parts = line.split()
        key = parts[0].replace(":", "")
        numeric_data[key] = parts[1:]

    data_by_transistor = {}
    for i, label in enumerate(labels):
        data_by_transistor[label] = {
            "Id": float(numeric_data["Id"][i]),
            "Vgs": float(numeric_data["Vgs"][i]),
            "Vds": float(numeric_data["Vds"][i]),
            "Vth": float(numeric_data["Vth"][i])
        }
    return data_by_transistor

# === DETERMINE NODE VOLTAGES FROM .LOG ===
def reconstruct_voltages(params, is_pmos, v_d=None, v_s=None):
    vgs = params["Vgs"]
    vds = params["Vds"]
    vth = params["Vth"]
    id_val = params["Id"]
    vov = abs(vgs - vth)

    if is_pmos:
        v_s = 12.0
        v_d = v_s + vds
        v_g = v_s + vgs
    else:
        v_g = 0.0
        # v_s and v_d passed in from node voltages

    return {
        "I_D": abs(id_val * 1000),   # Convert to mA (optional: match your
            ↪ table units)
        "V_OV": vov,
        "V_G": v_g,
        "V_D": v_d,
        "V_S": v_s
    }


# ======== EXECUTE SIMULATED =============
raw_mos_data = extract_transistor_columns(log_data)

results = {
    "Q1": reconstruct_voltages(raw_mos_data["m§q1"], is_pmos=False, v_d=
        ↪ v_vl, v_s=v_vo),
    "Q2": reconstruct_voltages(raw_mos_data["m§q2"], is_pmos=False, v_d=
        ↪ v_vr, v_s=v_vo),
    "Q3": reconstruct_voltages(raw_mos_data["m§q3"], is_pmos=True),
    "Q4": reconstruct_voltages(raw_mos_data["m§q4"], is_pmos=True),
}

# === PRINT RESULTS ===
def print_table(q_data, label):
```

```
    print(f"Device: {label}")
    print(f"I_D     = {q_data['I_D']:.6f} A")
    print(f"|V_OV|  = {q_data['V_OV']:.4f} V")
    print(f"V_G     = {q_data['V_G']:.4f} V")
    print(f"V_D     = {q_data['V_D']:.4f} V")
    print(f"V_S     = {q_data['V_S']:.4f} V")
    print("-" * 30)

for label in ["Q1", "Q2", "Q3", "Q4"]:
    print_table(results[label], label)

########################## Measured I_Dk Calculations ###################
    ↪ ####

def calculate_percentage_difference(sim: float, meas: float) -> Optional[
    ↪ float]:
    if meas == 0:
        return None
    try:
        return abs((sim - meas) / meas) * 100
    except ZeroDivisionError:
        return None

def format_value(val):
    return f"{val:.4f}" if isinstance(val, float) else "N/A"

########################## Simulated DC Operating Point Parsing for
    ↪ printing ########################

def extract_simulated_dc_points(file_path):
    simulated = {"Q1": {}, "Q2": {}, "Q3": {}, "Q4": {}}
    try:
        # Detect file encoding
        with open(file_path, 'rb') as f:
            raw_data = f.read()
            encoding = chardet.detect(raw_data)['encoding']

        # Read the log file content
        with open(file_path, 'r', encoding=encoding) as file:
            content = file.read()

        # Anchor each parameter to the start of a line (MULTILINE mode)
        id_match = re.search(r'^Id:\s*([-\d.e+]+)\s+([-\d.e+]+)\s+([-\d.e
            ↪ +]+)\s+([-\d.e+]+)', content, re.MULTILINE)
        vgs_match = re.search(r'^Vgs:\s*([-\d.e+]+)\s+([-\d.e+]+)\s+([-\d.e
            ↪ +]+)\s+([-\d.e+]+)', content, re.MULTILINE)
        vds_match = re.search(r'^Vds:\s*([-\d.e+]+)\s+([-\d.e+]+)\s+([-\d.e
            ↪ +]+)\s+([-\d.e+]+)', content, re.MULTILINE)
        vth_match = re.search(r'^Vth:\s*([-\d.e+]+)\s+([-\d.e+]+)\s+([-\d.e
            ↪ +]+)\s+([-\d.e+]+)', content, re.MULTILINE)

        if id_match and vgs_match and vds_match and vth_match:
            devices = ["Q1", "Q2", "Q3", "Q4"]
            for i, device in enumerate(devices, start=1):
                # Convert the simulated drain current from A to µA
                simulated[device]["ID"] = float(id_match.group(i)) * 1e6
                simulated[device]["VGS"] = float(vgs_match.group(i))
                simulated[device]["VDS"] = float(vds_match.group(i))
```

```
                simulated[device]["VTH"] = float(vth_match.group(i))
                simulated[device]["VOV"] = abs(simulated[device]["VGS"] -
                    ↪ simulated[device]["VTH"])
                simulated[device]["VG"] = simulated[device]["VGS"]
                simulated[device]["VD"] = simulated[device]["VDS"]
                simulated[device]["VS"] = 0.0  # Assumed for simulation
        else:
            print("One or more parameter lines could not be found in the
                ↪ log file.")
    except Exception as e:
        print(f"Error parsing log file: {e}")

    return simulated

########################## Measured DC Calculation ########################

def calculate_measured_id(vg, vs, device_type):
    if device_type == "N":  # For Q1 and Q2 (N-channel)
        Vth = 0.62      # Typical threshold voltage (V)
        k = 580e-6      # Transconductance parameter in A/V² (580 µA/V²)
        vgs = vg - vs
        vov = vgs - Vth
        return 0.5 * k * (vov ** 2)
    elif device_type == "P":  # For Q3 and Q4 (P-channel)
        Vth = 1.00      # Typical threshold magnitude (V)
        k = 189e-6      # Transconductance parameter in A/V² (189 µA/V²)
        vsg = vs - vg
        vov = vsg - Vth
        return 0.5 * k * (vov ** 2)
    else:
        return None

# Measured node voltages from the lab (as obtained from your setup)
measured_nodes = {
    "Q1": {"VG": 0.000, "VD": 8.606, "VS": -1.985},
    "Q2": {"VG": 0.000, "VD": 8.606, "VS": -1.985},
    "Q3": {"VG": 8.606, "VD": 8.606, "VS": 12.000},
    "Q4": {"VG": 8.606, "VD": 8.606, "VS": 12.000},
}

