AD101 — Signals

Use notes and chords to learn to read signals in time and in frequency, and to see what a filter does to them.

UofT ASIC Team

Contents

1 AD101 — Signals 5

2 Getting started 9

I Movement I: A single note

3 What is a signal? 13

4 Pitch and loudness 15

5 A family of waveforms 17

6 Two voices in time 19

7 Lab 01: Signal explorer 21

II Movement II: Chords and timbre

8 Adding signals 31

9 Timbre and harmonics 33

10 Rhythm and duty cycle 35

11 Lab 02: Harmonic builder 37

III Movement III: The mixing board

12 The FFT as an equalizer 45

13 Finding a quiet note 49

14 Distortion and THD 51

15 Sampling fast enough 53

16 Lab 03: Spectrum detective 55

IV Movement IV: Tone controls

17 Filters are tone knobs 65

18 Reading a Bode plot 67
19 Filtering a beat 69
20 More filter shapes 71
21 Lab 04: RC filter and Bode 73

V Capstone

22 Lab 05: Signal workshop 83
Appendices 87
A Reading plots 89
B Music and signals 91
C Python in the workbench 93

1. AD101 — Signals

Use notes and chords to learn to read signals in time and in frequency, and to see what a filter does to them.

AD101 is the first course in the UofT ASIC Team's analog track, published at https://uoftasic.com/ad101/. It's for students who have finished IC101 and want to read the three plots analog engineers use every day: the waveform (voltage against time), the spectrum (how strong each frequency is) and the Bode plot (what a circuit does to each frequency).

The course uses music as its running example. A single signal plays the part of a note, a chord is several signals added together, timbre is the recipe of harmonics in a sound, and a filter works like a tone knob. You don't need any musical background to follow it.

Before you start

Prerequisite	IC101: Docker and the IIC-OSIC-TOOLS noVNC desktop working, with the shared `workspace` repo from IC101 cloned. The labs run inside that workspace.
Math	High-school level: sines, logarithms and basic algebra. You won't need calculus.
Tools	Python, numpy and matplotlib, which are already installed in IIC-OSIC-TOOLS
Time	About 10 hours 20 minutes: 2 hours 50 minutes of reading and 7.5 hours of labs, not counting the optional spectral-leakage part at the end of Lab 03

What you'll learn

1. Read a voltage against time the way you'd read an oscilloscope: amplitude, frequency, phase, DC offset
2. Build complicated waveforms from simple sines, and see their spectra
3. Use the frequency domain to find tones that time plots hide
4. Read a Bode plot and watch an RC filter reshape a square wave
5. Write your own signal-analysis script, and listen to what a filter does to a chord

Course path

The guide pages are grouped into four units, which the course calls movements, and each movement is followed by a lab. A capstone lab comes last. Work down this table in order. The Read rows are guide pages, with concepts, figures and worked examples but nothing for you to run; the Do rows are labs, which is where you run everything. The times are estimates, and they add up to the total above.

#		What	Time
1	Read	Getting started, then Movement I, a single note: What is a signal? · Pitch and loudness · A family of waveforms · Two voices in time	50 min
2	Do	Lab 01: Signal explorer. Load AD101, then pre-	1.5 h

#		What	Time
		dict and measure a sine's period, peak-to-peak, RMS and phase delay.	
3	Read	Movement II, chords and timbre: Adding signals · Timbre and harmonics · Rhythm and duty cycle	30 min
4	Do	Lab 02: Harmonic builder. Build a square wave from sines, predict its harmonic amplitudes, and watch duty cycle change its spectrum.	1 h
5	Read	Movement III, the mixing board: The FFT as an equalizer · Finding a quiet note · Distortion and THD · Sampling fast enough	45 min
6	Do	Lab 03: Spectrum detective. Find a hidden tone and check your answer with a script, measure total harmonic distortion (THD), and make a tone alias.	1.5 h
7	Read	Movement IV, tone controls: Filters are tone knobs · Reading a Bode plot · Filtering a beat · More filter shapes	45 min
8	Do	Lab 04: RC filter and Bode. Predict an RC low-pass's cutoff and gain, check them on a Bode plot, and filter a square wave.	1.5 h
9	Do	Lab 05: Signal workshop. Write your own chord synthesizer, spectrum plot and filter, then listen to the result.	2 h

Three reference pages are worth keeping open while you work: Reading plots is a cheat sheet for the three plots, Music and signals translates each musical term and lists note frequencies, and Python in the workbench covers running the labs and what to do when a plot window won't open.

After AD101

- AD102: Linear Circuits & Fabrication
- AD103: Nonlinear Circuits (XSchem)
- AD104: Layout (Magic / Netgen)

2. Getting started

AD101 is a primer on **signals**, the voltages that move through every analog circuit. By the end of it you'll be able to read the three plots an analog engineer works with most: the **waveform**, which shows voltage against time; the **spectrum**, which shows how strong each frequency is; and the **Bode plot**, which shows what a circuit does to a signal.

We start from something you already know, which is music. A signal is a voltage that wiggles the same way sound pressure wiggles when you hum a note, and a chord is several of those signals added together. An instrument's timbre comes from the particular recipe of extra harmonics it adds on top of the main pitch, and a filter does the same job as a tone knob. Every idea in this course has a musical twin like these, and the guide uses them on purpose.

You don't need a musical background to follow along. Each analogy is explained the first time it's used, and Music and signals collects them all in one table. The math is high-school level (sine, logarithm and basic algebra) and there's no calculus.

How the course is organized

The course has four movements followed by a capstone lab, Lab 05. Each movement is a few short guide pages followed by a lab, and the Course path table on the home page lists every step in order with a time estimate.

Guide pages are for reading: they hold the concepts, figures and worked examples, and you don't need a terminal open while you read them. The labs are where you run things, and they teach each idea in two ways:

1. **Drag it.** Labs 01–04 come with interactive explorers, which are plot windows with sliders, numbered F1 to F15 (F for figure). Dragging a slider and watching the plot update is the fastest way to build intuition for how a knob changes a curve.
2. **Write it.** The same labs give you short Python snippets that you save, run and change yourself. Lab 05: Signal workshop has no sliders at all. In that lab you put those ideas together into your own signal-analysis script, end to end, and listen to what a filter does to a chord.

It's worth doing both. Dragging a slider builds intuition quickly, but writing the script is what lets you apply that intuition to a signal nobody has built a slider for.

The figures on these guide pages are the PNG files that each lab's explorer script writes in its `--headless` mode, so you can make any of them yourself. The live explorer windows add sliders and open on their own starting settings, so they won't look exactly like the figures here.

If an explorer window won't open, the same `--headless` mode gets you its figures as PNG files instead. Python in the workbench explains that mode and covers the other common problems, and Reading plots is a one-page cheat sheet for all three kinds of plot.

Out of scope

- AD101 doesn't replace the ECE signals courses.
- It doesn't go deep into Fourier proofs or complex numbers.
- It doesn't use XSchem or ngspice yet; you'll start on those in AD102 and AD103.

You'll load AD101 into your IC101 workspace in Part 1 of Lab 01.

Next: What is a signal?

I

Movement I: A single note

3 **What is a signal? 13**
3 The oscilloscope 13
3 A signal as two lists of numbers 13

4 **Pitch and loudness 15**
4 The sine wave's knobs 15
4 Two ways to measure "how loud" 16
4 Why an engineer cares 16

5 **A family of waveforms 17**
5 Four more shapes to know by sight 17
5 Why an engineer cares 18

6 **Two voices in time 19**
6 Phase as a time delay 19
6 Adding two offset voices 19
6 Why an engineer cares 19

7 **Lab 01: Signal explorer 21**
7 Before you start 21
7 Part 1: Load the course 21
7 Part 2: Plot your first signal 22
7 Part 3: Measure a sine (F1) 23
7 Part 4: Compare waveform shapes (F2) 24
7 Part 5: Turn phase into delay (F3) 26
7 Finishing 27
7 Answers 27

3. What is a signal?

Hum a single note and hold it. Your vocal cords push the air back and forth, the same number of times every second. If you plotted how far the air is pushed against time, you'd draw a wiggly line that goes up and down and repeats.

That line is a **signal**, and on a chip the thing that wiggles is a voltage instead of air. Every wire, pin and node carries a voltage that can move up and down over time, just like the air pressure of your hummed note. Once you can describe that voltage (how big it swings, how fast it repeats, where it sits) you can start reasoning about the circuit it lives in.

The oscilloscope

A microphone turns your hummed note into a voltage that wiggles the same way the air did. An **oscilloscope** is the instrument that draws that voltage against time, the way sheet music lays pitches out along a timeline. When you look at a scope trace, you're reading the "score" of whatever voltage the probe is touching. This course is about getting fluent at reading that score, and later at writing it.

A signal as two lists of numbers

Without the instrument and the circuit, what's left of a signal is two lists of numbers:

- a list of **times**, and
- a list of **voltages**, one for each time.

You don't need calculus to work with a signal in this form. In Python, the two lists are two `numpy` arrays:

```
import numpy as np

t = np.arange(0, 0.05, 1 / 10_000)   # 50 ms of timestamps, 10,000 per second
v = np.sin(2 * np.pi * 440 * t)      # a voltage at each of those timestamps
```

`t` and `v` are the same length, and `v[i]` is the voltage at time `t[i]`. Every plot, measurement and script in this course starts from a pair of arrays like these.

This pair describes one held note: a sine wave at 440 Hz, the note A4 that orchestras tune to. If you plot `v` against `t`, you get what an oscilloscope would show for that note. You'll write and plot it yourself in Lab 01, Part 2.

Next: Pitch and loudness, which gives you the words you'll use for the rest of the course to describe these two arrays: amplitude, frequency, phase and DC offset.

4. Pitch and loudness

The held note from the last page is a sine wave, the simplest interesting signal there is. Almost every other signal in this course is built out of sines, so it's worth getting comfortable with the sine's four knobs before you move on.

The sine wave's knobs

$$v(t) = A\sin(2\pi f t + \phi) + V_{\mathrm{DC}} \tag{4.1}$$

Each symbol in the formula is a knob you could turn on a synthesizer. The table also lists the period T, which isn't a separate knob, because the frequency sets it:

Symbol	Name	Musical intuition	What it means
A	Amplitude	Loudness	How far the wave swings from its center (volts)
f	Frequency	Pitch	How many cycles the wave completes per second (hertz, Hz)
$T = 1/f$	Period	Length of one "beat"	How long one cycle lasts (seconds)
ϕ	Phase	Where in the beat you start	How far the wave is shifted sideways, which sets where in its cycle it starts
V_{DC}	DC offset	No musical equivalent (a mixing desk calls this "bias")	The center line the wave rides on

If you turn the amplitude knob up, the note gets louder but its pitch stays the same. If you turn the frequency knob up, the pitch rises but the loudness stays the same. The two are independent, which is why the formula gives them separate symbols.

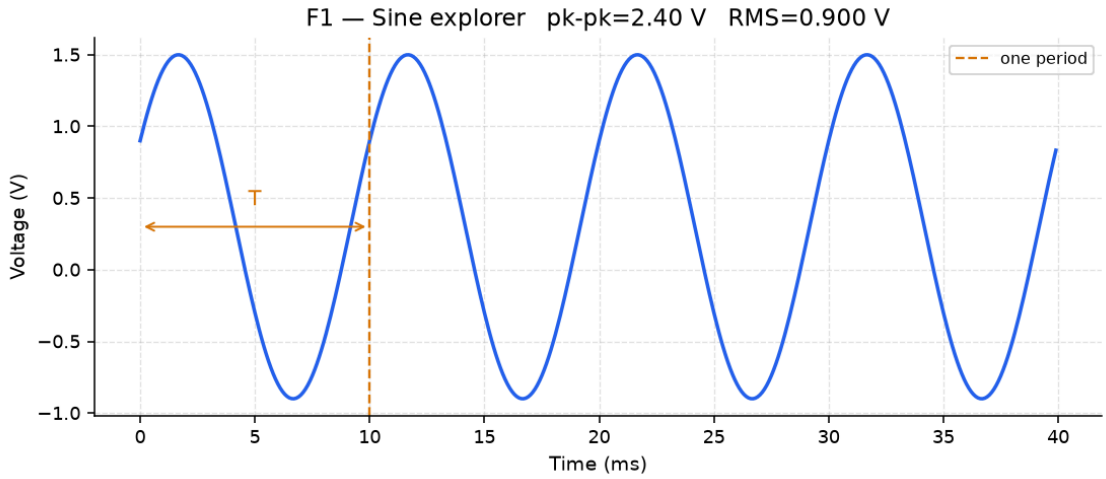


Figure 4.1: F1 — Sine explorer

Two ways to measure “how loud”

- **Peak-to-peak** is the full vertical swing of the wave, from its highest point to its lowest. For a pure sine it’s $2A$, because the wave swings A above its center line and A below it.
- **RMS** (“root mean square”) measures the signal’s effective strength. For a sine with no DC offset, $\text{RMS} = A/\sqrt{2} \approx 0.707\,A$. The power a signal delivers into a resistor depends on its RMS value, not its peak, which is why a multimeter reports RMS rather than amplitude when you measure an AC voltage.

In Lab 01, Part 3, you’ll read the period, peak-to-peak and RMS off the F1 explorer, and then compute the same three numbers in your own script, working out the period from $T = 1/f$ and measuring the other two from an array of samples.

Why an engineer cares

Outside music, the same quantities come up all the time in circuit work:

Quantity	Everyday use
Amplitude compared with the supply rails	Headroom: whether the signal will clip
DC offset	The bias, or operating point, of a circuit (AD103 will make this precise)
Frequency	Bandwidth budgets and sampling rates
RMS	The number your multimeter reports and your power calculations use

Next: A family of waveforms

5. A family of waveforms

Not every signal looks like a sine. A flute, a snare drum and radio static all make sound, but each has a different waveform, and the signals inside a chip don't all share one shape either.

Four more shapes to know by sight

Shape	Where you'll meet it	What it looks like
Square	Digital clocks	Snaps between two levels, with nothing in between
Triangle	ADC ramps, test signals	Straight-line rise, straight-line fall
Pulse	Digital clocks with a tunable on-time (the **duty cycle**)	A square that spends more time low than high, or the other way round
Noise	Sensor readouts, the thermal noise floor	Never repeats, so it has no period at all

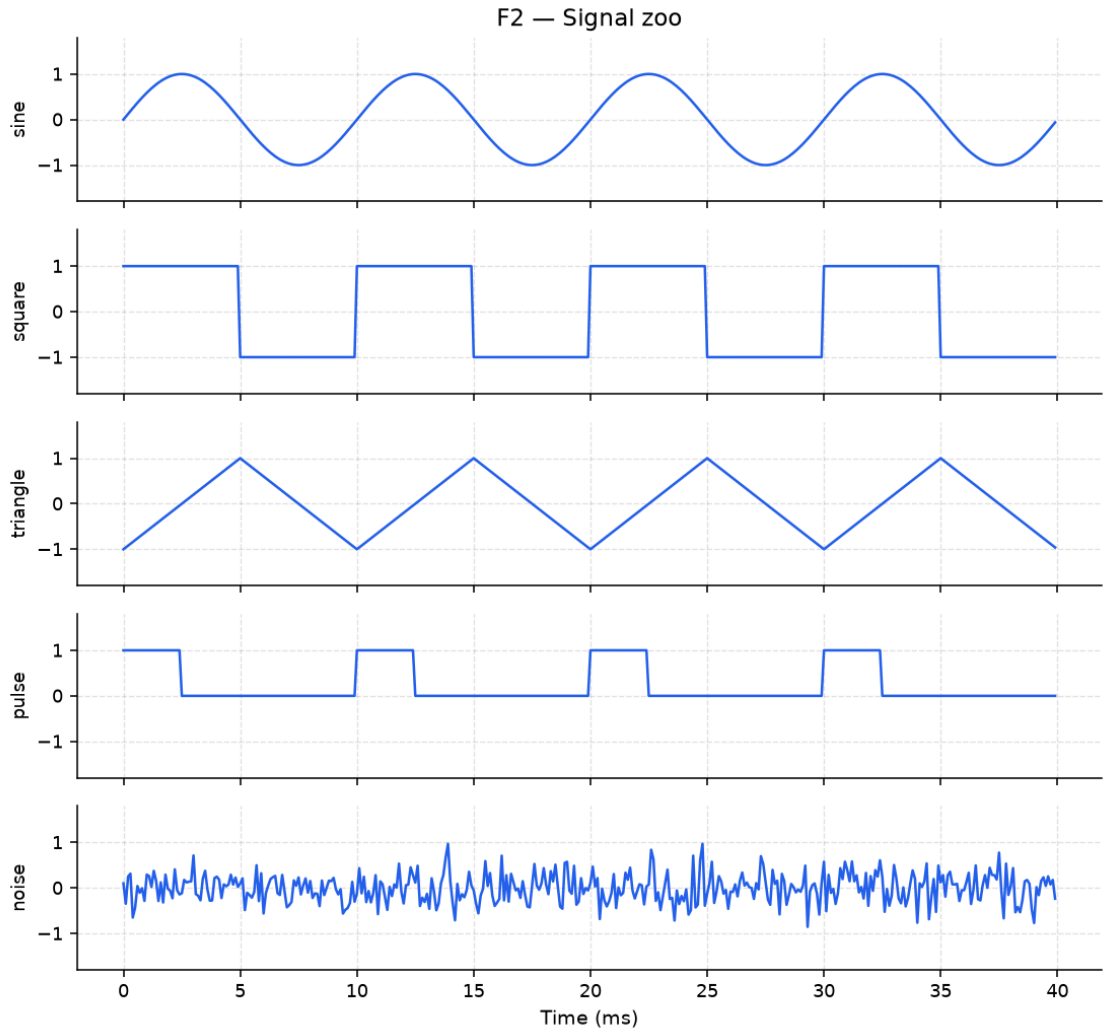


Figure 5.1: F2 — Signal zoo

The F2 explorer drives every shape from the same amplitude setting, and they still look completely different. That's why you can't judge a trace by its height alone: two waveforms can have the same height on a datasheet and still carry completely different information.

You can build the square, pulse and triangle from the same time array `t` you met on What is a signal?, using a phase that counts from 0 to 1 once per cycle. If you compare the phase to a threshold you get a square or a pulse, and if you ramp it up and then down you get a triangle. You'll build those three this way in Lab 01, Part 4, along with a noise trace, and the duty cycle comes back in Rhythm and duty cycle.

Why an engineer cares

- The clocks in a digital chip are square waves.
- The ramps an ADC uses are triangle waves.
- The noise trace is what the **thermal noise floor** looks like: the chip's own randomness, which sets the smallest signal the chip can resolve.

Next: Two voices in time

6. Two voices in time

Suppose two singers hold the same note at the same pitch and the same loudness, but one of them comes in a fraction of a beat late, so the two voices are offset in time. That offset is called **phase**, and it shows up wherever two signals of the same frequency meet: in two clock domains, in the two channels of a stereo signal, or on two probes of the same scope.