# Calculate the measured DC parameters including ID and |VOV|
measured_dc = {}
for device in ["Q1", "Q2", "Q3", "Q4"]:
    if device in ["Q1", "Q2"]:
        id_meas = calculate_measured_id(measured_nodes[device]["VG"],
                                        measured_nodes[device]["VS"],
                                        "N")
        vov = (measured_nodes[device]["VG"] - measured_nodes[device]["VS"])
            ↪  - 0.62
        measured_dc[device] = {
            "ID": id_meas * 1e6,  # Convert A to µA
            "VOV": vov,
            "VG": measured_nodes[device]["VG"],
            "VD": measured_nodes[device]["VD"],
            "VS": measured_nodes[device]["VS"]
        }
    else:
        id_meas = calculate_measured_id(measured_nodes[device]["VG"],
```

```
                                        measured_nodes[device]["VS"],
                                        "P")
        vov = (measured_nodes[device]["VS"] - measured_nodes[device]["VG"])
            ↪  - 1.00
        measured_dc[device] = {
            "ID": id_meas * 1e6,  # in µA
            "VOV": vov,
            "VG": measured_nodes[device]["VG"],
            "VD": measured_nodes[device]["VD"],
            "VS": measured_nodes[device]["VS"]
        }

########################## Table Printer ########################

def print_dc_summary(simulated, measured):
    print("\n========== DC Summary Table ==========")
    headers = (f"{'Device':<6} | {'Quantity':<6} | {'Simulated':>10} | "
               f"{'Measured':>10} | Units | {'Difference':>10} | {'%
                   ↪ Difference':>12}")
    print(headers)
    print("-" * len(headers))

    for device in ["Q1", "Q2", "Q3", "Q4"]:
        quantities = [
            ("ID", "µA"),
            ("VOV", "V"),
            ("VG", "V"),
            ("VD", "V"),
            ("VS", "V")
        ]
        for q, unit in quantities:
            sim_val = simulated[device].get(q, None)
            meas_val = measured[device].get(q, None)
            # If either value is missing, use "N/A"
            if sim_val is None or meas_val is None:
                sim_str = sim_val if sim_val is not None else "N/A"
                meas_str = meas_val if meas_val is not None else "N/A"
                diff_str = "N/A"
                pct_str = "N/A"
            else:
                sim_str = f"{sim_val:.3f}"
                meas_str = f"{meas_val:.3f}"
                diff = abs(sim_val - meas_val)
                pct_diff = calculate_percentage_difference(sim_val,
                    ↪ meas_val)
                diff_str = f"{diff:.3f}"
                pct_str = f"{pct_diff:.3f}" if pct_diff is not None else "N
                    ↪ /A"
            print(f"{device:<6} | {q:<6} | {sim_str:>10} | {meas_str:>10} |
                ↪  {unit:<5} | {diff_str:>10} | {pct_str:>12}")
    print("=" * len(headers))

########################## Main Execution ########################

# File path to the simulated log file (update the path as needed)
log_file_path = "/opt/miniconda3/envs/pycharm-env/EE322_ee_Lab_II/Labs/Lab
    ↪ -07/EE322_Lab_07_DC_LTspice_Sim_Placzek.log"
simulated_dc = extract_simulated_dc_points(log_file_path)
```

```
246
247 # For demonstration, if simulated_dc could not be parsed, we use dummy
    ↪ simulation data:
248 if not simulated_dc["Q1"]:
249     simulated_dc = {
250         "Q1": {"ID": 0.000, "VOV": 1.267, "VG": 0.000, "VD": 9.011, "VS":
            ↪ -1.844},
251         "Q2": {"ID": 0.000, "VOV": 1.267, "VG": 0.000, "VD": 9.011, "VS":
            ↪ -1.844},
252         "Q3": {"ID": 0.000, "VOV": 2.343, "VG": 9.010, "VD": 9.010, "VS":
            ↪ 12.000},
253         "Q4": {"ID": 0.000, "VOV": 2.343, "VG": 9.010, "VD": 9.010, "VS":
            ↪ 12.000}
254     }
255
256 # Print the measured node voltages
257 print("Measured Node Voltages:")
258 for transistor, values in measured_nodes.items():
259     print(f"{transistor}:")
260     for param, value in values.items():
261         print(f"  {param}: {value:.3f}")
262     print()
263
264
265
266 import csv
267
268
269 def save_dc_summary_to_csv(simulated, measured, filename="/opt/miniconda3/
    ↪ envs/pycharm-env/EE322_ee_Lab_II/Labs/Lab-07/dc_summary.csv"):
270     rows = []
271     header = ["Device", "Quantity", "Simulated", "Measured", "Units", "
        ↪ Difference", "% Difference"]
272     for device in ["Q1", "Q2", "Q3", "Q4"]:
273         quantities = [
274             ("ID", "µA"),
275             ("VOV", "V"),
276             ("VG", "V"),
277             ("VD", "V"),
278             ("VS", "V")
279         ]
280         for q, unit in quantities:
281             sim_val = simulated[device].get(q, None)
282             meas_val = measured[device].get(q, None)
283             if sim_val is None or meas_val is None:
284                 sim_str = sim_val if sim_val is not None else "N/A"
285                 meas_str = meas_val if meas_val is not None else "N/A"
286                 diff_str = "N/A"
287                 pct_str = "N/A"
288             else:
289                 sim_str = f"{sim_val:.3f}"
290                 meas_str = f"{meas_val:.3f}"
291                 diff = abs(sim_val - meas_val)
292                 pct_diff = calculate_percentage_difference(sim_val,
                    ↪ meas_val)
293                 diff_str = f"{diff:.3f}"
294                 pct_str = f"{pct_diff:.3f}" if pct_diff is not None else "N
                    ↪ /A"
```

```
            rows.append([device, q, sim_str, meas_str, unit, diff_str,
                ↪ pct_str])

    # Save to CSV
    with open(filename, "w", newline="") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(header)
        writer.writerows(rows)

    print(f"\n DC summary saved to {filename}")
# Print the DC summary table comparing simulated and measured values
print_dc_summary(simulated_dc, measured_dc)
save_dc_summary_to_csv(simulated_dc, measured_dc)
```

Code Listing F.2: **Lab 7: Experiment 2: Calculations for AC Summary, Table 3**