Phase as a time delay

$$t_{\text{delay}} = \frac{\phi}{360^\circ} \cdot T \tag{6.1}$$

A phase difference is a time delay, measured in degrees of one cycle instead of in seconds. A 90° shift is a quarter of a cycle, so at 100 Hz, where the period is 10 ms, it's a 2.5 ms delay. It works the way "a quarter note late" does in music: the same fraction of a beat is a different amount of time at a fast tempo than at a slow one, so you need the frequency before you can turn a phase into a delay.

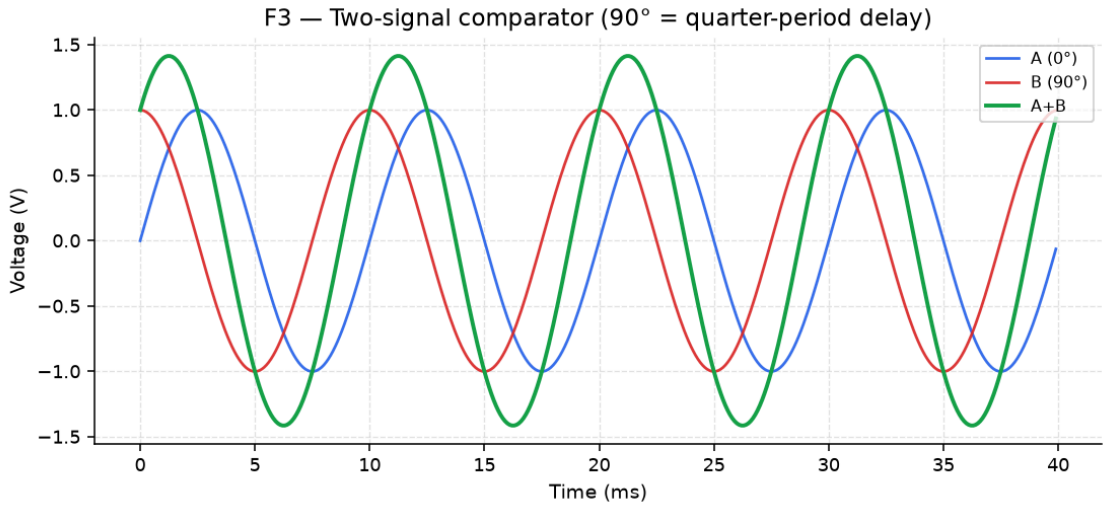


Figure 6.1: F3 — Two-signal comparator

Adding two offset voices

When you add two equal sines, the phase between them decides what the sum looks like. At 0° the two voices reinforce each other and the sum is tall. As the phase approaches 180° they cancel more and more, and the sum flattens out, which is what happens to the sound from two speakers wired backwards.

You'll set the phase on the F3 explorer, check the delay readout against the formula, and plot the two voices and their sum yourself in Lab 01, Part 5.

Why an engineer cares

Quantity	Everyday use
Phase / delay	Timing, synchronization, and the reason digital timing closure exists

Quantity	Everyday use
Phase at high frequency	Clock skew between two paths that “should” arrive together
Phase margin	A preview of stability analysis in AD201

This page ends Movement I. Next comes Lab 01: Signal explorer. After the lab, Movement II starts with Adding signals.

7. Lab 01: Signal explorer

You'll load AD101 into your workspace, write your first signal scripts, and use three slider explorers (F1–F3) to measure a sine's period, peak-to-peak and RMS, compare common waveform shapes, and turn a phase difference into a time delay. Before each script runs, you'll write down the numbers you expect it to print.

Lab package: `labs/lab-01-signal-explorer/`

Before you start

- Read Getting started and Movement I: What is a signal?, Pitch and loudness, A family of waveforms and Two voices in time.
- Keep Reading plots open as a cheat sheet.
- Have a sheet of paper or a text file for your predictions.

Time: about 1.5 hours.

How the labs work

Part 1 loads the course and Part 2 is a snippet on its own. Parts 3 to 5 each pair an explorer with a snippet, and the list below explains what each of those is.

- An **explorer** is a plot window with sliders, which you open with `python3 src/explore.py --figure <id>`. The terminal stays busy until you close the window.
- A **snippet** is a short Python script that you save as a new `.py` file in the lab folder and run with `python3 <file>.py`. The steps suggest a file name for each one, but any name works. Snippets use only numpy and matplotlib, and the desktop has `gedit`, `nano` and `vim` for editing them.

Run explorers and snippets from the lab folder, which Part 2 takes you to. If you open a new terminal partway through, run `. /foss/designs/common/.designinit`, `mod ad101` and `cd labs/lab-01-signal-explorer` first, to load the environment and get back to the lab folder. Your snippet files are under `/foss/designs`, which is your `workspace` folder on the host, so they survive a container restart.

Part 1: Load the course

If you haven't set up the workbench, do IC101 Lab 03 first.

1. Start the desktop from your `workspace` folder on the host (`./scripts/start_vnc.sh`, or `scripts\start_vnc.bat` on Windows), open it in your browser as in IC101, and open a terminal in the desktop.
2. Load the workspace environment and get AD101:

```
. /foss/designs/common/.designinit
mod add ad101    # first time only: clones github.com/uoftasic/ad101 into modules/
ad101
mod ad101        # every time after that
```

You should see `cwd: /foss/designs/modules/ad101` after `mod ad101`.

If you'd rather clone AD101 on the host, you can run the commands below instead of `mod add ad101`, from the folder that contains your `workspace` clone. The clone ends up in the same place that `mod add ad101` would put it, so afterwards you run `mod ad101` in the desktop terminal as above:

```
cd workspace/modules
git clone https://github.com/uoftasic/ad101.git
```

3. Check the Python environment, including the numpy and matplotlib versions:

```
python3 scripts/check_env.py
```

You should see:

```
=== AD101 environment check ===
Python: 3.12.3  (/usr/bin/python3)
OK   numpy 2.5.2
OK   matplotlib 3.11.1
OK   matplotlib backend: qtagg
OK   interactive GUI backend available
OK   adsig smoke: peak ≈ 100 Hz (expect ~100)

=== All checks passed ===
```

We checked the outputs printed in Labs 01–05 with these numpy and matplotlib versions, in the IIC-OSIC-TOOLS 2026.08 image that the workbench runs. If your versions differ and a printed number disagrees with these pages, the difference may come from the version rather than from your code. The backend is the part of matplotlib that opens plot windows, and the `matplotlib backend:` line names the GUI toolkit that matplotlib found to do it. `adsig` is the course's small signal library, in `labs/common/`, and the explorers use it, so the `adsig smoke` line checks that it works.

If `check_env.py` prints one of these lines, here's what to do:

What you see	What to do
`WARN no interactive GUI backend (Agg / headless)`	matplotlib has no way to open a window, so the explorers won't work. First check that you ran `check_env.py` in a terminal in the noVNC desktop, not on your host. If you did, see "If the GUI won't open" in Python in the workbench.
`FAIL numpy` or `FAIL matplotlib`	Run `pip install --user numpy matplotlib` in the desktop terminal, then run `check_env.py` again. The packages are installed inside the container rather than in your `workspace` folder, so you'll have to repeat the install if you remove the container.

For anything else about the desktop, Docker or `mod`, see IC101's Troubleshooting page.

Part 2: Plot your first signal

What does the pair of arrays from What is a signal? look like as a plot?

1. Go to the lab folder:

```
cd labs/lab-01-signal-explorer
```

2. Save this as `first_signal.py`. It's the pair of arrays from What is a signal?, plus four lines that plot them:

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

t = np.arange(0, 0.05, 1 / 10_000)   # 50 ms of timestamps, 10,000 per second
v = np.sin(2 * np.pi * 440 * t)      # a voltage at each of those timestamps

plt.plot(t * 1e3, v)          # ×1e3 to show milliseconds instead of seconds
plt.xlabel("Time (ms)")
```

```
plt.ylabel("Voltage (V)")
plt.show()
```

3. Before you run it, work out how many full cycles of a 440 Hz sine fit in the 50 ms window and write the number down. Then run it with `python3 first_signal.py`.

You should see a sine wave swinging between −1 V and +1 V across 0 to 50 ms. It's one held note, A4 at 440 Hz, and 22 cycles of it fit in the window. Close the window to get the terminal back.

Part 3: Measure a sine (F1)

How does each of a sine's four knobs (amplitude, frequency, phase and DC offset) change its period, peak-to-peak and RMS?

1. Open the sine explorer:

```
python3 src/explore.py --figure f1
```

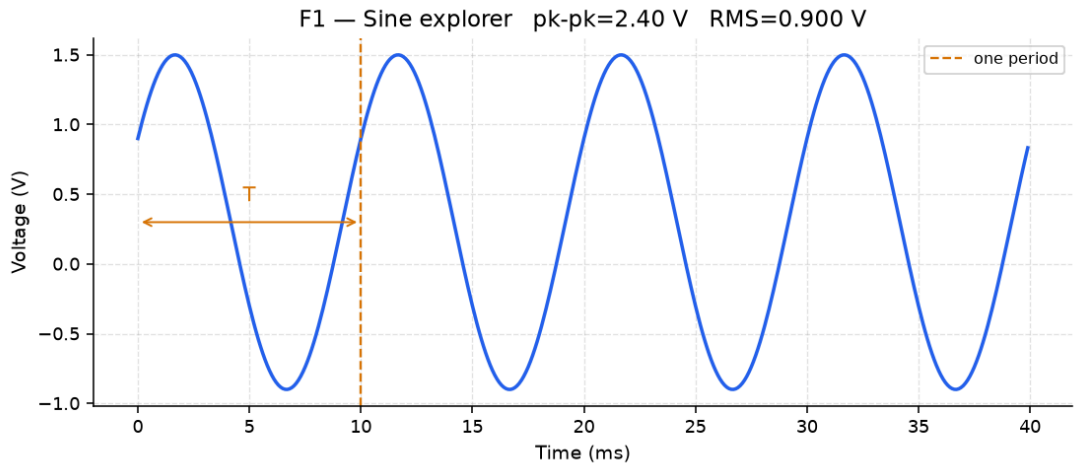


Figure 7.1: F1

2. Leave the sliders at their starting values (amplitude 1, 100 Hz, phase 0°, DC offset 0) and read the box in the top-left corner of the plot.

   You should see `T = 10.00 ms  f = 100 Hz` and `pk-pk = 2.00 V  RMS = 0.707 V`. The dashed vertical line marks one period.

3. Drag Amplitude, Freq (Hz), Phase (°) and DC offset one at a time, putting each back to its starting value before you move the next. Before each drag, write down what you expect the readout to do. The formulas $T = 1/f$, peak-to-peak $= 2A$ and RMS $= A/\sqrt{2}$ tell you what Amplitude and Freq will do; for Phase and DC offset you'll have to make a guess.

   You should see the period marker track $T = 1/f$, RMS stay at about 0.707× the amplitude while DC is 0, Phase slide the wave sideways, and the DC slider lift the whole wave without changing the size of the wiggle. With DC offset at 0.5 and the other sliders at their starting values, the box reads `pk-pk = 2.00 V  RMS = 0.866 V`.

   Why does RMS go up when you add DC, while pk-pk stays the same?

   Answer: A1, in Answers at the end.

   Close the window.

4. Save this as `measure_sine.py`, but don't run it yet. It builds a sine at a chosen amplitude and frequency, computes the period from $T = 1/f$, and measures peak-to-peak and RMS from the samples in the array, rather than off a scope trace by eye:

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

fs = 10_000                          # samples per second
t = np.arange(0, 0.05, 1 / fs)       # 50 ms window

A, f = 1.5, 220.0                    # amplitude (V), frequency (Hz) — the note A3
v = A * np.sin(2 * np.pi * f * t)

period_s = 1 / f
peak_to_peak = v.max() - v.min()
rms = np.sqrt(np.mean(v**2))

print(f"T = {period_s*1e3:.2f} ms   pk-pk = {peak_to_peak:.2f} V   RMS = {rms:.3f} V")

plt.plot(t * 1e3, v)
plt.axvline(period_s * 1e3, color="orange", linestyle="--", label="one period")
plt.xlabel("Time (ms)")
plt.ylabel("Voltage (V)")
plt.legend()
plt.show()
```

Before you run it, use the three formulas from step 3 to work out what it will print for A = 1.5 V and f = 220 Hz, and write the numbers down. Then run it.

You should see `T = 4.55 ms    pk-pk = 3.00 V    RMS = 1.061 V` printed, and a plot with an orange dashed line at one period.

5. Pick your own values of `A` and `f`. Write down the T, pk-pk and RMS you expect, then change the two values in the script and re-run. Keep your numbers and the printout for the Finishing section at the end of this page.

   You should see your predicted numbers in the printout. If a number is off by more than its last digit, recheck your arithmetic and your edit before you move on.

   If a number is off in its last digit, why?

   Answer: A2, in Answers at the end.

Part 4: Compare waveform shapes (F2)

How do square, triangle, pulse and noise differ from a sine, and what does the duty cycle change?

1. Open the signal zoo:

```
python3 src/explore.py --figure f2
```

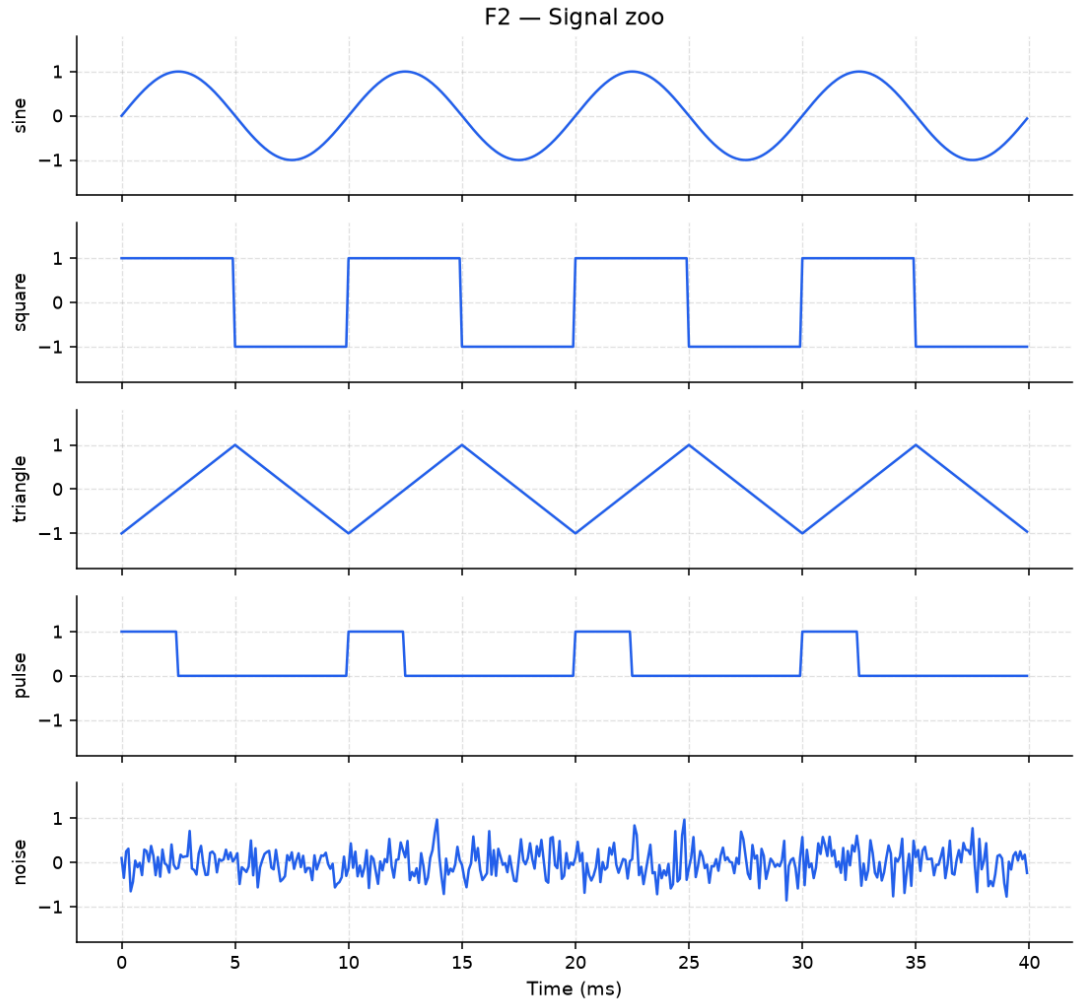


Figure 7.2: F2

2. Click through sine, square, triangle, pulse and noise on the left. With pulse selected, sweep the Duty slider (it starts at 0.25), and compare 0.25 with 0.5.

   You should see five very different shapes from the same Amplitude setting, and noise is the only one that never repeats. At 25% duty the pulse looks clearly lopsided next to the pulse at 50%. Close the window.