```
import ltspice
import numpy as np
import math

# === Helper Functions ===
def signal_magnitude(signal):
    return (max(signal) - min(signal)) / 2

def log_dB(value):
    return 20 * math.log10(value)

# === Differential Mode ===
l_diff = ltspice.Ltspice("EE322_Lab_07_Differential_Input_Placzek.raw")
l_diff.parse()

VR_diff = l_diff.get_data('V(vr)')
Vn_diff = l_diff.get_data('V(vn)')

VRmag_diff = signal_magnitude(VR_diff)
Vnmag_diff = signal_magnitude(Vn_diff)

AD_single_R = (VRmag_diff / Vnmag_diff) / 2
AD_single_R_dB = log_dB(AD_single_R)

# === Common Mode ===
l_cm = ltspice.Ltspice("EE322_Lab_07_Common_Mode_Placzek.raw")
l_cm.parse()

VR_cm = l_cm.get_data('V(vr)')
Vn_cm = l_cm.get_data('V(vn)')
Vp_cm = l_cm.get_data('V(vp)')

VRmag_cm = signal_magnitude(VR_cm)
Vcm = (signal_magnitude(Vn_cm) + signal_magnitude(Vp_cm)) / 2

Acm_single_R = VRmag_cm / Vcm
Acm_single_R_dB = log_dB(Acm_single_R)

# === CMRR Calculation ===
CMRR = AD_single_R / Acm_single_R
CMRR_dB = log_dB(CMRR)

# === Output ===
print('-------------------------------------')
print('Ad_Single (V/V):', round(AD_single_R, 5))
print('Ad_Single (dB):', round(AD_single_R_dB, 5))
print('-------------------------------------')
print('Acm_Single (V/V):', round(Acm_single_R, 5))
print('Acm_Single (dB):', round(Acm_single_R_dB, 5))
print('-------------------------------------')
print('CMRR (V/V):', round(CMRR, 5))
print('CMRR (dB):', round(CMRR_dB, 5))
print('-------------------------------------')
```

Code Listing F.3: **Lab 7: Python Data Analysis and Plots**

```
import ltspice
import numpy as np
import matplotlib.pyplot as plt

# Function to plot data from a given .raw file
def plot_v_output(filename, label, color):
    l = ltspice.Ltspice(filename)
    l.parse()

    time = l.get_data('time')
    V_vo = l.get_data('V(vo)')

    plt.plot(time, V_vo, label=label, color=color)
    plt.title(label)
    plt.xlabel("Time (s)")
    plt.ylabel("V(vo) (V)")
    plt.grid(True)
    plt.legend(loc='upper right')

    # Add y-axis margin
    margin = 0.085
    y_min, y_max = min(V_vo), max(V_vo)
    y_range = y_max - y_min
    plt.ylim(y_min - margin * y_range, y_max + margin * y_range)

    # --- Max Value Annotation ---
    idx_max = np.argmax(V_vo)
    t_max, v_max = time[idx_max], V_vo[idx_max]
    plt.plot(t_max, v_max, 'go')  # green dot
    plt.text(
        t_max, v_max + y_range * 0.035,
        f"Max: {v_max:.3f} V",
        color='black',
        ha='center',
        bbox=dict(facecolor='white', edgecolor='grey', boxstyle='round,pad
            ↪ =0.3')
    )

    # --- Min Value Annotation ---
    idx_min = np.argmin(V_vo)
    t_min, v_min = time[idx_min], V_vo[idx_min]
    plt.plot(t_min, v_min, 'ro')  # red dot
    plt.text(
        t_min, v_min - y_range * 0.055,
        f"Min: {v_min:.3f} V",
        color='black',
        ha='center',
        bbox=dict(facecolor='white', edgecolor='grey', boxstyle='round,pad
            ↪ =0.3')
    )

    plt.tight_layout()
    plt.show()


# File names
```

```
filename_180_deg = "EE322_Lab_07_Common_Mode_Placzek.raw"
filename_common_mode = "EE322_Lab_07_Differential_Input_Placzek.raw"

# Create and show two separate plots
plt.figure(figsize=(10, 6))
plot_v_output(filename_180_deg, label="AC - Differential Input V(vo)", color='blue')

plt.figure(figsize=(10, 6))
plot_v_output(filename_common_mode, label="AC - Common Mode V(vo)", color='red')
```

Appendix G

Python Code Listings for Lab 08

Code Listing G.1: **Lab 8: Low-Pass Filter "Open-Short-Thru" S-Parameter De-Embedding Method**

```
import skrf as rf
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt

open_standard = rf.Network('Open.s2p')
short_standard = rf.Network('Short.s2p')
thru_standard = rf.Network('Thru.s2p')
dut = rf.Network('lpf.s2p')

def deembed_osp(dut, open_std, short_std, thru_std):
    thru_half = thru_std.copy()
    thru_half.s = np.zeros_like(thru_std.s)
    for i in range(len(thru_std.f)):
        s = thru_std.s[i]
        s_3d = s.reshape(1, *s.shape)
        t = rf.s2t(s_3d)
        t_half = scipy.linalg.sqrtm(t[0])
        t_half_3d = t_half.reshape(1, *t_half.shape)
        s_half = rf.t2s(t_half_3d)[0]
        thru_half.s[i] = s_half

    y_open = open_std.y
    z_short = short_std.z
    y_parasitic = y_open
    y_short = short_std.y
    y_short_corrected = y_short - y_parasitic
    z_parasitic = rf.network.y2z(y_short_corrected)

    dut_y = dut.y
    dut_y_corrected = dut_y - y_parasitic
    dut_z = rf.network.y2z(dut_y_corrected)
    dut_z_corrected = dut_z - z_parasitic

    dut_corrected = rf.Network(s=rf.z2s(dut_z_corrected),
                               frequency=dut.frequency)
    return dut_corrected

```

```
lpf_deembedded = deembed_osp(dut, open_standard, short_standard,
    ↪ thru_standard)
lpf_deembedded.write_touchstone('lpf_deembedded.s2p')

plt.figure(figsize=(12, 8))

plt.subplot(2, 1, 1)
plt.title('LPF - OST De-Embedding - S21 - Transmission')
dut.plot_s_db(m=0, n=1, label='Original')
lpf_deembedded.plot_s_db(m=0, n=1, label='De-embedded')
plt.grid(True)
plt.legend()

plt.subplot(2, 1, 2)
plt.title('LPF - OST De-Embedding - S11 - Reflection')
dut.plot_s_db(m=0, n=0, label='Original')
lpf_deembedded.plot_s_db(m=0, n=0, label='De-embedded')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.savefig('lpf_deembedded_comparison.png')
plt.show()
```

Code Listing G.2: **Lab 8: "Thru" S-Parameter De-Embedding Method (IEEE std. P370)**

```
import skrf as rf
import ieee_p370_2x_thru_alt
# from ieee_p370_2x_thru_alt import deembed_s_params
import numpy as np

def deembed_s_params(s_fixture_dut_fixture, s_side1, s_side2):
    """
    De-embeds the S-parameters of a DUT from measurements that include test fixtures.

    Parameters
    ----------
    s_fixture_dut_fixture : rf.Network
        S-parameter network object of the DUT with fixtures
    s_side1 : rf.Network
        S-parameter network object of the error box for port 1
    s_side2 : rf.Network
        S-parameter network object of the error box for port 2

    Returns
    -------
    s_deembedded_dut : rf.Network
        De-embedded S-parameter network object of the DUT
    """
    # Create T-parameters for the error boxes
    t_side1 = rf.s2t(s_side1.s)
    t_side2 = rf.s2t(s_side2.s)

    # Create T-parameters for the fixture+DUT+fixture
    t_fixture_dut_fixture = rf.s2t(s_fixture_dut_fixture.s)

    # De-embed the DUT
    t_dut = np.zeros(t_fixture_dut_fixture.shape, dtype=complex)

    # T_DUT = inv(T_side1) * T_fixture_dut_fixture * inv(T_side2)
    for i in range(len(t_fixture_dut_fixture)):
        t_dut[i] = (
            np.linalg.inv(t_side1[i])
            @ t_fixture_dut_fixture[i]
            @ np.linalg.inv(t_side2[i])
        )

    # Convert back to S-parameters
    s_deembedded_dut = s_fixture_dut_fixture.copy()
    s_deembedded_dut.s = rf.t2s(t_dut)

    return s_deembedded_dut

# 1. Load your 2x-thru calibration measurement
s_2xthru = rf.Network('Thru.s2p')

# 2. Generate the error boxes (fixture models) from the 2x-thru
# Modified to use use_ifft=False to avoid the problematic IFFT calculation
s_side1, s_side2 = ieee_p370_2x_thru_alt.ieee_p370_2x_thru_alt(s_2xthru, use_ifft=False)

```

```
# 3. Load your DUT measurement (with fixtures)
s_fixture_dut_fixture = rf.Network('lpf.s2p')

# 4. De-embed the fixture effects to get the actual DUT response
s_deembedded_dut = deembed_s_params(s_fixture_dut_fixture, s_side1, s_side2 )

# 5. Save the de-embedded DUT S-parameters to a file
s_deembedded_dut.write_touchstone('lpf_ieee_deembedded_dut.s2p')

# Optional: Plot the results to verify
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.title('LPF S-parameters Before and After De-embedding')

# Plot S11 before and after de-embedding
plt.plot(s_fixture_dut_fixture.f/1e9, 20*np.log10(np.abs(s_fixture_dut_fixture.s[:,0,0])),
         label='S11 with fixtures')
plt.plot(s_deembedded_dut.f/1e9, 20*np.log10(np.abs(s_deembedded_dut.s[:,0,0])),
         label='S11 de-embedded')

# Plot S21 before and after de-embedding
plt.plot(s_fixture_dut_fixture.f/1e9, 20*np.log10(np.abs(s_fixture_dut_fixture.s[:,1,0])),
         label='S21 with fixtures')
plt.plot(s_deembedded_dut.f/1e9, 20*np.log10(np.abs(s_deembedded_dut.s[:,1,0])),
         label='S21 de-embedded')

plt.xlabel('Frequency (MHz)')
plt.ylabel('Magnitude (dB)')
plt.legend()
plt.grid(True)
plt.show()
```

Code Listing G.3: **Lab 8: Low-Pass Filter "Thru" S-Parameter De-Embedding Method (IEEE std. P370)**

```
import skrf as rf
import ieee_p370_2x_thru
# from ieee_p370_2x_thru import deembed_s_params
import numpy as np

def deembed_s_params(s_fixture_dut_fixture, s_side1, s_side2):
    """
    De-embeds the S-parameters of a DUT from measurements that include test fixtures.

    Parameters
    ----------
    s_fixture_dut_fixture : rf.Network
        S-parameter network object of the DUT with fixtures
    s_side1 : rf.Network
        S-parameter network object of the error box for port 1
    s_side2 : rf.Network
        S-parameter network object of the error box for port 2

    Returns
    -------
    s_deembedded_dut : rf.Network
        De-embedded S-parameter network object of the DUT
    """
    # Create T-parameters for the error boxes
    t_side1 = rf.s2t(s_side1.s)
    t_side2 = rf.s2t(s_side2.s)

    # Create T-parameters for the fixture+DUT+fixture
    t_fixture_dut_fixture = rf.s2t(s_fixture_dut_fixture.s)

    # De-embed the DUT
    t_dut = np.zeros(t_fixture_dut_fixture.shape, dtype=complex)

    # T_DUT = inv(T_side1) * T_fixture_dut_fixture * inv(T_side2)
    for i in range(len(t_fixture_dut_fixture)):
        t_dut[i] = (
            np.linalg.inv(t_side1[i])
            @ t_fixture_dut_fixture[i]
            @ np.linalg.inv(t_side2[i])
        )

    # Convert back to S-parameters
    s_deembedded_dut = s_fixture_dut_fixture.copy()
    s_deembedded_dut.s = rf.t2s(t_dut)

    return s_deembedded_dut

# 1. Load your 2x-thru calibration measurement
s_2xthru = rf.Network('Thru.s2p')

# 2. Generate the error boxes (fixture models) from the 2x-thru
# Modified to use use_ifft=False to avoid the problematic IFFT calculation
s_side1, s_side2 = ieee_p370_2x_thru.ieee_p370_2x_thru(s_2xthru, use_ifft=False)
```