3. Save this as `waveform_zoo.py` and run it. It builds a sine, which it doesn't plot, and then square, pulse and triangle waves from the same time array `t`, using a `phase` variable that counts from 0 to 1 once per cycle. It also adds a noise trace:

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

fs = 10_000
t = np.arange(0, 0.02, 1 / fs)      # 20 ms window
f = 200.0                           # Hz

sine = np.sin(2 * np.pi * f * t)

# phase goes 0 -> 1 once per cycle; comparing it to a threshold builds
# square, pulse, and triangle without any new math
phase = (f * t) % 1.0
```

```
square = np.where(phase < 0.5, 1.0, -1.0)        # snap high/low at the halfway point
pulse = np.where(phase < 0.25, 1.0, 0.0)              # on 25% of the cycle (duty
= 0.25)
triangle = np.where(phase < 0.5, -1 + 4 * phase, 3 - 4 * phase)   # ramp up, ramp down
noise = 0.3 * np.random.default_rng(0).standard_normal(len(t))

fig, axes = plt.subplots(4, 1, sharex=True, figsize=(7, 6))
for ax, y, name in zip(axes, [square, pulse, triangle, noise], ["square", "pulse
(25%)", "triangle", "noise"]):
    ax.plot(t * 1e3, y)
    ax.set_ylabel(name, fontsize=9)
axes[-1].set_xlabel("Time (ms)")
plt.tight_layout()
plt.show()
```

You should see four stacked plots (square, pulse at 25%, triangle and noise) over a 20 ms window.

4. Change the `0.25` in the `pulse` line to `0.5`, then to `0.75`, re-running each time.

   You should see the on-time of each pulse grow. The number you changed is the duty cycle, the fraction of each cycle that the pulse is on, and you'll meet it again in Rhythm and duty cycle.

Part 5: Turn phase into delay (F3)

How long is a phase difference in milliseconds, and what does it do to the sum of two equal sines?

1. Open the two-signal comparator:

```
python3 src/explore.py --figure f3
```

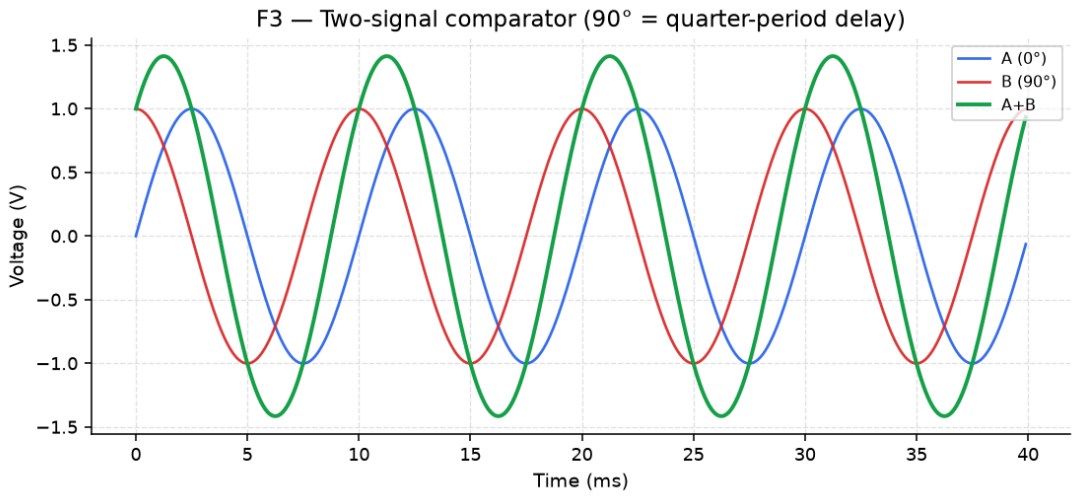


Figure 7.3: F3

The explorer starts at Phase Δ = 60° and 100 Hz. The Phase Δ slider sets the phase difference ϕ between the two signals, which the readout labels Δφ. Signal A is $\sin(2\pi f t)$ and signal B is $\sin(2\pi f t + \phi)$, so B is offset from A in time by $t_{\text{delay}} = \frac{\phi}{360^\circ} T$. With a positive ϕ, B runs ahead of A by t_{delay}, and the readout shows that time as `delay`.

2. Before you move anything, use the formula to predict the delay for a 90° phase difference at 100 Hz, and write it down.
3. Set Phase Δ to 90° and read the delay in the box.

   You should see `delay = 2.500 ms`, a quarter of the 10 ms period.

4. Sweep Phase Δ from 0° to 180° and watch the green sum A+B.

   You should see the sum at its tallest at 0° and cancelled to a flat line at 180°. Close the window.

5. Save this as `two_voices.py` and run it:

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

fs = 10_000
t = np.arange(0, 0.02, 1 / fs)
f = 100.0                       # Hz
phase_deg = 90.0                 # try 0, 60, 180, 360

voice_a = np.sin(2 * np.pi * f * t)
voice_b = np.sin(2 * np.pi * f * t + np.deg2rad(phase_deg))
together = voice_a + voice_b

delay_ms = (phase_deg / 360.0) * (1000.0 / f)
print(f"delay = {delay_ms:.3f} ms  (period = {1000.0/f:.2f} ms)")

plt.plot(t * 1e3, voice_a, label="voice A")
plt.plot(t * 1e3, voice_b, label="voice B")
plt.plot(t * 1e3, together, label="A + B", linewidth=2)
plt.xlabel("Time (ms)")
plt.ylabel("Voltage (V)")
plt.legend()
plt.show()
```

   You should see `delay = 2.500 ms  (period = 10.00 ms)`, the same delay F3 showed.

6. Predict the delay the script will print for `phase_deg` = 0, 60, 180 and 360, and write the four numbers down. Then set `phase_deg` to each in turn and re-run.

   You should see `delay = 0.000 ms`, `delay = 1.667 ms`, `delay = 5.000 ms` and `delay = 10.000 ms`. The `A + B` trace is tall at 0° and flat at 180°, and 360° looks the same as 0°.

   Why is the sum flat at 180°, and why does 360° look like 0° when the delay is 10 ms?

   Answer: A3, in Answers at the end.

Finishing

You're done when `measure_sine.py`, run with an A and f you chose, prints the T, pk-pk and RMS you wrote down before running it, to within the last digit (Part 3, step 5), and `two_voices.py` prints the four delays you predicted (Part 5, step 6).

Questions

1. Why does RMS go up with a DC offset while peak-to-peak stays the same? (Answered in Part 3, step 3.)
2. Why does the sum of two equal sines cancel at 180°, and come back at 360°? (Answered in Part 5, step 6.)

Submit

Nothing to submit.

Keep your snippet files anyway, because Lab 05 asks you to write the same kinds of pieces again and assemble them into one script.

Next on the Course path: read Movement II, Adding signals, Timbre and harmonics and Rhythm and duty cycle, then do Lab 02: Harmonic builder.

Answers

A1. Part 3: Measure a sine (F1)

RMS is the square root of the mean of v^2 over the window. A DC offset shifts every sample by the same amount, which changes every v^2, so the offset shows up in the RMS. Peak-to-peak is the maximum minus the minimum, and because the shift moves both of them by the same amount, it cancels out of the difference.

A2. Part 3: Measure a sine (F1)

The script measures the sampled array rather than evaluating the formulas, and a finite set of samples doesn't always match them exactly. If $f \times 0.05$ s isn't a whole number, the 50 ms window ends partway through a cycle and RMS comes out slightly off; for example, A = 1, f = 333 Hz prints `RMS = 0.705 V`. If no sample lands exactly on a peak, pk-pk comes out a little short; A = 2, f = 200 Hz prints `pk-pk = 3.99 V`.

A3. Part 5: Turn phase into delay (F3)

At 180°, B is offset from A by half a period, 5 ms at 100 Hz, so B is at its lowest point whenever A is at its highest. In other words B is $-A$, so the two cancel. At 360°, the offset is a whole period, 10 ms, and a sine shifted by one full period lands back on itself, so B is A again and the two reinforce each other as they do at 0°.

II Movement II: Chords and timbre

8 Adding signals 31
8 Superposition 31
8 Reading a sum by eye 31
8 Why an engineer cares 31

9 Timbre and harmonics 33
9 The square wave's recipe 33
9 The frequency domain 33
9 Why an engineer cares 34

10 Rhythm and duty cycle 35
10 Duty cycle and the recipe 35
10 Why an engineer cares 35

11 Lab 02: Harmonic builder 37
11 Before you start 37
11 Part 1: Build a chord 37
11 Part 2: Build a square wave from sines (F4/F5) . . 38
11 Part 3: Duty cycle and even harmonics (F6) . . . 39
11 Finishing 41
11 Answers 41

8. Adding signals

If you play three notes on a piano at once, you hear a **chord**: a single sound, richer than any one of the notes, made by adding the three together. Signals in a circuit combine in the same way.

Superposition

If $v_1(t)$ and $v_2(t)$ are voltages on the same wire, the wire carries $v_1(t) + v_2(t)$. The two signals don't interact with each other or multiply; they simply add, sample by sample, and this is called **superposition**. If you add two sines of different frequencies, the sum can look nothing like either of them, in the same way that a C-E-G chord doesn't sound like a C, an E or a G on its own. That one fact, that signals on a wire add, is enough to build almost any waveform you'll meet in this course.

Reading a sum by eye

When you add the three notes of a C-major chord (C4, E4 and G4: 261.6, 329.6 and 392.0 Hz) and plot the result, the sum no longer looks like three sines. Once you add more than one or two tones together, the shape of a plot against time stops telling you what went into it, so you can't read the notes back off the curve by eye. Movement III, starting with The FFT as an equalizer, gives you a tool that undoes the sum and recovers the parts.

You'll build this chord yourself in Lab 02, Part 1.

Why an engineer cares

Every wire on a chip is a point where signals add: what's on it is the signal you meant to put there, plus noise from the supply, plus whatever couples in from a neighbouring trace, plus anything else that touches it. The usual way to answer "What's actually on this node?" is to split the sum back into its parts, which is what Movement III teaches. Before that, the next page, Timbre and harmonics, uses superposition to build a square wave out of nothing but sines.

Next: Timbre and harmonics

9. Timbre and harmonics

Play A4 (440 Hz) on a violin and then on a trumpet, at the same loudness. The two notes have the same pitch and a tuner would read them as identical, but nobody would mistake one instrument for the other. What differs is their **timbre**: each instrument adds its own recipe of extra, quieter tones on top of the main note. Those extra tones are called **harmonics**, and a square wave is the cleanest example of what they do.

The square wave's recipe

A square wave of amplitude 1 and fundamental frequency f_0 can be written as an infinite chord:

$$v_{\text{square}}(t) = \frac{4}{\pi}\sin(2\pi f_0 t) + \frac{4}{3\pi}\sin(2\pi\, 3 f_0 t) + \frac{4}{5\pi}\sin(2\pi\, 5 f_0 t) + \cdots \tag{9.1}$$

Only the **odd harmonics** ($n = 1, 3, 5, \ldots$) appear, and their amplitudes fall off as $1/n$: the n-th one has amplitude $4/(n\pi)$.

The more terms you add, the closer the sum gets to a perfect square, but near each edge it overshoots and wiggles. This overshoot is called **Gibbs ringing**, and adding more harmonics won't make it go away, because it's a mathematical consequence of approximating a sharp edge with a finite number of sines. You'll see the same shape on an oscilloscope trace of a real digital clock.

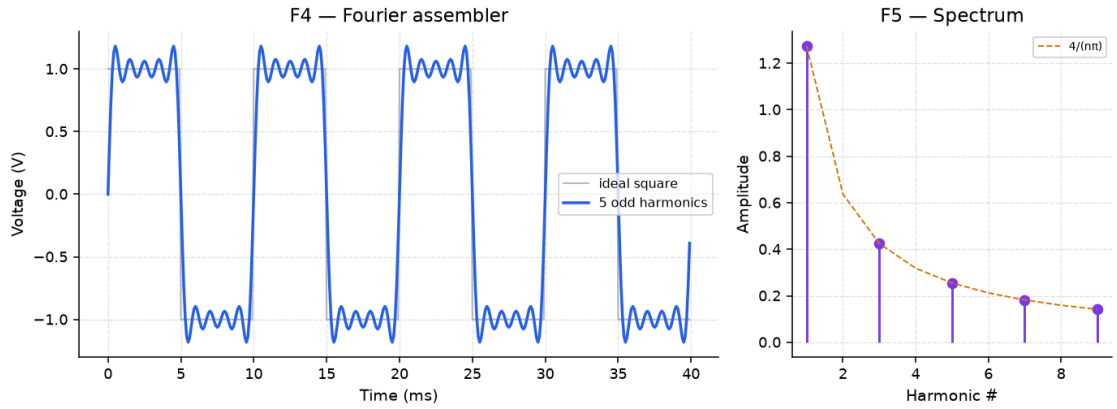


Figure 9.1: F4 / F5 — Harmonic builder

The frequency domain

Since any (nice) signal can be written as a sum of sines, the list of those sines' amplitudes and frequencies describes the signal completely. That list is the signal in the **frequency domain**, and for a musical note it's the instrument's recipe, the thing behind its timbre.

The natural way to draw the recipe is a **stem plot**, which has a stick at each harmonic whose height is that harmonic's amplitude. It's the same idea as the bar display on an equalizer, applied to a single note instead of a whole song. In the figure above, F5 (right) is the stem plot of the waveform in F4 (left).

In Lab 02, Part 2 you'll switch harmonics on and off in the F4/F5 explorer, and then add them up yourself in a script.

Why an engineer cares

Observation	Implication
A “1 GHz digital clock” needs analog bandwidth $\gg$ 1 GHz	Its sharp edges are made of the high harmonics, so the bandwidth has to reach them
Sharp edges ring	The ringing comes from a truncated harmonic series, or from real interconnect behaving the same way
Timbre is the harmonic recipe	Two circuits with the same fundamental can look very different on a spectrum

Next: Rhythm and duty cycle

10. Rhythm and duty cycle

A metronome that ticks 50/50, sounding for exactly as long as it stays silent, has a different rhythm from one that snaps briefly and then rests. A pulse train works the same way: it's a square wave that doesn't have to be on for half of each cycle and off for the other half. The fraction of each cycle that it spends on is called its **duty cycle**, and when you change the duty cycle, you change which harmonics show up in the wave's recipe.

Duty cycle and the recipe

At exactly 50% duty, the square-wave recipe from Timbre and harmonics holds, so only the odd harmonics are present. If you move the duty cycle away from 50% in either direction, **even** harmonics ($n = 2, 4, 6, \ldots$) come back into the spectrum, though not always every one of them: in the 25% and 75% panels of F6 below, harmonics 4 and 8 are still missing.

This is one of the reasons clock specifications care about duty cycle. The frequencies in a clock's spectrum drive the electromagnetic interference (**EMI**) it produces, and a clock with a clean, symmetric duty cycle radiates a cleaner spectrum than a lopsided one does.

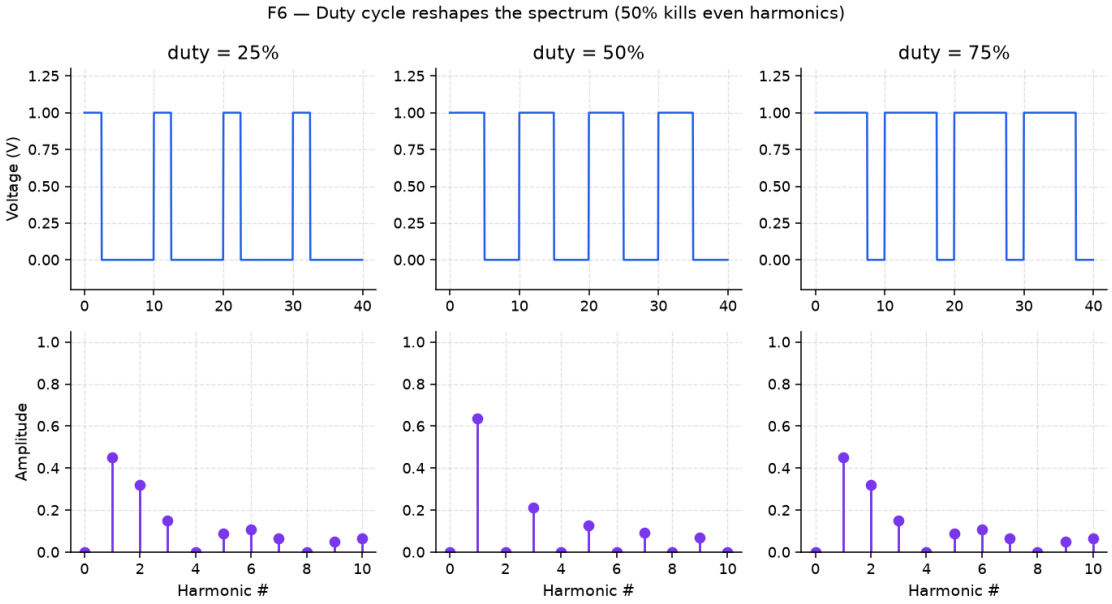


Figure 10.1: F6 — Duty cycle and spectrum

You'll sweep the duty cycle on the F6 explorer and plot the spectra of three pulse trains yourself in Lab 02, Part 3.

Why an engineer cares

Observation	Implication
A 50% duty cycle removes the even harmonics	The spectrum is cleaner, so there's less EMI
A duty cycle away from 50%	Even harmonics come back and radiate at their own frequencies

Observation	Implication
Clock spec sheets list a duty-cycle tolerance	The tolerance is there to keep the clock's spectrum clean

This page ends Movement II, and next comes Lab 02: Harmonic builder. After the lab, Movement III starts with The FFT as an equalizer, which gives you a tool that reads off the recipe of any signal automatically, so you no longer have to build it up by hand.

11. Lab 02: Harmonic builder

You'll add sines together into a chord, build a square wave out of odd harmonics while its spectrum updates beside it, and watch the duty cycle of a pulse train switch even harmonics on and off. Before the scripts run, you'll work out the harmonic amplitudes and frequencies they should show.

Lab package: `labs/lab-02-harmonic-builder/`

Before you start

- Finish Lab 01.
- Read Movement II: Adding signals, Timbre and harmonics and Rhythm and duty cycle.
- In a terminal in the noVNC desktop, go to the lab folder:

```
. /foss/designs/common/.designinit
mod ad101
cd labs/lab-02-harmonic-builder
```

Time: about 1 hour.

As in Lab 01, Part 1 is a snippet on its own, and the other parts each pair an explorer (`python3 src/explore.py --figure <id>`) with a snippet. You save each snippet as a new `.py` file in this folder and run it with `python3 <file>.py`.

Part 1: Build a chord

What does the sum of three notes look like in time?