```
54
55 # 3. Load your DUT measurement (with fixtures)
56 s_fixture_dut_fixture = rf.Network('lpf.s2p')
57
58 # 4. De-embed the fixture effects to get the actual DUT response
59 s_deembedded_dut = deembed_s_params(s_fixture_dut_fixture, s_side1, s_side2
       ↪ )
60
61 # 5. Save the de-embedded DUT S-parameters to a file
62 s_deembedded_dut.write_touchstone('lpf_ieee_deembedded_dut.s2p')
63
64 # Optional: Plot the results to verify
65 import matplotlib.pyplot as plt
66
67 plt.figure(figsize=(10, 6))
68 plt.title('S-parameters Before and After De-embedding')
69
70 # Plot S11 before and after de-embedding
71 plt.plot(s_fixture_dut_fixture.f/1e9, 20*np.log10(np.abs(
       ↪ s_fixture_dut_fixture.s[:,0,0])),
72          label='S11 with fixtures')
73 plt.plot(s_deembedded_dut.f/1e9, 20*np.log10(np.abs(s_deembedded_dut.s
       ↪ [:,0,0])),
74          label='S11 de-embedded')
75
76 # Plot S21 before and after de-embedding
77 plt.plot(s_fixture_dut_fixture.f/1e9, 20*np.log10(np.abs(
       ↪ s_fixture_dut_fixture.s[:,1,0])),
78          label='S21 with fixtures')
79 plt.plot(s_deembedded_dut.f/1e9, 20*np.log10(np.abs(s_deembedded_dut.s
       ↪ [:,1,0])),
80          label='S21 de-embedded')
81
82 plt.xlabel('Frequency (GHz)')
83 plt.ylabel('Magnitude (dB)')
84 plt.legend()
85 plt.grid(True)
86
87 plt.savefig('ieee_thru_sparameters_plot.png', dpi=300, bbox_inches='tight')
88
89 plt.show()
```

Code Listing G.4: **Lab 8: "2x Thru" S-Parameter De-Embedding Method (IEEE std. P370)**

```
import numpy as np
import skrf as rf
import scipy.signal as signal
import scipy.interpolate as interp
import warnings
from typing import Tuple, Optional, List, Union


def ieee_p370_2x_thru_alt(
    s_2xthru: rf.Network, use_ifft: bool = True
) -> Tuple[rf.Network, rf.Network]:
    """
    Creates error boxes from a test fixture 2x thru.

    Parameters
    ----------
    s_2xthru : skrf.Network
        An s parameter network object of the 2x thru
    use_ifft : bool, optional
        Whether to use the time domain method (True) or impedance method (
            ↪ False), by default True

    Returns
    -------
    s_side1 : skrf.Network
        An s parameter network object of the error box representing the
            ↪ half of the 2x thru connected to port 1
    s_side2 : skrf.Network
        An s parameter network object of the error box representing the
            ↪ half of the 2x thru connected to port 2
    """
    f = s_2xthru.f  # Frequency array
    s = s_2xthru.s  # S-parameters (shape: freq_points x 2 x 2)

    # ----------- main --------------------
    if use_ifft:
        # strip DC point if one exists
        if f[0] == 0:
            warnings.warn(
                "DC point detected. An interpolated DC point will be
                    ↪ included in the errorboxes."
            )
            flag_DC = 1
            fold = f.copy()
            f = f[1:]
            s = s[1:, :, :]
        else:
            flag_DC = 0

        # interpolate S-parameters if the frequency vector is not
            ↪ acceptable
        if f[1] - f[0] != f[0]:
            # set the flag
            flag_df = 1
            warnings.warn(
                "Non-uniform frequency vector detected. A spline
```

```
                ↪ interpolated S-parameter matrix will be created for
                ↪ this calculation. The output results will be re-
                ↪ interpolated to the original vector."
            )
            fold = f.copy()

            df = f[1] - f[0]
            projected_n = round(f[-1] / f[0])

            if projected_n <= 10000:
                if f[-1] % f[0] == 0:
                    fnew = np.arange(f[0], f[-1] + f[0], f[0])
                else:
                    fnew = np.arange(f[0], f[-1] - (f[-1] % f[0]) + f[0], f
                        ↪ [0])
            else:
                new_df = f[-1] / 10000
                fnew = np.arange(new_df, f[-1] + new_df, new_df)
                print(f"interpolating from {new_df}Hz to {f[-1]}Hz with
                    ↪ 10000 points.")

            snew = np.zeros((len(fnew), 2, 2), dtype=complex)
            for i in range(2):
                for j in range(2):
                    snew[:, i, j] = interp.interp1d(f, s[:, i, j], kind="
                        ↪ cubic")(fnew)

            s = snew
            f = fnew
        else:
            flag_df = 0

        n = len(f)
        s11 = s[:, 0, 0]

        # get e001 and e002
        # e001
        s21 = s[:, 1, 0]
        dcs21 = dc_interp(s21, f)
        t21 = np.fft.fftshift(
            np.fft.ifft(make_symmetric(np.vstack([dcs21, s21])), axis=0)
        )
        x = np.argmax(np.abs(t21))

        dcs11 = dc(s11, f)
        t11 = np.fft.fftshift(
            np.fft.ifft(make_symmetric(np.vstack([dcs11, s11])), axis=0)
        )
        step11 = make_step(t11)
        z11 = -50 * (step11 + 1) / (step11 - 1)
        z11x = z11[x]

        # Convert to new reference impedance
        temp = rf.Network(f=f, s=s, z0=50)
        temp.renormalize(z11x)
        sr = temp.s

        s11r = sr[:, 0, 0]
```

```
        s21r = sr[:, 1, 0]
        s12r = sr[:, 0, 1]
        s22r = sr[:, 1, 1]

        dcs11r = dc(s11r, f)
        t11r = np.fft.fftshift(
            np.fft.ifft(make_symmetric(np.vstack([dcs11r, s11r])), axis=0)
        )
        t11r[x:] = 0
        e001 = np.fft.fft(np.fft.ifftshift(t11r))
        e001 = e001[1 : n + 1]

        dcs22r = dc(s22r, f)
        t22r = np.fft.fftshift(
            np.fft.ifft(make_symmetric(np.vstack([dcs22r, s22r])), axis=0)
        )
        t22r[x:] = 0
        e002 = np.fft.fft(np.fft.ifftshift(t22r))
        e002 = e002[1 : n + 1]