1. Save this as `chord.py` and run it. It adds the three notes of a C-major triad, sample by sample:

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

fs = 10_000
t = np.arange(0, 0.02, 1 / fs)

# a C-major triad: C4, E4, G4 (Hz)
c = np.sin(2 * np.pi * 261.6 * t)
e = np.sin(2 * np.pi * 329.6 * t)
g = np.sin(2 * np.pi * 392.0 * t)

chord = c + e + g

fig, axes = plt.subplots(2, 1, sharex=True, figsize=(7, 5))
axes[0].plot(t * 1e3, c, label="C4", alpha=0.7)
axes[0].plot(t * 1e3, e, label="E4", alpha=0.7)
axes[0].plot(t * 1e3, g, label="G4", alpha=0.7)
axes[0].legend(fontsize=8)
axes[0].set_ylabel("three notes")

axes[1].plot(t * 1e3, chord, color="black")
axes[1].set_ylabel("sum (the chord)")
axes[1].set_xlabel("Time (ms)")
plt.tight_layout()
plt.show()
```

You should see the three notes in the top panel and their sum in black in the bottom panel. From the bottom trace alone you couldn't tell by eye that it's made of three sines, and Movement III gives you a tool that can.

Part 2: Build a square wave from sines (F4/F5)

Which sines, at which amplitudes, add up to a square wave?

1. Open the Fourier assembler, which shows F4 (the waveform) and F5 (its spectrum) in the same window:

```
python3 src/explore.py --figure f4
```

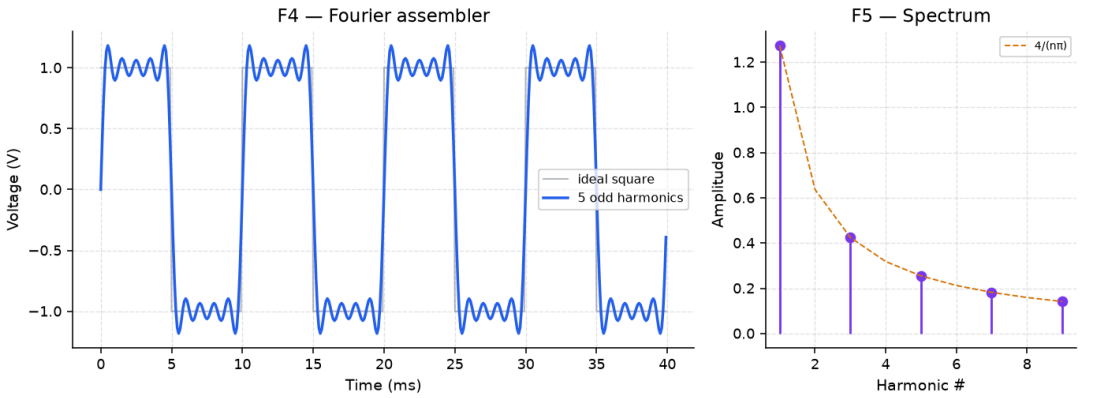


Figure 11.1: F4/F5

2. All five harmonics (n=1, 3, 5, 7, 9) start ticked, with their amplitudes set to $4/(n\pi)$ rounded to the sliders' 0.02 steps, which is why A1 reads 1.28. Untick 3, 5, 7 and 9 so that only n=1 is left.

   You should see a plain sine in F4 and a single stem in F5.

3. Next you'll tick 3, 5, 7 and 9 back on, one at a time. Before you do, predict whether the overshoot at the square's edges will shrink away as you add harmonics, and write your answer down. Then tick them on.

   You should see the sum walk toward the gray ideal square, with an overshoot near each edge that doesn't go away. The F5 stems update in step with the waveform and sit on the dashed $4/(n\pi)$ curve, the square-wave recipe from Timbre and harmonics.

   Why doesn't the overshoot go away as you add harmonics?

   Answer: A1, in Answers at the end.

4. Drag the amplitude sliders (A1 to A9) toward and away from $4/(n\pi)$.

   You should see the sum stop looking like a square as the stems move off the dashed curve, and look like one again as they move back onto it. Close the window.

5. Save this as `square_from_sines.py`, but don't run it yet. It prints the amplitude it uses for each harmonic:

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

fs = 10_000
t = np.arange(0, 0.02, 1 / fs)
f0 = 200.0                          # fundamental frequency, Hz

harmonics = [1, 3, 5, 7, 9]         # odd harmonics only
built = np.zeros_like(t)
for n in harmonics:
```

```
    amplitude = 4 / (n * np.pi)   # the square-wave recipe
    built += amplitude * np.sin(2 * np.pi * n * f0 * t)
    print(f"n = {n}   amplitude = {amplitude:.3f}")

ideal_square = np.sign(np.sin(2 * np.pi * f0 * t))

plt.plot(t * 1e3, ideal_square, color="gray", linestyle="--", label="ideal square")
plt.plot(t * 1e3, built, label=f"sum of {len(harmonics)} harmonics")
plt.xlabel("Time (ms)")
plt.ylabel("Voltage (V)")
plt.legend()
plt.show()
```

The `4 / (n * np.pi)` line is the recipe from the square-wave formula in Timbre and harmonics. Before you run it, work out $4/(n\pi)$ for n = 1, 3, 5, 7 and 9 to three decimal places, and write the five numbers down. Then run it.

You should see:

```
n = 1   amplitude = 1.273
n = 3   amplitude = 0.424
n = 5   amplitude = 0.255
n = 7   amplitude = 0.182
n = 9   amplitude = 0.141
```

These are the values the F4 sliders showed, before they were rounded to 0.02. The plot shows the sum of five harmonics close to the dashed square, with ringing at the edges.

6. Change the list to `harmonics = [1]` and re-run. Then add `3`, then `5`, `7` and `9`, re-running each time.

   You should see the sum start as a plain sine and walk toward the dashed square with each sine you add, and one more `amplitude` line printed each time. The overshoot right at the edges appears and stays, as it did on F4. That's the Gibbs ringing from step 3, not a bug in your code.

Part 3: Duty cycle and even harmonics (F6)

Which harmonics does a pulse train's duty cycle switch on and off?

1. Open the duty-cycle explorer:

```
python3 src/explore.py --figure f6
```

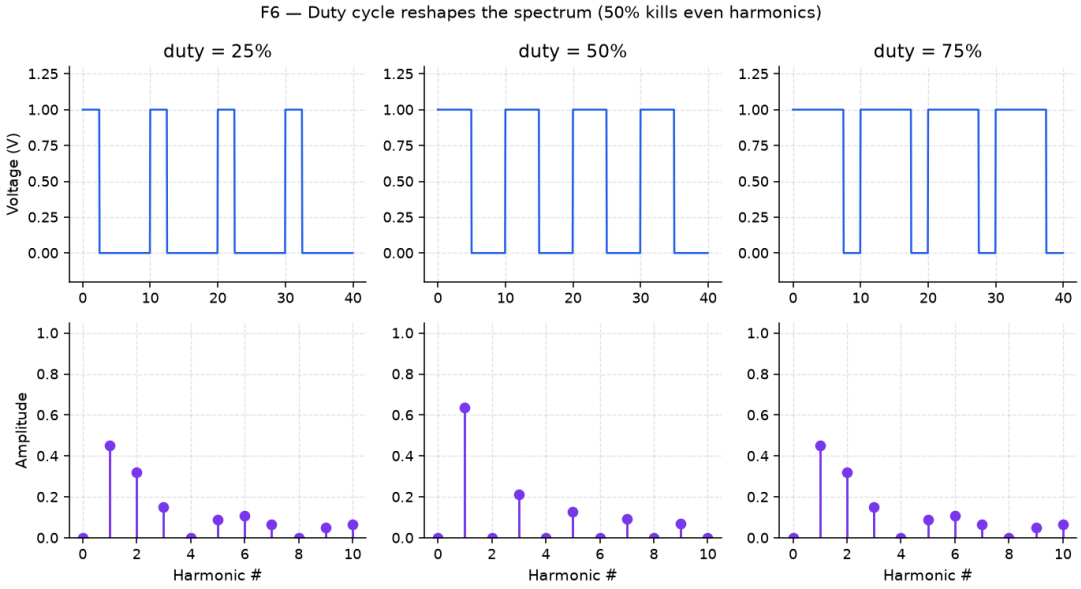


Figure 11.2: F6

It starts at 50% duty, and the box under the plots reads `duty = 50% → even harmonics GONE (50% duty)`.

2. Sweep Duty from 25% to 50% to 75% and watch the even-numbered stems in the spectrum.

   You should see the even stems vanish at exactly 50% and come back on either side of it (harmonic 2 is the easiest to spot), with the box switching to `even harmonics PRESENT`. Close the window.

3. Save this as `duty_sweep.py`, but don't run it yet. It uses `np.fft.rfft`, the "count the harmonics for me" machine you'll meet properly in The FFT as an equalizer. For now, treat it as a black box:

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

fs = 10_000
t = np.arange(0, 0.05, 1 / fs)
f0 = 200.0

fig, axes = plt.subplots(1, 3, figsize=(10, 3), sharey=True)
for ax, duty in zip(axes, [0.25, 0.50, 0.75]):
    phase = (f0 * t) % 1.0
    pulse = np.where(phase < duty, 1.0, 0.0)
    spectrum = np.abs(np.fft.rfft(pulse - pulse.mean()))
    freqs = np.fft.rfftfreq(len(pulse), d=1 / fs)
    ax.stem(freqs[:100], spectrum[:100])
    ax.set_title(f"duty = {duty:.0%}")
    ax.set_xlabel("Frequency (Hz)")
plt.tight_layout()
plt.show()
```

   It plots the spectra of 200 Hz pulse trains at 25%, 50% and 75% duty. Before you run it, write down the frequencies between 0 and 2000 Hz where you expect stems in the 50% panel. Then run it.

You should see three stem plots from 0 to 2000 Hz. In the middle panel (50%) the only stems are at odd multiples of 200 Hz: 200, 600, 1000, 1400 and 1800 Hz. The 25% and 75% panels also have stems at even multiples, such as 400 Hz.

Finishing

You're done when the five amplitudes you worked out match what `square_from_sines.py` prints (Part 2, step 5), and the stems in the 50% panel of `duty_sweep.py` sit at the frequencies you wrote down (Part 3, step 3).

Questions

1. Why doesn't adding more harmonics remove the overshoot at a square's edges? (Answered in Part 2, step 3.)

Submit

Nothing to submit.

Next on the Course path: read Movement III, The FFT as an equalizer, Finding a quiet note, Distortion and THD and Sampling fast enough, then do Lab 03: Spectrum detective.

Answers

A1. Part 2: Build a square wave from sines (F4/F5)

The overshoot is called Gibbs ringing. It comes from approximating a sharp edge with a finite number of sines, so it stays however many harmonics you add. A real digital clock on an oscilloscope shows the same shape.

III Movement III: The mixing board

12 The FFT as an equalizer 45
12 The FFT as a machine 45
12 Reading a spectrum 46
12 Decibels 47

13 Finding a quiet note 49
13 A tone buried in noise 49
13 Spectral leakage (stretch) 49
13 Why an engineer cares 50

14 Distortion and THD 51
14 New frequencies from one 51
14 Why an engineer cares 51

15 Sampling fast enough 53
15 Sampling and aliasing 53
15 Worked example: a 900 Hz tone 53
15 Why an engineer cares 53

16 Lab 03: Spectrum detective 55
16 Before you start 55
16 Part 1: Your first FFT (F7) 55
16 Part 2: Find the hidden tone (F8) 57
16 Part 3: Distortion and THD (F9) 58
16 Part 4: Sampling and aliasing (F11) 60
16 Part 5 (optional): Spectral leakage (F10) 61
16 Finishing 61
16 Answers 62

12. The FFT as an equalizer

In Movement II you built waveforms out of sines by hand, choosing which harmonics to add and how big each one should be. The **Fast Fourier Transform (FFT)** runs that process backwards: you give it a time waveform, and it gives you back the recipe, which lists the frequencies in the waveform and how strong each one is. It's also the math behind every equalizer display you've seen bouncing along to a song.

The FFT as a machine

Treat the FFT as a black box for now:

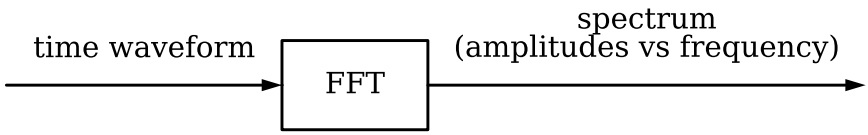


Figure 12.1: A time waveform goes in, and its spectrum comes out.

You don't need to know how the FFT is derived to use it: `numpy.fft.rfft` does the whole calculation in one line, and the Lab 03 explorers run it for you and plot the result. F7, below, shows five test signals, each with its time plot on the left and its spectrum on the right.

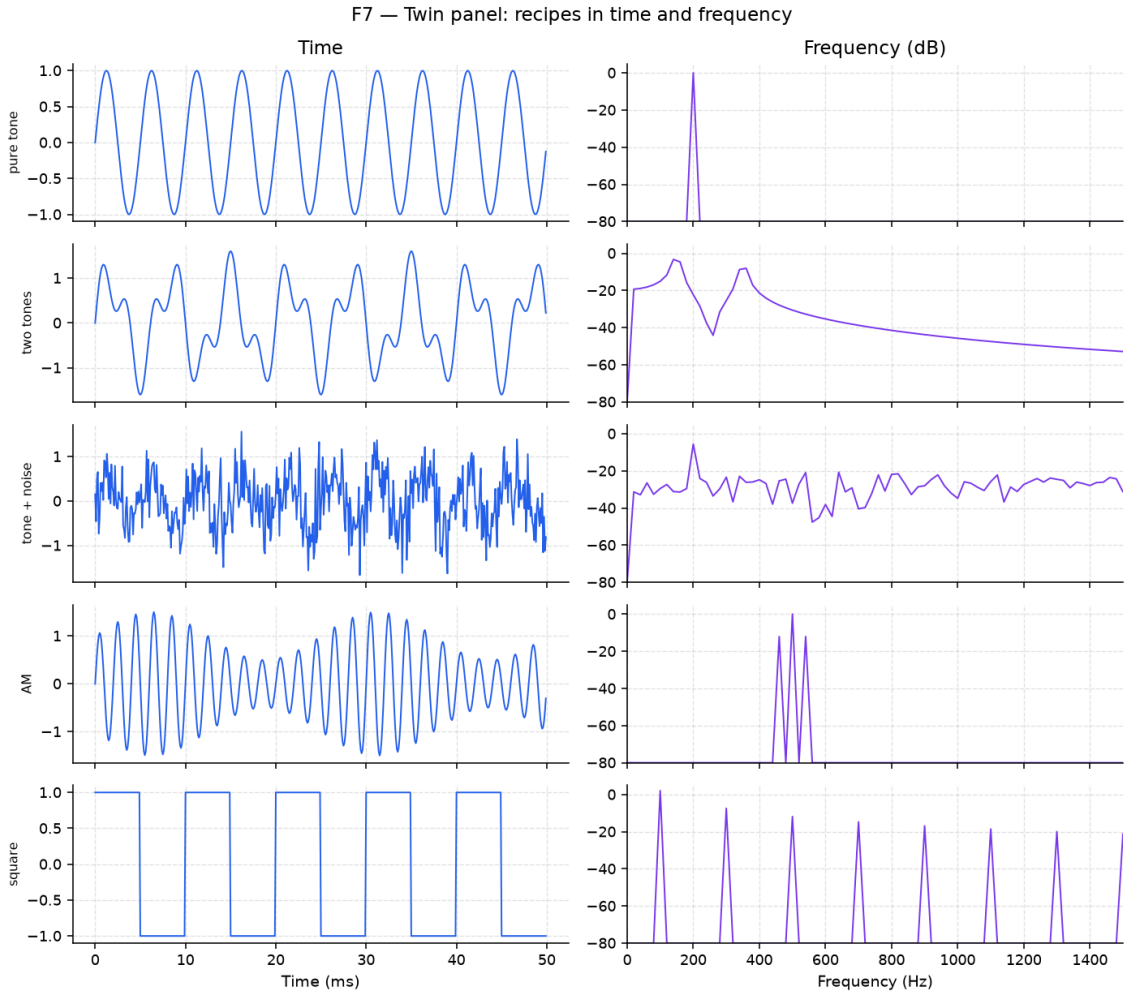


Figure 12.2: F7 — Twin panel

Reading a spectrum

Look for	Meaning
Where peaks sit	The frequencies present in the signal, which for a chord are its notes
How tall peaks are	The relative strength of each tone, which is how loud that note is in the mix
The noise floor	The level of the hiss, which sets the smallest feature you can still see
Harmonic combs	Distortion, clocks or square-ish shapes; in music, a comb is what you hear as the timbre from Movement II

Take a two-note chord, A4 (440 Hz) plus a quieter E5 (659 Hz). Its time plot is two wiggles on top of each other, and by eye it's hard to tell that there are exactly two notes. Its spectrum shows two clear peaks, one at 440 Hz and one at 659 Hz. That difference between the two views is the reason this movement exists, because identifying a signal by the shape of its spectrum is a core debugging skill.

In Lab 03, Part 1, you'll compute this chord's spectrum yourself and then flip between test signals on the F7 explorer.

Decibels

Human ears and radio-frequency (RF) engineers both care about ratios rather than differences. Going from a whisper to a normal voice feels like as big a jump as going from a normal voice to a shout, even though the second jump is a much bigger change in raw voltage. The **decibel** describes a ratio with a logarithm, so that equal ratios become equal steps:

$$\mathrm{dB} = 20 \log_{10}\!\left(\frac{A}{A_{\mathrm{ref}}}\right) \tag{12.1}$$

These few conversions are worth memorizing:

Linear ratio	dB
$\times 2$	$\approx +6$ dB
$\times 10$	$+20$ dB
$\times 0.5$	≈ -6 dB
$\times 0.1$	-20 dB
$1/\sqrt{2} \approx 0.707$	-3 dB

On a dB axis, a loud tone and a quiet tone can share one plot without the quiet one disappearing into the baseline, which is why the meters on a studio equalizer are marked in dB rather than volts.

Next: Finding a quiet note

13. Finding a quiet note

Stand in a loud room and try to pick out one person humming a note under the noise, and your ear will struggle with it. A spectrum analyzer for sound would have no such trouble, because the hum would show up as a single sharp spike sitting above the noise floor. Picking out a quiet tone like this is one of the main reasons the course teaches the frequency domain.

A tone buried in noise

A small sine wave under loud noise is invisible on a scope trace, but on a spectrum it shows up as an obvious spike.

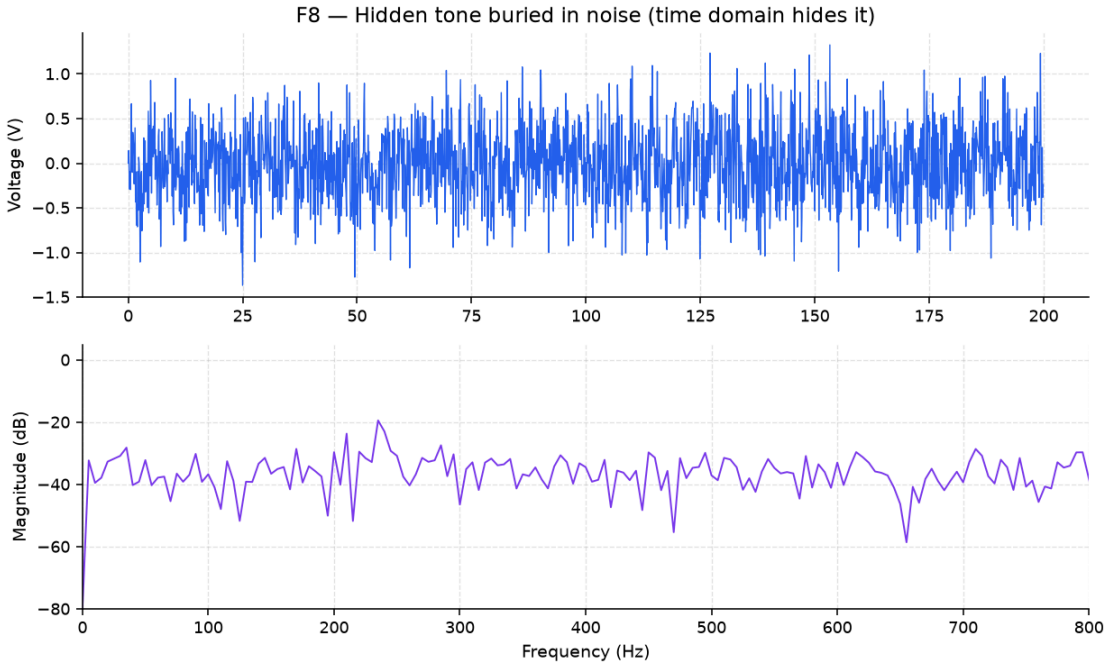


Figure 13.1: F8 — Hidden tone

Whenever you're hunting for a small interferer on a node, you use exactly this skill: you stop staring at the time trace and look at the spectrum instead. Once the spike is there, you can read its frequency straight off the data rather than guessing it from a picture.

In Lab 03, Part 2 you'll hunt for a hidden tone on the F8 explorer, confirm your answer with a checker script, and then repeat the hunt in a script of your own.

Spectral leakage (stretch)

Real FFTs don't draw perfect sticks, though. If the measurement window doesn't hold a whole number of cycles, the peak smears sideways into the neighbouring frequency bins (an FFT reports the spectrum only at a set of evenly spaced frequencies, and each of those is called a bin), and this smearing is called **leakage**. A **window function**, such as a Hann window, reduces the smear by tapering the start and end of the recorded segment toward zero.

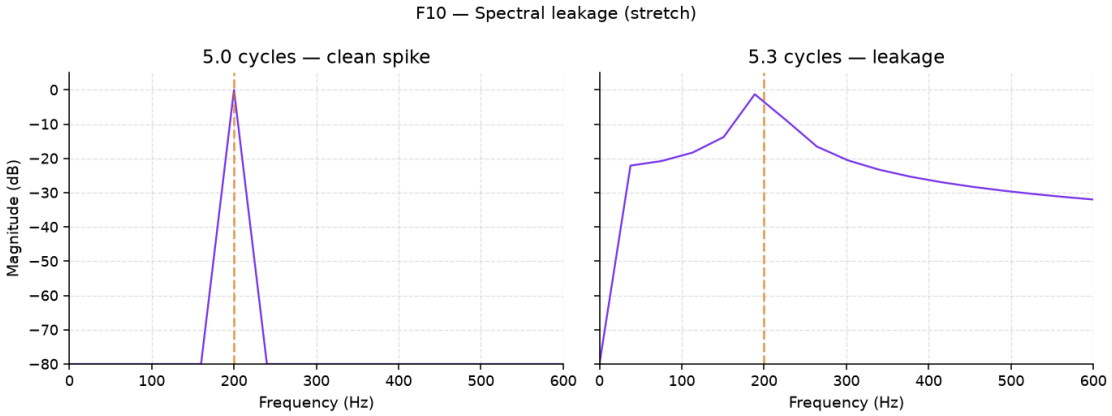


Figure 13.2: F10 — Leakage

Leakage is why measured spectra look soft and smeared rather than like clean textbook sticks. You've already seen it once: in F7 on the last page, The FFT as an equalizer, the two-tone spectrum is smeared while the pure tone is a clean stick. It's worth knowing about before it confuses you on a real bench measurement. Lab 03, Part 5 (optional), lets you smear a peak on the F10 explorer and then reduce the smear with a Hann window.

Why an engineer cares

Reading the spectrum is how you track down 60 Hz hum, a clock coupling into a sensitive analog node, or any other small interferer that you can't spot by eye.

Next: Distortion and THD

14. Distortion and THD

Turn a guitar amp's gain up past the clean setting and the sound gets crunchy, with harsh new overtones that were never in the original note. The input's frequency hasn't changed; the circuit has clipped the wave, and the clipping created the new frequencies. That's **distortion**, and only a nonlinear system can produce it, because a truly linear circuit can only rescale and delay a sine and can never add new tones to it.

New frequencies from one

Push a clean sine through a soft nonlinearity such as tanh (a smooth version of clipping), and harmonics that weren't in the input appear in the spectrum. The harder you drive it, the more it clips, and the bigger those harmonics get.

Total harmonic distortion (THD) sums up how much energy has moved into those extra harmonics, relative to the fundamental. A_1 is the amplitude of the fundamental and $A_2, A_3, \ldots$ are the amplitudes of the harmonics:

$$\mathrm{THD} = \frac{\sqrt{A_2^2 + A_3^2 + \cdots}}{A_1} \tag{14.1}$$

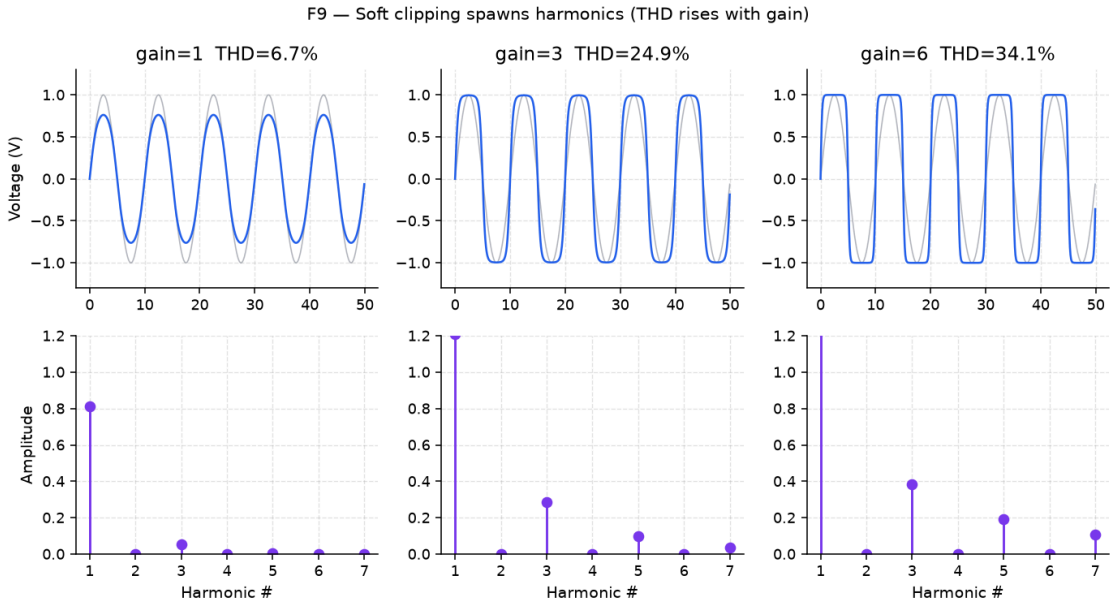


Figure 14.1: F9 — THD

THD turns the question "how distorted does it sound?" into a single number you can read off, so you don't need a listening test to answer it. In Lab 03, Part 3, you'll raise the gain on the F9 explorer and watch THD climb, and then clip a sine in your own script and measure its THD.

Why an engineer cares

Every amplifier datasheet quotes a linearity or THD number, so this is a figure you'll keep meeting. AD103 comes back to distortion with MOSFET curves and AD201 with amplifiers, and both treat it more precisely than this page does.

Next: Sampling fast enough

15. Sampling fast enough

You've probably seen a car's wheels seem to spin backwards in a video while the car drives forward. The camera captures only a handful of frames per second, and between frames the spokes move almost, but not quite, a full turn, so your brain reconstructs a slow, wrong rotation. Nothing is wrong with the camera; it just isn't sampling often enough to keep up with the real motion, and an ADC measuring a voltage can fail in exactly the same way.

Sampling and aliasing

If you measure a tone too slowly, a fast sine passes itself off as a slow one, and the fake low-frequency signal you end up seeing is called an **alias**. The rule that prevents it is the Nyquist criterion, which says the sample rate has to be more than twice the highest frequency in the signal:

$$f_{\text{sample}} > 2\, f_{\text{max}} \qquad \text{(Nyquist)} \tag{15.1}$$

Two related numbers both get called "Nyquist", so it's worth keeping them apart. The Nyquist rate is $2f_{\text{max}}$, the rate your sample rate has to beat. Half the sample rate, $f_s/2$, is the highest frequency that a given sample rate can represent, and that's the one the F11 explorer labels Nyquist.

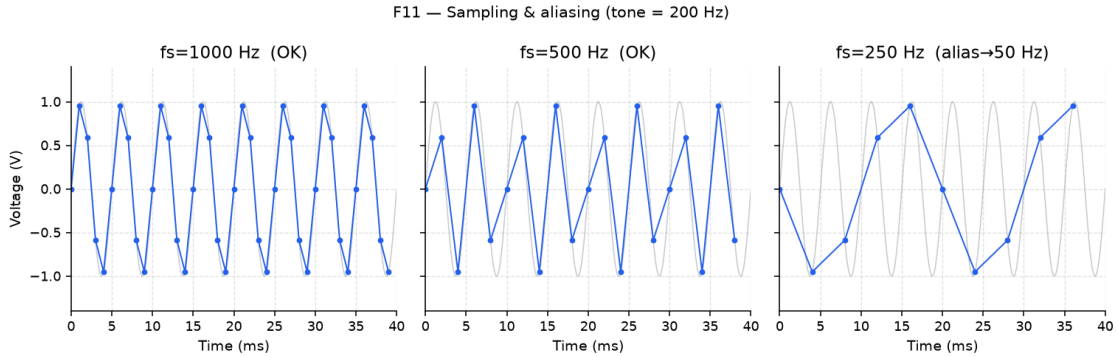


Figure 15.1: F11 — Aliasing

Worked example: a 900 Hz tone

Sample a 900 Hz tone at 5000 samples per second, well above the Nyquist rate, and the sample points trace out a recognizable 900 Hz wiggle. Sample the same tone at 1000 samples per second, below the $2 \times 900 = 1800$ Hz Nyquist rate, and the points join up into a slow, wrong-looking wave, just like the backwards-spinning wheel. The tone is the same in both cases, and the only thing that changed is how often you looked at it.

In Lab 03, Part 4 you'll drop the sample rate on the F11 explorer until the tone aliases, and then reproduce this 900 Hz example in a script of your own.

Why an engineer cares

The sample rate you choose for an ADC decides whether you'll ever see this failure, which is why sample rates are chosen carefully. It's also the starting point for AD202 (mixed signal).

This page ends Movement III, and next comes Lab 03: Spectrum detective. After the lab, Movement IV starts with Filters are tone knobs, where you ask what a circuit does to a signal's recipe instead of only reading the recipe off.

16. Lab 03: Spectrum detective

In this lab you'll compute your first FFT and match test signals to their spectra. Then you'll find a tone hidden under noise and confirm it with a checker, watch clipping create harmonics and read off the THD, and make a tone lie about its frequency by sampling it too slowly. Along the way you'll predict three numbers before a script prints them: a dB gap, a THD and an alias frequency.

Lab package: `labs/lab-03-spectrum-detective/`

Before you start

- Finish Lab 02.
- Read Movement III: The FFT as an equalizer, Finding a quiet note, Distortion and THD and Sampling fast enough.
- Know the dB table in Reading plots.
- In a terminal in the noVNC desktop, go to the lab folder:

```
. /foss/designs/common/.designinit
mod ad101
cd labs/lab-03-spectrum-detective
```

Time: about 1.5 hours, not counting the optional Part 5.

As before, most parts pair an explorer (`python3 src/explore.py --figure <id>`) with a short snippet that you save as a new `.py` file in this folder and run with `python3 <file>.py`.

Part 1: Your first FFT (F7)

How does the same signal look in time and in frequency?

1. Save this as `first_fft.py`, but don't run it yet. It builds a two-note chord, A4 and E5, and plots it in time and in frequency:

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

fs = 10_000
t = np.arange(0, 0.2, 1 / fs)          # a longer window gives a sharper spectrum

# a two-note chord: A4 and E5
v = np.sin(2 * np.pi * 440 * t) + 0.5 * np.sin(2 * np.pi * 659 * t)

spectrum = np.abs(np.fft.rfft(v - v.mean()))
freqs = np.fft.rfftfreq(len(v), d=1 / fs)
spectrum_db = 20 * np.log10(np.maximum(spectrum, 1e-6))

fig, axes = plt.subplots(2, 1, figsize=(7, 6))
axes[0].plot(t * 1e3, v)
axes[0].set_xlabel("Time (ms)")
axes[0].set_ylabel("Voltage (V)")
axes[0].set_title("time domain")

axes[1].plot(freqs, spectrum_db)
axes[1].set_xlim(0, 1000)
```

```
axes[1].set_xlabel("Frequency (Hz)")
axes[1].set_ylabel("Magnitude (dB)")
axes[1].set_title("frequency domain")
plt.tight_layout()
plt.show()
```

In the script, `v - v.mean()` removes any DC offset before the FFT, and `np.maximum(..., 1e-6)` makes sure `log10` is never handed a zero.

E5 has half A4's amplitude. Before you run it, use the dB table in Reading plots to predict how many dB below the A4 peak the E5 peak will sit, and write it down. Then run it.

You should see a tangle of two overlapping wiggles in the top plot and two clear peaks in the bottom one: A4 at 440 Hz and E5 near 659 Hz, which sit at 60.0 dB and 53.4 dB in the course image. The dB values are raw FFT magnitudes, so the peaks sit far above 0 dB, and only the gap between them means anything. Close the window.

How big is the gap, and why isn't it exactly the 6 dB the table gives?

Answer: A1, in Answers at the end.

2. Open the twin-panel explorer:

```
python3 src/explore.py --figure f7
```

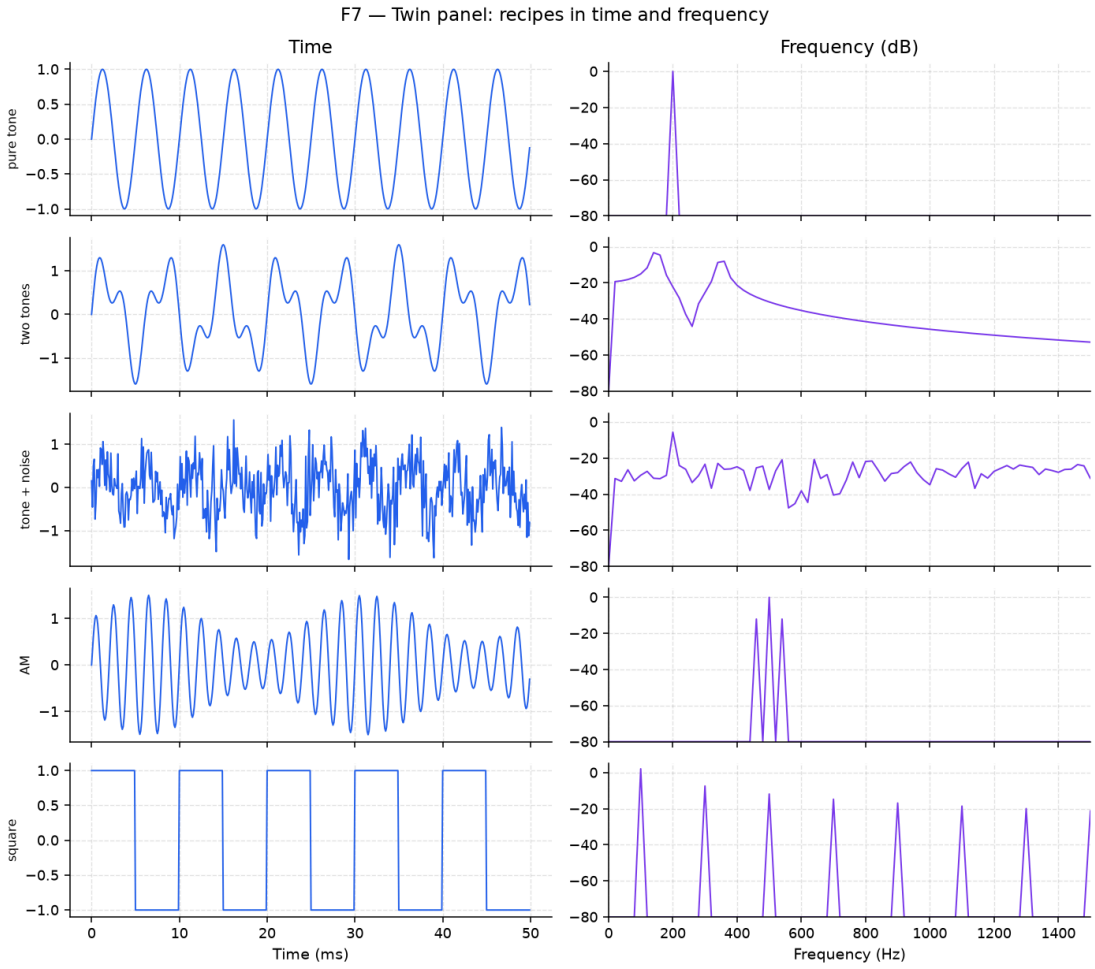


Figure 16.1: F7

3. Click through the signals on the left: pure tone, two tones, tone + noise, AM and square. AM (amplitude modulation) is a 500 Hz tone, called the carrier, whose loudness wobbles up and down at 40 Hz. Then switch the “dB scale” and “log freq” boxes on and off.

   You should see one spike for the pure tone, two spikes for two tones, a carrier with a sideband on each side for AM, and a comb of odd harmonics for the square. With dB on, you can also see small peaks that the linear axis hides. Close the window.

Part 2: Find the hidden tone (F8)

Can you find a tone you can’t see in the time plot?