        # calc e111 and e112
        e111 = (s22r - e002) / s12r
        e112 = (s11r - e001) / s21r

        # calc e01
        k = 1
        test = k * np.sqrt(s21r * (1 - e111 * e112))
        e01 = np.zeros(n, dtype=complex)
        for i in range(n):
            if i > 0:
                if np.angle(test[i]) - np.angle(test[i - 1]) > 0:
                    k = -1 * k
            e01[i] = k * np.sqrt(s21r[i] * (1 - e111[i] * e112[i]))

        # calc e10
        k = 1
        test = k * np.sqrt(s12r * (1 - e111 * e112))
        e10 = np.zeros(n, dtype=complex)
        for i in range(n):
            if i > 0:
                if np.angle(test[i]) - np.angle(test[i - 1]) > 0:
                    k = -1 * k
            e10[i] = k * np.sqrt(s12r[i] * (1 - e111[i] * e112[i]))

        # S-parameters are setup correctly
        if flag_DC == 0 and flag_df == 0:
            fixture_model_1r = np.zeros((n, 2, 2), dtype=complex)
            fixture_model_1r[:, 0, 0] = e001
            fixture_model_1r[:, 1, 0] = e01
            fixture_model_1r[:, 0, 1] = e01
            fixture_model_1r[:, 1, 1] = e111

            fixture_model_2r = np.zeros((n, 2, 2), dtype=complex)
            fixture_model_2r[:, 1, 1] = e002
            fixture_model_2r[:, 0, 1] = e10
            fixture_model_2r[:, 1, 0] = e10
            fixture_model_2r[:, 0, 0] = e112
```

```
        else:  # S-parameters are not setup correctly
            if flag_DC == 1:  # DC Point was included in the original file
                fixture_model_1r = np.zeros((n + 1, 2, 2), dtype=complex)
                fixture_model_1r[1:, 0, 0] = e001
                fixture_model_1r[0, 0, 0] = dc_interp(fixture_model_1r[1:,
                    ↪ 0, 0], f)
                fixture_model_1r[1:, 1, 0] = e01
                fixture_model_1r[0, 1, 0] = dc_interp(fixture_model_1r[1:,
                    ↪ 1, 0], f)
                fixture_model_1r[1:, 0, 1] = e01
                fixture_model_1r[0, 0, 1] = dc_interp(fixture_model_1r[1:,
                    ↪ 0, 1], f)
                fixture_model_1r[1:, 1, 1] = e111
                fixture_model_1r[0, 1, 1] = dc_interp(fixture_model_1r[1:,
                    ↪ 1, 1], f)

                fixture_model_2r = np.zeros((n + 1, 2, 2), dtype=complex)
                fixture_model_2r[1:, 1, 1] = e002
                fixture_model_2r[0, 0, 0] = dc_interp(fixture_model_2r[1:,
                    ↪ 0, 0], f)
                fixture_model_2r[1:, 0, 1] = e10
                fixture_model_2r[0, 1, 0] = dc_interp(fixture_model_2r[1:,
                    ↪ 1, 0], f)
                fixture_model_2r[1:, 1, 0] = e10
                fixture_model_2r[0, 0, 1] = dc_interp(fixture_model_2r[1:,
                    ↪ 0, 1], f)
                fixture_model_2r[1:, 0, 0] = e112
                fixture_model_2r[0, 1, 1] = dc_interp(fixture_model_2r[1:,
                    ↪ 1, 1], f)
                f = np.concatenate(([0], f))
            else:  # DC Point wasn't included in the original file, but the
                ↪  DF was not the same as f[0]
                fixture_model_1r = np.zeros((n, 2, 2), dtype=complex)
                fixture_model_1r[:, 0, 0] = e001
                fixture_model_1r[:, 1, 0] = e01
                fixture_model_1r[:, 0, 1] = e01
                fixture_model_1r[:, 1, 1] = e111

                fixture_model_2r = np.zeros((n, 2, 2), dtype=complex)
                fixture_model_2r[:, 1, 1] = e002
                fixture_model_2r[:, 0, 1] = e10
                fixture_model_2r[:, 1, 0] = e10
                fixture_model_2r[:, 0, 0] = e112

            if flag_df == 1:  # if df was different from f[0]
                # save the current error boxes
                fixture_model_1r_temp = fixture_model_1r.copy()
                fixture_model_2r_temp = fixture_model_2r.copy()
                # initialize the new errorboxes
                fixture_model_1r = np.zeros((len(fold), 2, 2), dtype=
                    ↪ complex)
                fixture_model_2r = np.zeros((len(fold), 2, 2), dtype=
                    ↪ complex)
                # interpolate the errorboxes to the original frequency
                    ↪ vector
                for i in range(2):
                    for j in range(2):
                        fixture_model_1r[:, i, j] = interp.interp1d(
```

```
                            f, fixture_model_1r_temp[:, i, j], kind="cubic"
                        )(fold)
                        fixture_model_2r[:, i, j] = interp.interp1d(
                            f, fixture_model_2r_temp[:, i, j], kind="cubic"
                        )(fold)

            # replace the vector used for the calculation with the original
              ↪ vector.
            f = fold

        # create the S-parameter objects for the errorboxes
        s_fixture_model_r1 = rf.Network(f=f, s=fixture_model_1r, z0=z11x)
        s_fixture_model_r2 = rf.Network(f=f, s=fixture_model_2r, z0=z11x)

        # renormalize the S-parameter errorboxes to the original reference
          ↪ impedance (assumed to be 50)
        s_side1 = s_fixture_model_r1.copy()
        s_side1.renormalize(50)
        s_side2 = s_fixture_model_r2.copy()
        s_side2.renormalize(50)
    else:
        # Use impedance method instead of time domain method
        # Convert S-parameters to Z-parameters
        z = rf.s2z(s_2xthru.s, s_2xthru.z0)
        ZL = np.zeros(z.shape, dtype=complex)
        ZR = np.zeros(z.shape, dtype=complex)

        for i in range(len(z)):
            ZL[i, 0, 0] = z[i, 0, 0] + z[i, 1, 0]
            ZL[i, 0, 1] = 2 * z[i, 1, 0]
            ZL[i, 1, 0] = 2 * z[i, 1, 0]
            ZL[i, 1, 1] = 2 * z[i, 1, 0]