1. Open the hidden-tone hunt:

```
python3 src/explore.py --figure f8
```

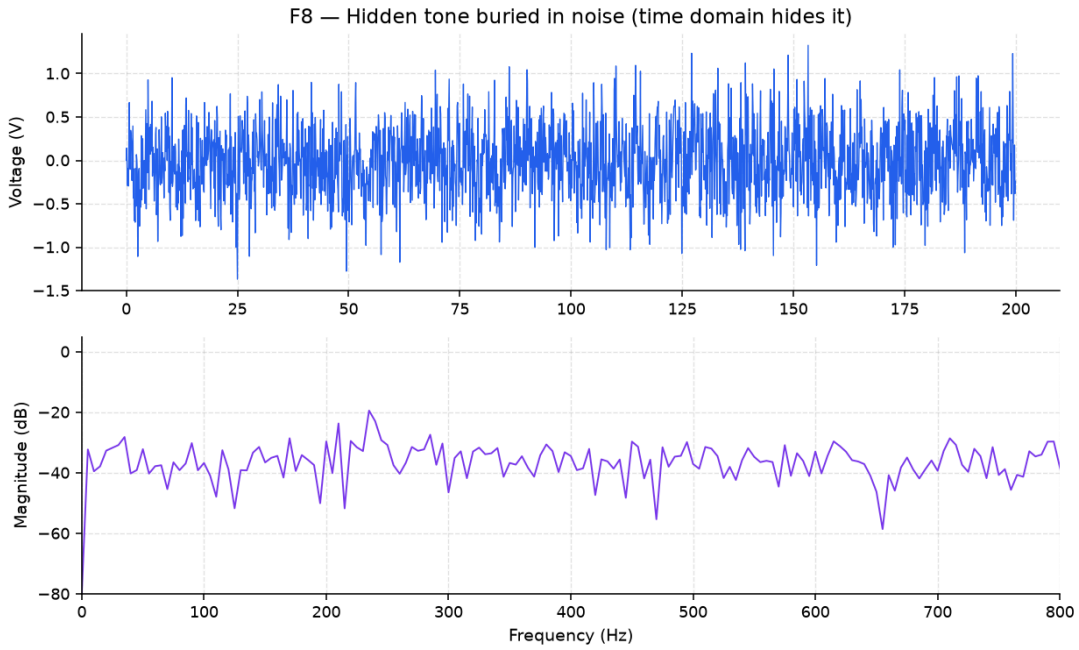


Figure 16.2: F8

2. Raise “Noise amp” until the time plot (top) is garbage. Then look for a narrow peak above the noise floor in the spectrum (bottom), and write down its frequency. To read it precisely, hover the mouse over the peak, and the window’s toolbar will show the cursor’s x and y coordinates. If you lower the noise again, both views become clearer.
3. Close the window and check your answer, putting your frequency in place of `<Hz>`:

```
python3 src/check.py --guess <Hz>
```

   You should see a line starting `Correct! Hidden tone is`. If you see `Not quite` instead, the message tells you how far off you were. The checker accepts anything within 5 Hz of the answer, so if you’re further off than that, read the spike again more carefully and try another guess.
4. Save this as `hidden_tone.py` and run it. It hides a tone at the same frequency as F8′s, but quieter and under different noise, and then reads the peak’s frequency off the data with `np.argmax` instead of by eye. The code gives away the tone’s frequency, so do steps 1–3 before you read it:

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

```
fs = 10_000
t = np.arange(0, 0.5, 1 / fs)

rng = np.random.default_rng(0)
hidden_freq = 237.0                              # you're not allowed to peek at this
in real life —
                                                 # here it's revealed so you can check
your answer
noisy = 0.05 * np.sin(2 * np.pi * hidden_freq * t) + 0.3 * rng.standard_normal(len(t))

spectrum = np.abs(np.fft.rfft(noisy - noisy.mean()))
freqs = np.fft.rfftfreq(len(noisy), d=1 / fs)

plt.subplot(2, 1, 1)
plt.plot(t[:200] * 1e3, noisy[:200])
plt.title("time domain — can you see the tone?")
plt.xlabel("Time (ms)")

plt.subplot(2, 1, 2)
plt.plot(freqs, spectrum)
plt.xlim(0, 500)
plt.title("frequency domain — now can you?")
plt.xlabel("Frequency (Hz)")
plt.tight_layout()
plt.show()

guess = freqs[np.argmax(spectrum)]
print(f"Loudest peak sits at ≈ {guess:.0f} Hz")
```

You should see pure noise in the time plot and one obvious spike in the spectrum. After you close the window, the script prints `Loudest peak sits at ≈ 236 Hz`, within 5 Hz of the hidden 237 Hz.

Part 3: Distortion and THD (F9)

How much new energy does clipping put into the harmonics?

1. Open the distortion explorer:

```
python3 src/explore.py --figure f9
```

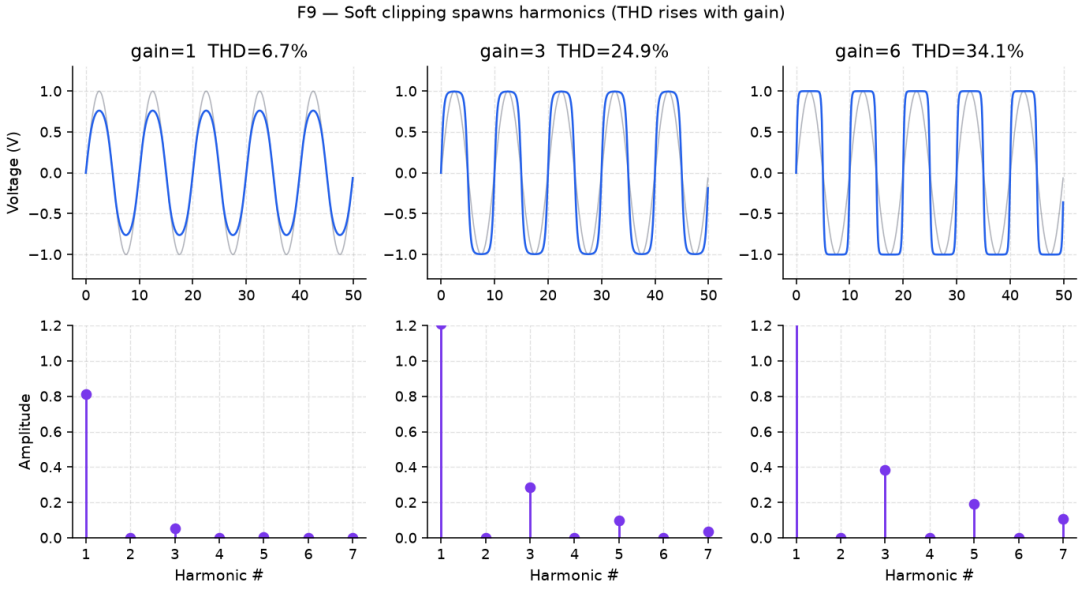


Figure 16.3: F9

2. Raise “Gain (into tanh)” and watch the waveform, the harmonic stems and the THD readout. tanh is the smooth clipping curve from Distortion and THD, and more gain drives the sine further into it.

   You should see the soft clipping flatten the peaks of the sine, the odd harmonics grow, and the THD climb from about 2% at the lowest gain (0.5) to about 35% at gain 6. Set the gain to 1 and write down the THD it reads. Close the window.

3. Save this as `clip_thd.py`, but don’t run it yet. It clips a sine at three gains, 1, 3 and 8, and measures the THD from the spectrum. It only prints its results, so no plot window opens:

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

fs = 10_000
t = np.arange(0, 0.1, 1 / fs)
f0 = 200.0
clean = np.sin(2 * np.pi * f0 * t)

for gain in (1.0, 3.0, 8.0):
    distorted = np.tanh(gain * clean)               # soft clip — bigger gain =
harder clip

    spectrum = np.abs(np.fft.rfft(distorted - distorted.mean()))
    freqs = np.fft.rfftfreq(len(distorted), d=1 / fs)

    def amp_at(f_hz):
        return spectrum[np.argmin(np.abs(freqs - f_hz))]

    a1 = amp_at(f0)
    harmonics_power = sum(amp_at(f0 * n) ** 2 for n in (2, 3, 4, 5))
    thd = (harmonics_power ** 0.5) / a1

    print(f"gain={gain:>4.1f}   THD = {thd*100:5.1f}%")
```

Before you run it, use your F9 reading to predict the THD it will print at gain 1, and decide whether gain 3 and gain 8 will print more or less. Write your predictions down, then run it.

You should see:

```
gain= 1.0   THD =   6.7%
gain= 3.0   THD =  24.9%
gain= 8.0   THD =  36.0%
```

F9 at gain 1 reads `THD = 6.7%`, the same as the script. The two don't measure quite the same thing, though: the script sums harmonics 2 to 5 while F9 sums harmonics 2 to 7, so they can differ slightly at high gain.

Part 4: Sampling and aliasing (F11)

How slowly can you sample a tone before it shows up at the wrong frequency, and which frequency does it show up at?

1. Open the aliasing explorer:

```
python3 src/explore.py --figure f11
```

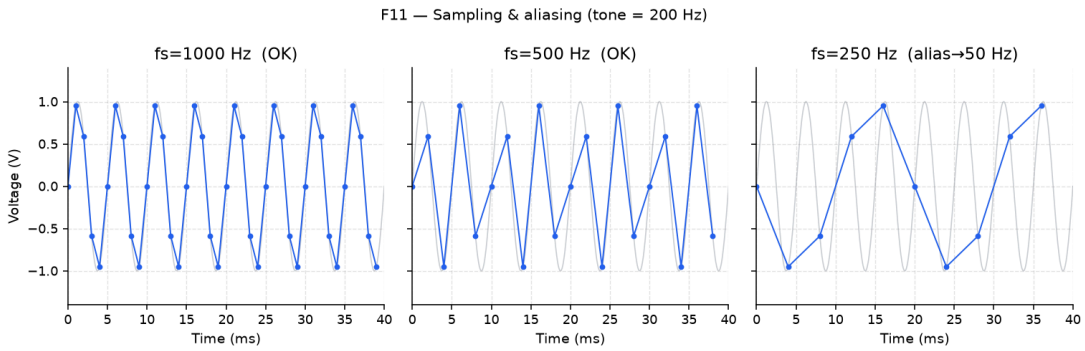


Figure 16.4: F11

The tone is 200 Hz and the sample rate starts at 1000 Hz, so the box at the bottom ends in `OK (fs > 2f)`. The `Nyquist` value in that box is half the sample rate.

2. Before you move anything, work out the sample rate below which a 200 Hz tone aliases, and write it down.
3. Drag "Sample rate (Hz)" down slowly until the box changes, then set it to 250 Hz.

   You should see the box switch to `ALIASED → appears as … Hz` once the sample rate drops below 400 Hz, a bogus slow sine appear over the samples, and the spectrum peak fold below Nyquist. At 250 Hz the box reads `ALIASED → appears as 50 Hz`. Close the window.

4. Save this as `aliasing.py`, but don't run it yet. It samples a 900 Hz tone twice, once above Nyquist and once below. In this script "Nyquist" means the Nyquist rate, $2f$ = 1800 Hz, rather than F11's `Nyquist` value of $f_s/2$, and Sampling fast enough explains the difference between the two:

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

true_freq = 900.0                # Hz — the real tone
duration = 0.02

for fs in (5000.0, 1000.0):      # first well above Nyquist, then below it
    t = np.arange(0, duration, 1 / fs)
    v = np.sin(2 * np.pi * true_freq * t)

    nyquist_ok = fs > 2 * true_freq
    label = "OK — above Nyquist" if nyquist_ok else "ALIASED — below Nyquist"
```

```
    plt.plot(t * 1e3, v, marker="o", label=f"fs={fs:.0f} Hz  ({label})")

plt.xlabel("Time (ms)")
plt.ylabel("Voltage (V)")
plt.legend(fontsize=8)
plt.title(f"true tone = {true_freq:.0f} Hz")
plt.show()
```

In step 3, at 250 Hz, F11 showed the 200 Hz tone as 50 Hz, which is 250 − 200. That's a general rule: a tone that lies between half the sample rate and the sample rate shows up at the sample rate minus the tone's frequency. Use it to predict the frequency the `fs=1000` points will appear to have, and write it down. Then run the script.

You should see the `fs=5000` points trace a recognizable 900 Hz wiggle, while the `fs=1000` points join into a slow, wrong-looking wave with a period of about 10 ms (100 Hz, which is 1000 − 900), like the backwards-spinning wheel in Sampling fast enough.

Part 5 (optional): Spectral leakage (F10)

What happens to a peak when the window doesn't hold a whole number of cycles, and what does a Hann window do about it? Nothing in Parts 1 to 4 depends on this part, and it isn't in the time estimate.

1. Open the leakage explorer:

```
python3 src/explore.py --figure f10
```

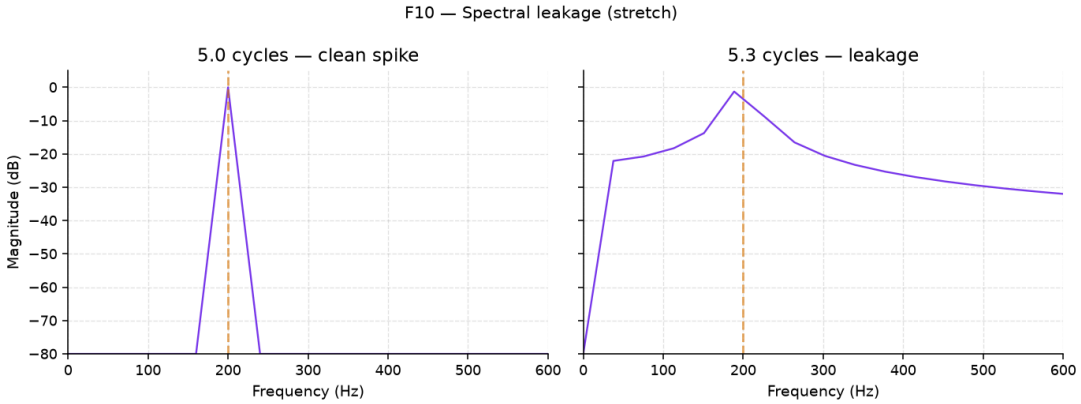


Figure 16.5: F10

2. Set "Cycles in window" to 5.0, then to 5.3.

   You should see a clean spike at 5.0 (the label reads `clean spike (integer cycles)`) and a smeared peak at 5.3 (`smeared peak (leakage!)`).

3. Leave it at 5.3 and tick "Hann window".

   You should see the skirts (the smeared energy on either side of the peak) drop. Close the window.

   Why do spectra measured on a real bench look soft instead of like textbook sticks?

   Answer: A2, in Answers at the end.

Finishing

You're done when `python3 src/check.py --guess <Hz>`, with the frequency you read off F8 yourself, prints `Correct!` (Part 2, step 3), and your predictions from Parts 1, 3 and 4 are written down next to what the scripts printed.

Questions

1. Why is the E5 peak 6.6 dB below A4 and not 6.0 dB? (Answered in Part 1, step 1.)
2. Below what sample rate does a 200 Hz tone alias? (Answered in Part 4, steps 2 and 3.)

Submit

Nothing to submit.

Next on the Course path: read Movement IV, Filters are tone knobs, Reading a Bode plot, Filtering a beat and More filter shapes, then do Lab 04: RC filter and Bode.

Answers

A1. Part 1: Your first FFT (F7)

A ratio of ×0.5 is about −6 dB, so E5 should sit about 6 dB below A4. The plot shows 60.0 − 53.4 = 6.6 dB. The extra 0.6 dB comes from leakage, which Finding a quiet note describes and the optional Part 5 shows you on F10. In the 0.2 s window, 440 Hz fits a whole number of cycles (88) but 659 Hz doesn't (131.8), so part of the E5 peak smears into the neighbouring bins and the peak itself comes out lower. If you change `659` to `660` in the script, the gap comes out at 6.0 dB.

A2. Part 5 (optional): Spectral leakage (F10)

It's leakage. A real measurement window doesn't hold a whole number of cycles of every tone in the signal, so the peaks smear sideways into their neighbouring frequency bins, just as they did at 5.3 cycles here. A window function such as Hann tapers the start and end of the recorded segment toward zero, which reduces the smear.

IV Movement IV: Tone controls

17 **Filters are tone knobs 65**
17 A circuit as a frequency machine 65
17 The RC low-pass 65

18 **Reading a Bode plot 67**
18 What each feature means 67
18 Gain and phase as a picture in time 67

19 **Filtering a beat 69**
19 A square wave through the filter 69
19 Parasitic RC 69

20 **More filter shapes 71**
20 The three plots 72
20 Where to go next 72

21 **Lab 04: RC filter and Bode 73**
21 Before you start 73
21 Part 1: The Bode explorer (F12) 73
21 Part 2: One Bode point in time (F13) 75
21 Part 3: A square wave through the RC (F14) . . . 76
21 Part 4: Three filter shapes (F15) 78
21 Finishing 79
21 Answers 79

17. Filters are tone knobs

The bass and treble knobs on a car stereo don't add any new notes to the music; they turn some frequencies up and others down. A **filter** does the same thing to a signal. It takes the signal's recipe of sines and turns each entry up or down, and it also delays each one a little. The report card for a filter is its **Bode plot**, which is the most-used graph in analog design.

A circuit as a frequency machine

A Bode plot shows two things, both plotted against frequency:

- **Magnitude**: how much of the signal comes out, $|H(f)|$, usually in dB
- **Phase**: how late it arrives, $\angle H(f)$, in degrees

The RC low-pass

A resistor R and a capacitor C in the classic low-pass arrangement let slow signals through and turn fast ones down, so together they act like a treble-cut knob. It's the first filter everyone builds, and its cutoff frequency is

$$f_c = \frac{1}{2\pi RC} \tag{17.1}$$

You don't need calculus here, because the course doesn't derive its magnitude and phase; take the two formulas below as given for now. They describe what you would measure if you swept the filter through frequencies one at a time:

$$|H(f)| = \frac{1}{\sqrt{1 + (f/f_c)^2}} \qquad \angle H(f) = -\arctan(f/f_c) \tag{17.2}$$

At $f = f_c$ the magnitude is $|H| = 1/\sqrt{2}$ (−3 dB) and the phase is −45°. With $R = 1\,\text{k}\Omega$ and $C = 100\,\text{nF}$, for example, $f_c \approx 1592$ Hz.

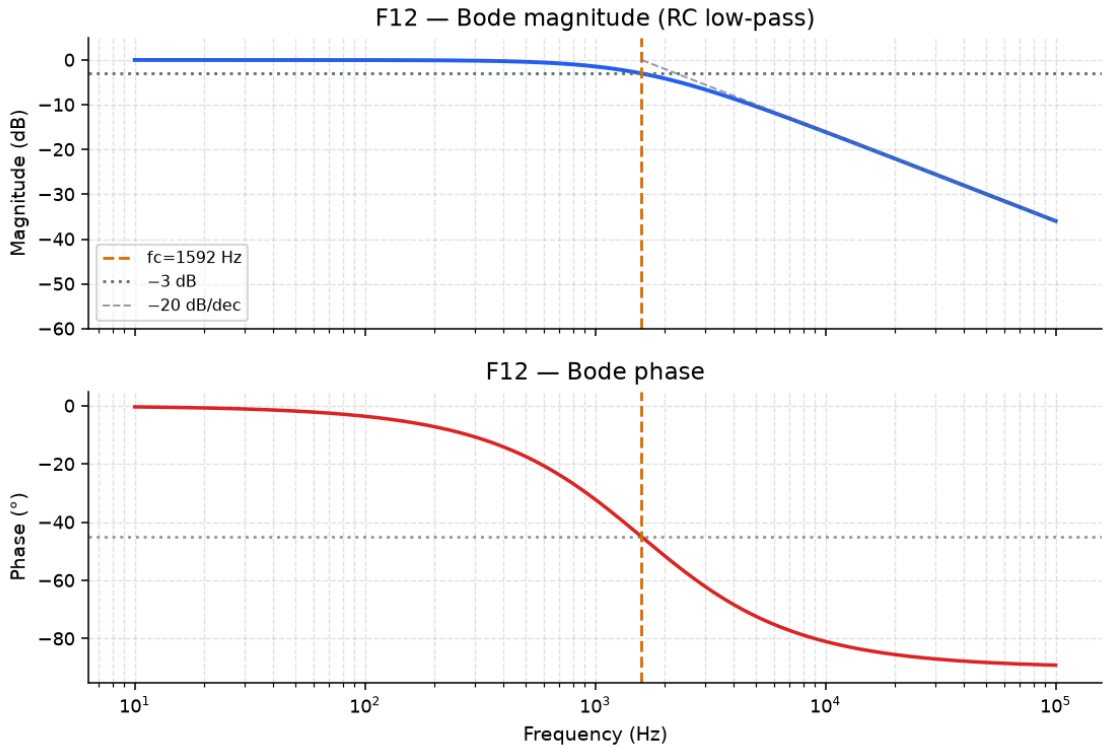


Figure 17.1: F12 — Bode explorer

Only the product RC matters, so if you double R and halve C, f_c stays where it is, like two different amp settings that happen to land on the same tone.