            ZR[i, 0, 0] = 2 * z[i, 0, 1]
            ZR[i, 0, 1] = 2 * z[i, 0, 1]
            ZR[i, 1, 0] = 2 * z[i, 0, 1]
            ZR[i, 1, 1] = z[i, 0, 1] + z[i, 1, 1]

        # Convert Z-parameters back to S-parameters
        SL = rf.z2s(ZL, s_2xthru.z0)
        SR = rf.z2s(ZR, s_2xthru.z0)

        s_side1 = rf.Network(f=f, s=SL, z0=50)
        s_side2 = rf.Network(f=f, s=SR, z0=50)

    return s_side1, s_side2


# ------------- supporting functions -----------------


def make_symmetric(nonsymmetric):
    """
    Takes the nonsymmetric frequency domain input and makes it symmetric.
    The function assumes the DC point is in the nonsymmetric data.

    Parameters
    ----------
```

```
    nonsymmetric : ndarray
        Nonsymmetric frequency domain data

    Returns
    -------
    symmetric : ndarray
        Symmetric frequency domain data
    """
    symmetric_abs = np.vstack(
        [np.abs(nonsymmetric), np.flip(np.abs(nonsymmetric[1:]), axis=0)]
    )
    symmetric_ang = np.vstack(
        [np.angle(nonsymmetric), -np.flip(np.angle(nonsymmetric[1:]), axis
            ↪ =0)]
    )
    symmetric = symmetric_abs * np.exp(1j * symmetric_ang)
    return symmetric


def make_step(impulse):
    """
    Creates a step response from an impulse response.

    Parameters
    ----------
    impulse : ndarray
        Impulse response

    Returns
    -------
    step : ndarray
        Step response
    """
    ustep = np.ones(len(impulse))
    step = np.convolve(ustep, impulse)
    step = step[: len(impulse)]
    return step


def dc(s, f):
    """
    Calculates the DC point for S-parameters.

    Parameters
    ----------
    s : ndarray
        S-parameter data
    f : ndarray
        Frequency data

    Returns
    -------
    DCpoint : float
        DC point value
    """
    DCpoint = 0.002  # seed for the algorithm
    err = 1  # error seed
    allowedError = 1e-12  # allowable error
```

```
    cnt = 0
    df = f[1] - f[0]
    n = len(f)
    t = np.linspace(-1 / df, 1 / df, n * 2 + 1)
    ts = np.argmin(np.abs(t - (-3e-9)))
    Hr = com_receiver_noise_filter(f, f[-1] / 2)

    while err > allowedError:
        h1 = make_step(
            np.fft.fftshift(np.fft.ifft(make_symmetric(np.vstack([DCpoint,
                ↪ s * Hr]))))
        )
        h2 = make_step(
            np.fft.fftshift(
                np.fft.ifft(make_symmetric(np.vstack([DCpoint + 0.001, s *
                    ↪ Hr]))))
            )
        )
        m = (h2[ts] - h1[ts]) / 0.001
        b = h1[ts] - m * DCpoint
        DCpoint = (0 - b) / m
        err = np.abs(h1[ts] - 0)
        cnt += 1

    return DCpoint


def com_receiver_noise_filter(f, fr):
    """
    Receiver filter in COM defined by eq 93A-20.

    Parameters
    ----------
    f : ndarray
        Frequency data
    fr : float
        Reference frequency

    Returns
    -------
    Hr : ndarray
        Filter response
    """
    fdfr = f / fr
    Hr = 1.0 / (1 - 3.414214 * (fdfr) ** 2 + fdfr**4 + 1j * 2.613126 * (
        ↪ fdfr - fdfr**3))
    return Hr


def dc_interp(sin, f):
    """
    Enforces symmetry upon the first 10 points and interpolates the DC
        ↪ point.

    Parameters
    ----------
    sin : ndarray
        Input data
```

```
    f : ndarray
        Frequency data

    Returns
    -------
    dc : float
        DC point value
    """
    sp = sin[:10]
    fp = f[:10]

    snp = np.concatenate([np.conj(np.flip(sp)), sp])
    fnp = np.concatenate([-np.flip(fp), fp])
    fnew = np.concatenate([-np.flip(fp), [0], fp])
    snew = interp.interp1d(fnp, snp, kind="cubic")(fnew)
    dc = np.real(snew[len(sp)])

    return dc


def deembed_s_params(s_fixture_dut_fixture, s_side1, s_side2):
    """
    De-embeds the S-parameters of a DUT from measurements that include test
    ↪  fixtures.

    Parameters
    ----------
    s_fixture_dut_fixture : rf.Network
        S-parameter network object of the DUT with fixtures
    s_side1 : rf.Network
        S-parameter network object of the error box for port 1
    s_side2 : rf.Network
        S-parameter network object of the error box for port 2

    Returns
    -------
    s_deembedded_dut : rf.Network
        De-embedded S-parameter network object of the DUT
    """
    # Create T-parameters for the error boxes
    t_side1 = s_side1.s2t
    t_side2 = s_side2.s2t

    # Create T-parameters for the fixture+DUT+fixture
    t_fixture_dut_fixture = s_fixture_dut_fixture.s2t

    # De-embed the DUT
    t_dut = np.zeros(t_fixture_dut_fixture.shape, dtype=complex)

    # T_DUT = inv(T_side1) * T_fixture_dut_fixture * inv(T_side2)
    for i in range(len(t_fixture_dut_fixture)):
        t_dut[i] = (
            np.linalg.inv(t_side1[i])
            @ t_fixture_dut_fixture[i]
            @ np.linalg.inv(t_side2[i])
        )

    # Convert back to S-parameters
```

```
    s_deembedded_dut = s_fixture_dut_fixture.copy()
    s_deembedded_dut.s = rf.t2s(t_dut)

    return s_deembedded_dut
```

Code Listing G.5: **Lab 8: High-Pass Filter "Open-Short-Thru" S-Parameter De-Embedding Method (IEEE std. P370)**

```
import skrf as rf
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt

open_standard = rf.Network('Open.s2p')
short_standard = rf.Network('Short.s2p')
thru_standard = rf.Network('Thru.s2p')
dut = rf.Network('hpf.s2p')

def deembed_osp(dut, open_std, short_std, thru_std):
    thru_half = thru_std.copy()
    thru_half.s = np.zeros_like(thru_std.s)
    for i in range(len(thru_std.f)):
        s = thru_std.s[i]
        s_3d = s.reshape(1, *s.shape)
        t = rf.s2t(s_3d)
        t_half = scipy.linalg.sqrtm(t[0])
        t_half_3d = t_half.reshape(1, *t_half.shape)
        s_half = rf.t2s(t_half_3d)[0]
        thru_half.s[i] = s_half

    y_open = open_std.y
    z_short = short_std.z
    y_parasitic = y_open
    y_short = short_std.y
    y_short_corrected = y_short - y_parasitic
    z_parasitic = rf.network.y2z(y_short_corrected)

    dut_y = dut.y
    dut_y_corrected = dut_y - y_parasitic
    dut_z = rf.network.y2z(dut_y_corrected)
    dut_z_corrected = dut_z - z_parasitic

    dut_corrected = rf.Network(s=rf.z2s(dut_z_corrected),
                               frequency=dut.frequency)
    return dut_corrected

hpf_deembedded = deembed_osp(dut, open_standard, short_standard,
    ↪ thru_standard)
hpf_deembedded.write_touchstone('hpf_deembedded.s2p')

plt.figure(figsize=(12, 8))

plt.subplot(2, 1, 1)
plt.title('HPF - OST - S21 - Transmission')
dut.plot_s_db(m=0, n=1, label='Original')
hpf_deembedded.plot_s_db(m=0, n=1, label='OST De-embedded')
plt.grid(True)
plt.legend()

plt.subplot(2, 1, 2)
plt.title('HPF - OST De-Embed - S11 - Reflection')
dut.plot_s_db(m=0, n=0, label='Original')
hpf_deembedded.plot_s_db(m=0, n=0, label='OST De-embedded')
```

```
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.savefig('hpf_deembedded_comparison.png')
plt.show()
```

Code Listing G.6: **Lab 8: High-Pass Filter "Thru" S-Parameter De-Embedding Method (IEEE std. P370)**

```
import skrf as rf
import ieee_p370_2x_thru_alt
# from ieee_p370_2x_thru_alt import deembed_s_params
import numpy as np

def deembed_s_params(s_fixture_dut_fixture, s_side1, s_side2):
    """
    De-embeds the S-parameters of a DUT from measurements that include test fixtures.

    Parameters
    ----------
    s_fixture_dut_fixture : rf.Network
        S-parameter network object of the DUT with fixtures
    s_side1 : rf.Network
        S-parameter network object of the error box for port 1
    s_side2 : rf.Network
        S-parameter network object of the error box for port 2

    Returns
    -------
    s_deembedded_dut : rf.Network
        De-embedded S-parameter network object of the DUT
    """
    # Create T-parameters for the error boxes
    t_side1 = rf.s2t(s_side1.s)
    t_side2 = rf.s2t(s_side2.s)

    # Create T-parameters for the fixture+DUT+fixture
    t_fixture_dut_fixture = rf.s2t(s_fixture_dut_fixture.s)

    # De-embed the DUT
    t_dut = np.zeros(t_fixture_dut_fixture.shape, dtype=complex)

    # T_DUT = inv(T_side1) * T_fixture_dut_fixture * inv(T_side2)
    for i in range(len(t_fixture_dut_fixture)):
        t_dut[i] = (
            np.linalg.inv(t_side1[i])
            @ t_fixture_dut_fixture[i]
            @ np.linalg.inv(t_side2[i])
        )

    # Convert back to S-parameters
    s_deembedded_dut = s_fixture_dut_fixture.copy()
    s_deembedded_dut.s = rf.t2s(t_dut)

    return s_deembedded_dut

# 1. Load your 2x-thru calibration measurement
s_2xthru = rf.Network('Thru.s2p')

# 2. Generate the error boxes (fixture models) from the 2x-thru
# Modified to use use_ifft=False to avoid the problematic IFFT calculation
s_side1, s_side2 = ieee_p370_2x_thru_alt.ieee_p370_2x_thru_alt(s_2xthru, use_ifft=False)
```

```
54
55 # 3. Load your DUT measurement (with fixtures)
56 s_fixture_dut_fixture = rf.Network('hpf.s2p')
57
58 # 4. De-embed the fixture effects to get the actual DUT response
59 s_deembedded_dut = deembed_s_params(s_fixture_dut_fixture, s_side1, s_side2
       ↪ )
60
61 # 5. Save the de-embedded DUT S-parameters to a file
62 s_deembedded_dut.write_touchstone('hpf_ieee_deembedded_dut.s2p')
63
64 # Optional: Plot the results to verify
65 import matplotlib.pyplot as plt
66
67 plt.figure(figsize=(10, 6))
68 plt.title('HPF S-parameters Before and After De-embedding')
69
70 # Plot S11 before and after de-embedding
71 plt.plot(s_fixture_dut_fixture.f/1e9, 20*np.log10(np.abs(
       ↪ s_fixture_dut_fixture.s[:,0,0])),
72          label='S11 with fixtures')
73 plt.plot(s_deembedded_dut.f/1e9, 20*np.log10(np.abs(s_deembedded_dut.s
       ↪ [:,0,0])),
74          label='S11 de-embedded')
75
76 # Plot S21 before and after de-embedding
77 plt.plot(s_fixture_dut_fixture.f/1e9, 20*np.log10(np.abs(
       ↪ s_fixture_dut_fixture.s[:,1,0])),
78          label='S21 with fixtures')
79 plt.plot(s_deembedded_dut.f/1e9, 20*np.log10(np.abs(s_deembedded_dut.s
       ↪ [:,1,0])),
80          label='S21 de-embedded')
81
82 plt.xlabel('Frequency (MHz)')
83 plt.ylabel('Magnitude (dB)')
84 plt.legend()
85 plt.grid(True)
86 plt.show()
```