In Lab 04, Part 1, you'll move R and C on the F12 explorer, and then draw this Bode plot from the two formulas in a script of your own.

Next: Reading a Bode plot

18. Reading a Bode plot

A Bode plot shows what a filter does to every frequency, so it works as the filter's spec sheet. You can read it the way you'd read an EQ curve on a mixing console: where the curve is flat, those frequencies come through unchanged; where it slopes down, they come out quieter; and the phase panel tells you how much later each frequency arrives.

What each feature means

Feature	Meaning
Flat region	Passband: the signal comes through almost unchanged
−3 dB point	The cutoff frequency f_c, which is defined as the point where the gain is down 3 dB
−20 dB/decade slope	First-order roll-off: each ×10 in frequency divides the amplitude by 10
Phase ⟶ −90°	The phase a single RC approaches at high frequencies

Gain and phase as a picture in time

A Bode plot is a summary of many tests, each done at one frequency. Pick a frequency on the curve and drive the filter with a sine at that frequency. What comes out is still a sine at the same frequency, but it's **smaller**, which is the gain, and **later**, which is the phase. A network analyzer draws the two curves by doing exactly this, one sweep point at a time.

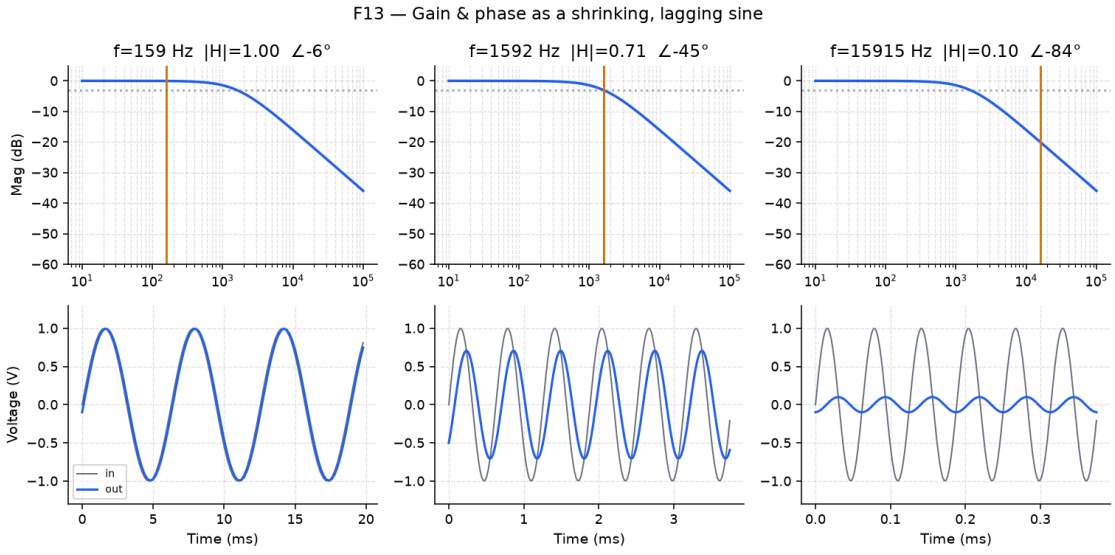


Figure 18.1: F13 — Bode linked to time

Take three points on the RC low-pass curve:

- A decade below f_c, the output is almost identical to the input.
- At f_c, the output shrinks to about 0.707× the input and lags it by about an eighth of a cycle (45°). Every Bode plot is anchored to this point, the −3 dB cutoff.
- A decade above f_c, the output is tiny and lags the input by nearly 90°.

In Lab 04, Part 2 you'll park the F13 marker at these three points, and then draw the input and output sines for each one in your own script.

Next: Filtering a beat

19. Filtering a beat

A kick drum heard through a wall sounds muffled. You still hear the thump, but the sharp attack is gone, smoothed into something rounder and softer. What you're hearing is a low-pass filter acting on a real signal with lots of harmonics, rather than on a lone sine. The clearest circuit example of the same effect is a square wave, so that's where we start.

A square wave through the filter

You saw in Movement II that a square wave is a pile of odd harmonics. The high harmonics are the ones that make the sharp edges, so when a low-pass filter turns them down, the edges round off. If you lower the cutoff further, the square turns into a soft, rounded blob, which is the electrical version of the muffled kick drum.

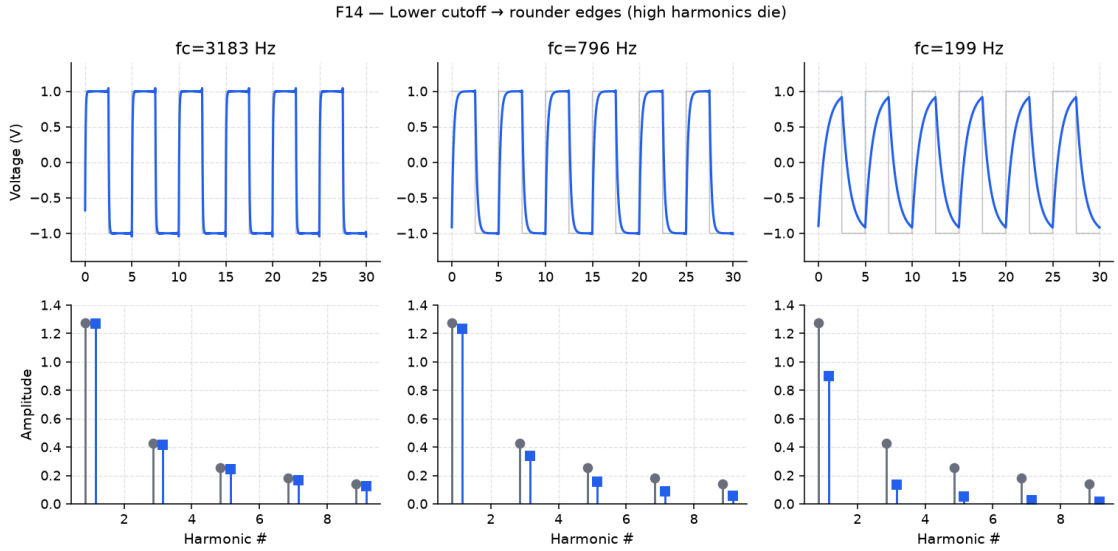


Figure 19.1: F14 — Square through RC

The knob that does this is the same one as on Filters are tone knobs: raise C and f_c drops, so the edges soften more with each step, and the high odd harmonics are the first to shrink.

In Lab 04, Part 3, you'll raise C on the F14 explorer and compare the harmonics going in with the ones coming out, and then filter a square wave in your own script.

Parasitic RC

Every node on a chip has some **parasitic** resistance and capacitance, that is, R and C that nobody drew on the schematic. That makes every node one of these low-pass filters, whether you designed one in or not, and it's why clock speeds have a ceiling. It's also a preview of the parasitic-extraction work you'll do in AD104.

Next: More filter shapes

20. More filter shapes

A three-band EQ has bass, mid and treble knobs, and behind each knob is a filter with its own shape, which keeps a different slice of the spectrum. Circuits make the same three shapes. The low-pass and the high-pass use the same R and C parts, arranged differently for each shape, but a band-pass that peaks as sharply as the one in F15 needs more than R and C.

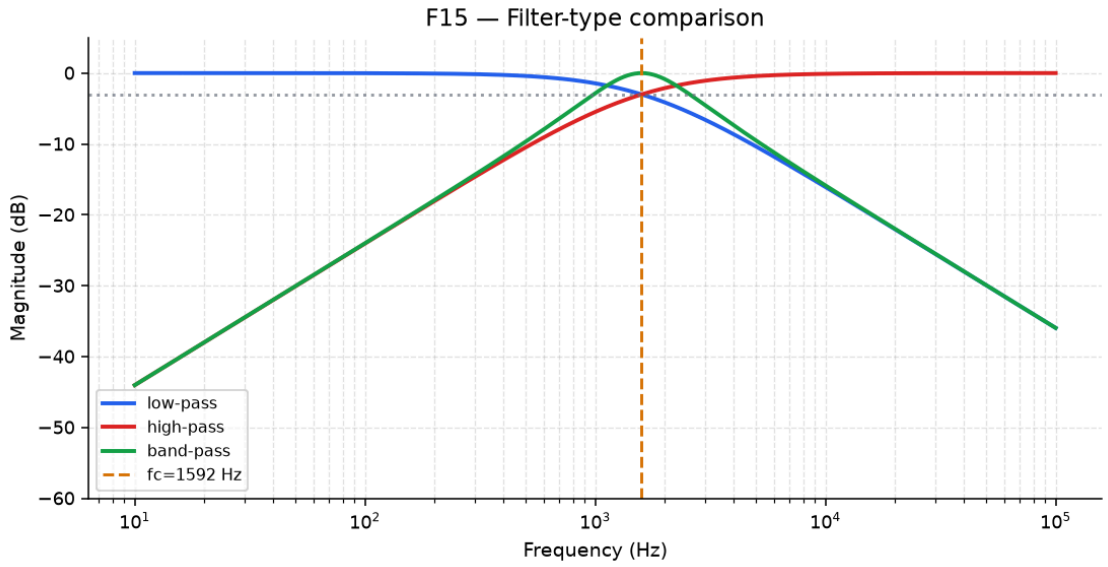


Figure 20.1: F15 — Filter types

Type	Passes	Blocks	EQ knob it resembles
Low-pass	Slow	Fast	Treble cut
High-pass	Fast	Slow	Bass cut
Band-pass	A middle band	Too slow and too fast	Mid boost / isolate

All three curves in F15 are built around the same f_c, and which side of f_c a filter keeps depends only on the shape of its formula. AD201 comes back to these shapes as blocks you design on purpose, not only as parasitics that turn up by accident.

The band-pass curves in this course come from the textbook second-order formula, which has a quality factor Q that sets how narrow the peak is: the higher the Q, the narrower the peak. F15 uses $Q = 1$ and Lab 04's `filter_shapes.py` uses $Q = 2$. You can build a band-pass from R and C alone by following an RC high-pass with an RC low-pass, but with equal parts in both sections its Q is only 1/3 and its peak sits 9.5 dB down, and no choice of R and C gets Q up to 0.5. A band-pass with $Q = 1$ needs an inductor or an amplifier as well.

In Lab 04, Part 4, you'll overlay the three shapes on the F15 explorer, and then plot them from their formulas in a script of your own.

The three plots

You now have the three plots this course set out to teach: a signal in time (Movement I), a signal as a recipe of sines (Movements II and III), and what a circuit does to that recipe (Movement IV). Analog engineers read these plots every day, and the later analog courses assume you can read them too.

That's the end of Movement IV. Next comes Lab 04: RC filter and Bode, and after it the capstone, Lab 05: Signal workshop, where you write a script that makes all three plots yourself.

Where to go next

Next course	What you carry forward
AD102	Passives on silicon: fabricating R, C, L
AD103	Nonlinear devices (diode, MOSFET) in XSchem
AD104	Layout parasitics (why every node is an RC)
AD202 (planned)	Sampling and ADCs, where Nyquist matters for real

21. Lab 04: RC filter and Bode

You'll sweep an RC low-pass on a live Bode plot, see what one point on that plot does to a sine (it comes out smaller and later than it went in), filter a square wave until its edges round off, and compare low-pass, high-pass and band-pass shapes. Each explorer has a matching snippet that computes the same thing from the formulas, and before you run each one you'll use the same formulas to work out what it should show.

Lab package: `labs/lab-04-rc-bode/`

Before you start

- Finish Lab 03.
- Read Movement IV: Filters are tone knobs, Reading a Bode plot, Filtering a beat and More filter shapes.
- In a terminal in the noVNC desktop, go to the lab folder:

```
. /foss/designs/common/.designinit
mod ad101
cd labs/lab-04-rc-bode
```

Time: about 1.5 hours.

The formulas you'll use throughout:

$$f_c = \frac{1}{2\pi RC} \qquad |H(f)| = \frac{1}{\sqrt{1 + (f/f_c)^2}} \qquad \angle H(f) = -\arctan(f/f_c) \tag{21.1}$$

As before, each part pairs an explorer (`python3 src/explore.py --figure <id>`) with a snippet you save as a new `.py` file in this folder and run with `python3 <file>.py`.

Part 1: The Bode explorer (F12)

What sets an RC low-pass's cutoff, and what do its magnitude and phase do there?

1. Open the Bode explorer:

```
python3 src/explore.py --figure f12
```

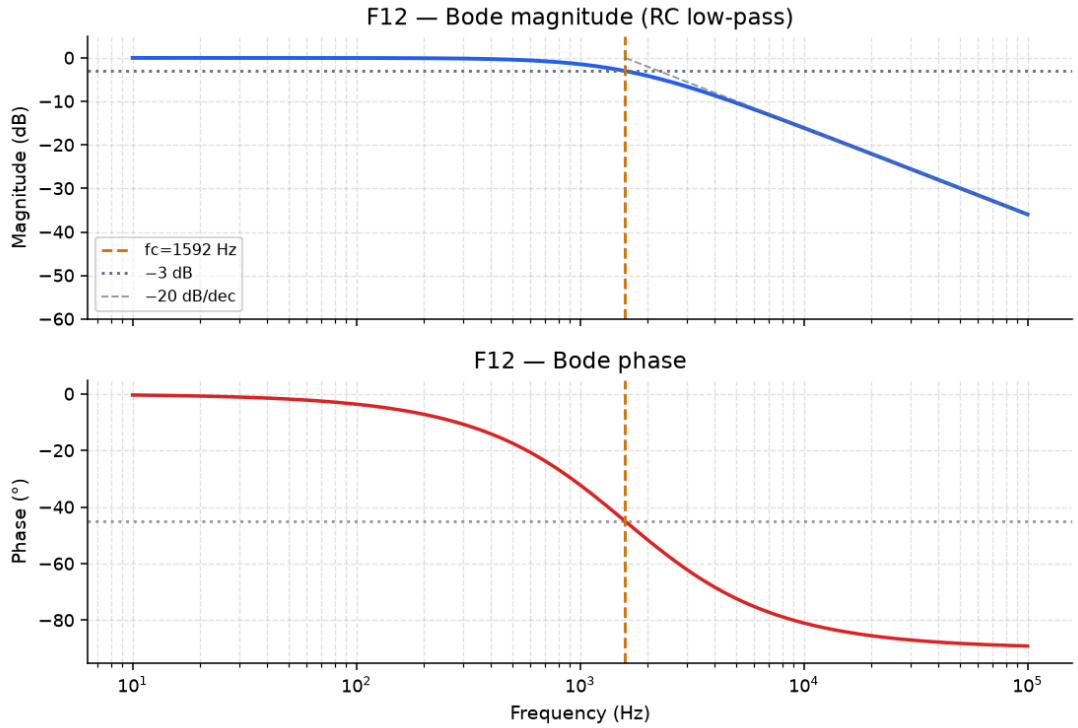


Figure 21.1: F12

It opens with R = 1000 Ω and C = 100 nF, and the box at the bottom starts with `fc = 1592 Hz`.

2. Check that number against $f_c = 1/(2\pi RC)$ by hand, and write down your result.
3. Look at where the magnitude curve crosses the dotted −3 dB line, and at the phase curve below it.

   You should see the magnitude cross −3 dB at the dashed f_c marker, the phase pass −45° there, and the dashed asymptote fall at −20 dB/decade past f_c.

4. Move R and C and watch f_c follow $1/(2\pi RC)$. Next you'll set R = 2000 Ω and C = 50 nF (double R, halve C). Predict the new f_c first and write it down, then set them.

   You should see the box start with `fc = 1592 Hz` again. Close the window.

   Why didn't f_c move?

   Answer: A1, in Answers at the end.

5. Save this as `bode.py` and run it. It draws a Bode plot from the two formulas alone:

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

R, C = 1_000.0, 100e-9              # 1 kΩ, 100 nF
fc = 1 / (2 * np.pi * R * C)
print(f"cutoff fc ≈ {fc:.0f} Hz")

f = np.logspace(1, 5, 400)          # 10 Hz to 100 kHz, log-spaced
mag = 1 / np.sqrt(1 + (f / fc) ** 2)
mag_db = 20 * np.log10(mag)
phase_deg = -np.rad2deg(np.arctan(f / fc))

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(7, 5))
ax1.semilogx(f, mag_db)
ax1.axvline(fc, color="orange", linestyle="--")
```

```
ax1.axhline(-3, color="gray", linestyle=":")
ax1.set_ylabel("Magnitude (dB)")

ax2.semilogx(f, phase_deg)
ax2.axvline(fc, color="orange", linestyle="--")
ax2.axhline(-45, color="gray", linestyle=":")
ax2.set_ylabel("Phase (°)")
ax2.set_xlabel("Frequency (Hz)")
plt.tight_layout()
plt.show()
```

You should see `cutoff fc ≈ 1592 Hz` printed, and a two-panel Bode plot where the magnitude crosses the dotted −3 dB line and the phase crosses the dotted −45° line, both at the orange dashed f_c line.

6. Pick your own values of `R` and `C`. Write down the cutoff you expect from $f_c = 1/(2\pi RC)$, then change the two values in the script and re-run. Keep your numbers for Finishing.

   You should see the printed cutoff match your prediction, with the −3 dB crossing still on the orange line.

Part 2: One Bode point in time (F13)

What does one point on the Bode curve do to a sine that goes through the filter?

1. Open the marker explorer:

```
python3 src/explore.py --figure f13
```

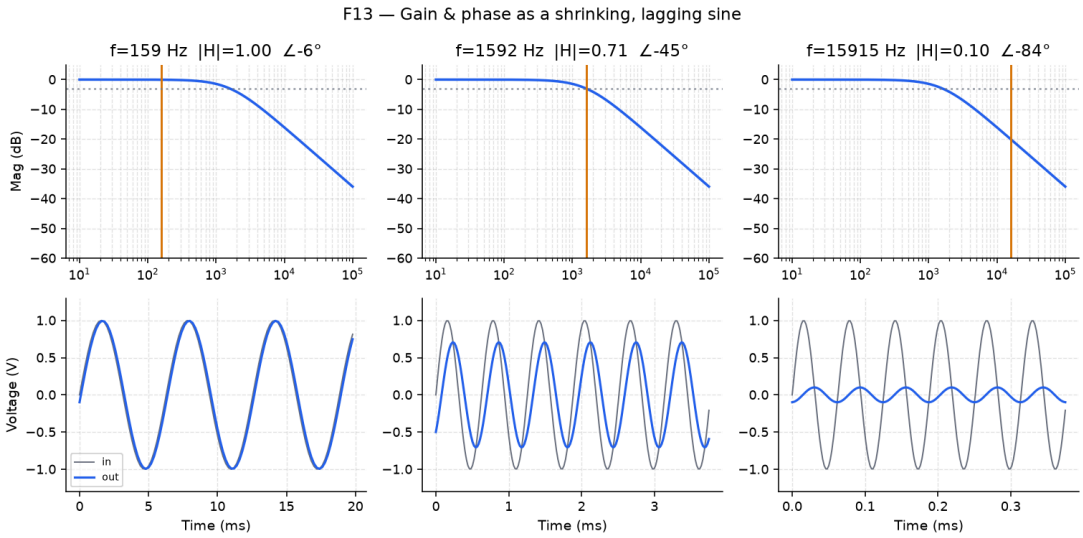


Figure 21.2: F13

The left panel is the Bode magnitude for R = 1 kΩ, C = 100 nF. The right panel shows the input and output sines at the marker's frequency. The marker starts as close to f_c as the slider's 0.02 steps allow, so the slider reads 3.2 and the box reads `f = 1585 Hz  |H| = 0.709 (-3.0 dB)  phase = -45°`.

2. With the marker at f_c, compare the two sines.

   You should see the output at about 0.707× the input, lagging by about 45° (an eighth of a cycle).

3. Before you move the slider, use the formulas at the top of this page to predict $|H|$ and the phase a decade below f_c ($f = f_c/10$) and a decade above ($f = 10f_c$). Write the four numbers down. Then move the slider, which sets `log10(f)`: moving it down by 1, to 2.2, puts the marker a decade below f_c, and moving it up to 4.2 puts it a decade above.

You should see the output almost identical to the input a decade below ($|H| \approx 0.995$, phase about $-6°$), and tiny with nearly $-90°$ of lag a decade above ($|H| \approx 0.100$, -20 dB, phase about $-84°$). Close the window.

4. Save this as `bode_point.py` and run it. At each of three frequencies it makes an input sine and draws the output that the Bode formulas predict for it:

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

R, C = 1_000.0, 100e-9
fc = 1 / (2 * np.pi * R * C)

fs = 200_000.0
t = np.arange(0, 0.02, 1 / fs)

for test_freq, label in [(fc / 10, "a decade below fc"), (fc, "at fc"), (fc * 10,
"a decade above fc")]:
    drive = np.sin(2 * np.pi * test_freq * t)
    mag = 1 / np.sqrt(1 + (test_freq / fc) ** 2)
    phase_rad = -np.arctan(test_freq / fc)
    output = mag * np.sin(2 * np.pi * test_freq * t + phase_rad)

    plt.figure(figsize=(6, 2.5))
    plt.plot(t * 1e3, drive, label="input", alpha=0.6)
    plt.plot(t * 1e3, output, label="output", linewidth=2)
    plt.title(f"{label}: gain={mag:.3f}, phase={np.rad2deg(phase_rad):.1f}°")
    plt.xlabel("Time (ms)")
    plt.legend(fontsize=8)
    plt.xlim(0, 3 / test_freq * 1e3)
    plt.tight_layout()
    plt.show()
```

The three plots open one at a time; close each to see the next. You should see the same three points on the Bode curve that F13 showed you, drawn as time plots and titled `a decade below fc: gain=0.995, phase=-5.7°`, `at fc: gain=0.707, phase=-45.0°` and `a decade above fc: gain=0.100, phase=-84.3°`. Compare them with your predictions from step 3.

Part 3: A square wave through the RC (F14)

What does a low-pass do to a square wave, and which harmonics go first?

1. Open the square-wave explorer:

```
python3 src/explore.py --figure f14
```

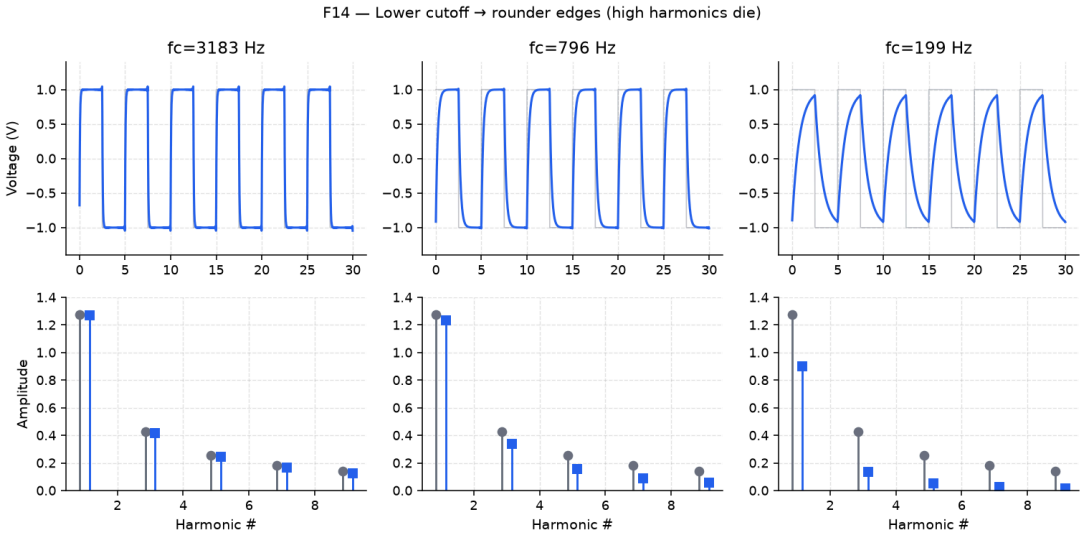


Figure 21.3: F14

A 200 Hz square wave goes through an RC low-pass with R = 1 kΩ and C = 200 nF. The box reads `fc = 796 Hz`. The stem plot on the right compares harmonics 1, 3, 5, 7 and 9 of the input (gray) and the output.

2. Raise C, which lowers f_c, and compare input and output on both panels.

   You should see the edges round off, and the output stems for the high odd harmonics (5, 7, 9) shrink first. Close the window.

3. Save this as `filter_square.py`, but don't run it yet. It filters a 500 Hz square wave by scaling its spectrum by $H(f)$ and transforming back, for three values of C:

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

fs = 50_000.0
t = np.arange(0, 0.01, 1 / fs)
f0 = 500.0
square = np.sign(np.sin(2 * np.pi * f0 * t))

def rc_lowpass_filter(x, fs, R, C):
    """Apply an RC low-pass by scaling the spectrum, then transforming back."""
    n = len(x)
    freqs = np.fft.rfftfreq(n, d=1 / fs)
    fc = 1 / (2 * np.pi * R * C)
    mag = 1 / np.sqrt(1 + (freqs / fc) ** 2)
    phase = -np.arctan(freqs / fc)
    H = mag * np.exp(1j * phase)
    X = np.fft.rfft(x)
    return np.fft.irfft(X * H, n=n)

for C in (10e-9, 100e-9, 1e-6):          # progressively lower cutoff
    filtered = rc_lowpass_filter(square, fs, R=1_000.0, C=C)
    fc = 1 / (2 * np.pi * 1_000.0 * C)
    plt.plot(t * 1e3, filtered, label=f"C={C*1e9:.0f} nF (fc≈{fc:.0f} Hz)")

plt.plot(t * 1e3, square, color="gray", linestyle="--", label="input square",
alpha=0.5)
plt.xlabel("Time (ms)")
```

```
plt.ylabel("Voltage (V)")
plt.legend(fontsize=8)
plt.show()
```

Before you run it, work out f_c for each of the three capacitors with R = 1 kΩ, and write them down. Then run it.

You should see three filtered versions of the dashed input square, labelled `C=10 nF (fc≈15915 Hz)`, `C=100 nF (fc≈1592 Hz)` and `C=1000 nF (fc≈159 Hz)`. The edges get softer with each step up in C, from nearly square at 10 nF to a small, rounded wave at 1000 nF.

Why do the edges soften as C goes up?

Answer: A2, in Answers at the end.

Part 4: Three filter shapes (F15)

How do low-pass, high-pass and band-pass shapes differ around the same f_c?

1. Open the filter-type comparison:

```
python3 src/explore.py --figure f15
```

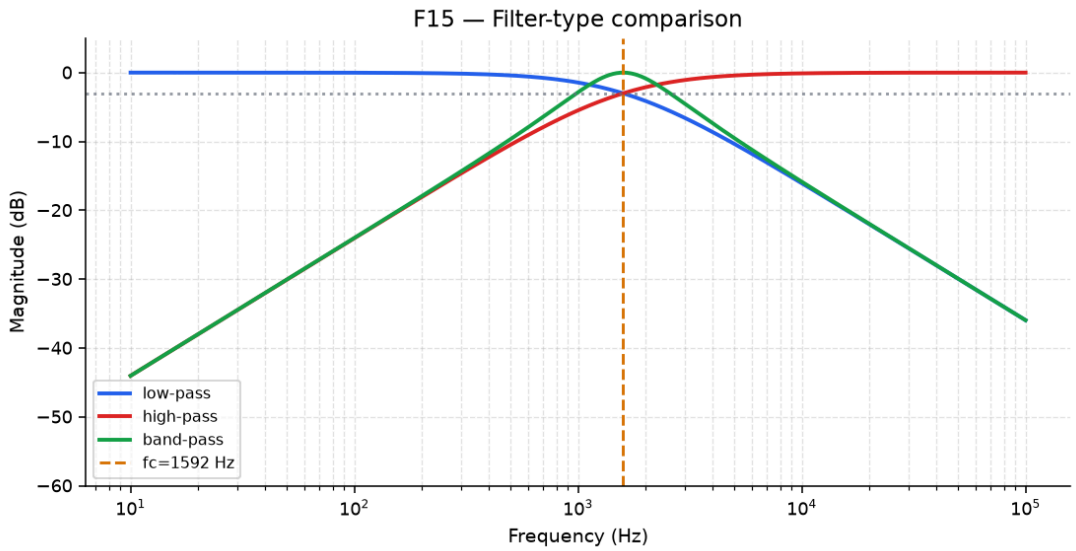


Figure 21.4: F15

2. Look at the low-pass, high-pass and band-pass curves together, then move R and C to shift f_c.

   You should see three different pass/stop patterns around the same f_c, all moving together when f_c moves. Close the window.

3. Save this as `filter_shapes.py`, but don't run it yet. It computes all three curves from the same `fc`:

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

R, C = 1_000.0, 100e-9
fc = 1 / (2 * np.pi * R * C)
f = np.logspace(1, 5, 400)
u = f / fc

lowpass = 1 / np.sqrt(1 + u**2)
highpass = u / np.sqrt(1 + u**2)
```

```
Q = 2.0
bandpass = (u / Q) / np.sqrt((1 - u**2) ** 2 + (u / Q) ** 2)

plt.semilogx(f, 20 * np.log10(lowpass), label="low-pass")
plt.semilogx(f, 20 * np.log10(highpass), label="high-pass")
plt.semilogx(f, 20 * np.log10(bandpass), label="band-pass")
plt.axvline(fc, color="gray", linestyle="--", label="fc")
plt.ylim(-40, 5)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude (dB)")
plt.legend()
plt.show()
```

At $f = f_c$, `u` is 1. Before you run it, put $u = 1$ into the `lowpass` and `highpass` lines and work out both values in dB. Then run it.

You should see the three curves meet the dashed `fc` line from different sides: the low-pass keeps what's below it, the high-pass keeps what's above it, and the band-pass peaks there. The low-pass and high-pass curves both pass through −3 dB at the dashed line.

Finishing

You're done when `bode.py`, run with an R and C you chose, prints the cutoff you predicted (Part 1, step 6), and your four predictions for the decade points match F13 and the `bode_point.py` titles (Part 2, steps 3 and 4).

Questions

1. Why does doubling R and halving C leave f_c where it was? (Answered in Part 1, step 4.)
2. Why does raising C round off a square wave's edges? (Answered in Part 3, step 3.)

Submit

Nothing to submit.

Next on the Course path: do Lab 05: Signal workshop, where you write a script that synthesizes, analyzes and filters a signal on its own.

Answers

A1. Part 1: The Bode explorer (F12)

Only the product RC appears in $f_c = 1/(2\pi RC)$, and 2000 Ω × 50 nF is the same product as 1000 Ω × 100 nF, so the cutoff stays put. If you double R without halving C, the box reads `fc = 796 Hz`.

A2. Part 3: A square wave through the RC (F14)

Raising C lowers f_c. A square wave is a pile of odd harmonics, and it's the high ones that make its edges sharp, so the lower f_c goes, the more of those high harmonics the filter turns down and the rounder the edges get. Filtering a beat has the same picture.

V

Capstone

22 **Lab 05: Signal workshop 83**
22 Before you start 83
22 How the scaffold works 83
22 Run the empty scaffold 84
22 Part 1: Build and plot the chord 84
22 Part 2: Compute and plot the spectrum 84
22 Part 3: Filter the chord 84
22 Part 4: Export and listen 85
22 Finishing 86
22 Answers 86

22. Lab 05: Signal workshop

You'll write a signal-analysis script yourself, one function at a time: it synthesizes a chord out of harmonic-rich tones, plots its waveform and spectrum, filters it with an RC low-pass, and exports the chord before and after filtering so you can hear the filter change its timbre without changing its pitch.

This lab works differently from Labs 01–04: instead of moving sliders, you fill in a scaffold, run it, and listen to what it produces.

Lab package: `labs/lab-05-signal-workshop/`

Before you start

- Finish Labs 01–04 first, because this lab puts the ideas from their snippets together into one script.
- Read all four movements, What is a signal? through More filter shapes.
- Keep Music and signals open for its note-frequency table.
- In a terminal in the noVNC desktop, go to the lab folder:

```
. /foss/designs/common/.designinit
mod ad101
cd labs/lab-05-signal-workshop
```

Time: about 2 hours.

How the scaffold works

Open `src/workshop.py` in an editor. Seven functions are stubbed out with `raise NotImplementedError(...)`, and each one has a docstring that tells you exactly what to write. The part numbers below match the `Part` labels in those docstrings and in the comments in `run()`:

Part	Function	What it does
1	`synthesize_chord`	Builds a C-major triad out of harmonic-rich tones
1	`plot_waveform`	Plots voltage against time
2	`compute_spectrum`	Wraps `common.adsig.spectrum`
2	`plot_spectrum`	Plots magnitude in dB against frequency
3	`apply_lowpass`	Filters a signal by multiplying its spectrum by $H(f)$
4	`export_wav`	Writes a signal to a `.wav` file
4	`play_audio`	Plays a `.wav` file, with a fallback if there's no audio device

The helpers the docstrings mention (`harmonic_sum`, `house_style`, `spectrum`, `to_db`, `rc_lowpass`) are already imported at the top of the file from `labs/common/adsig.py`, the course's signal library.

The script's `run()` function is already written. It calls your functions in that order and builds three figures from them: the waveform, the spectrum, and a before/after comparison that uses `plot_spectrum` twice. You'll run it two ways:

```
python3 src/workshop.py --headless  # save PNGs + WAVs into results/, skip live playback
python3 src/workshop.py             # full run: plot windows + live audio, if available
```

Until every function is filled in, the script stops at the first one that still raises `NotImplementedError`, and the error message says which step that is. That's how you'll know a part works: once its function is filled in, the error moves on to the next one.

If you're stuck on one function, `solutions/workshop_solution.py` has a filled-in reference, but try to get further on your own first. If you do run it, the end of this page shows how to do that without overwriting your own results.

Run the empty scaffold

1. Run it before changing anything:

```
python3 src/workshop.py --headless
```

You should see a traceback ending in `NotImplementedError: TODO: synthesize the chord — see the docstring above`.

Part 1: Build and plot the chord

How do you turn three notes into one harmonic-rich chord, and plot it?

1. Fill in `synthesize_chord`. Its docstring has you build each note with `harmonic_sum(t, fundamental=freq, harmonics=[1, 3, 5, 7, 9])`, add the three notes, and divide by the number of notes so the result stays roughly within ±1. The notes are `CHORD_NOTES` at the top of the file: C4, E4 and G4 at 261.6, 329.6 and 392.0 Hz.

   This is the chord from Lab 02, Part 1, except that each note now has odd harmonics, like the square wave from Lab 02, Part 2, and those high harmonics give the filter in Part 3 something it can audibly change.

2. Fill in `plot_waveform`: time in milliseconds on the x-axis, voltage on the y-axis, with `house_style(ax)` applied so it matches Labs 01–04.

3. Run `python3 src/workshop.py --headless` again.

You should see the traceback move on and end in `NotImplementedError: TODO: compute the spectrum — see the docstring above`. `run()` saves all three figures together after the filter step, so you'll check your waveform plot once Part 3 works.

Part 2: Compute and plot the spectrum

How do you get from the chord's samples to its spectrum in dB?

1. Fill in `compute_spectrum`. It calls `spectrum(v, fs)` from `common.adsig`, which wraps `numpy.fft.rfft` the way your Lab 03 snippets did by hand, and also scales the result so that a pure tone's peak comes out close to its amplitude rather than at a raw FFT magnitude.

2. Fill in `plot_spectrum`: magnitude in dB (use `to_db`) against frequency, with the x-axis limited to something readable for a chord. The docstring suggests `ax.set_xlim(0, 3000)`.

3. Run the script again.

You should see the error move on to `NotImplementedError: TODO: apply the RC low-pass — see the docstring above`.

Part 3: Filter the chord

What does an RC low-pass do to the chord's spectrum?

To filter a signal in the frequency domain, you transform it, multiply by the filter's response $H(f)$, and transform back. For the RC low-pass:

$$H(f) = \frac{1}{\sqrt{1 + (f/f_c)^2}}\, e^{-j \arctan(f/f_c)} \qquad f_c = \frac{1}{2\pi RC} \tag{22.1}$$

The first factor is the magnitude and the exponential carries the phase, the same two curves as the Bode plot in Lab 04. j is the imaginary unit ($j^2 = -1$; electrical engineers write j instead of i). You don't need to work with complex numbers by hand here, because numpy does the arithmetic once you've built `H`.

1. Fill in `apply_lowpass`, following the four numbered steps in its docstring: get `freqs`, get the magnitude and phase from `rc_lowpass` (the phase comes back in degrees, so convert it with `np.deg2rad`), build the complex `H`, then multiply in the frequency domain and transform back with `np.fft.irfft`. It's the same method as the `rc_lowpass_filter` snippet in Lab 04, Part 3.
2. `FILTER_R` and `FILTER_C` at the top of `src/workshop.py` set the filter to R = 1 kΩ and C = 150 nF. Work out f_c and write it down. Then run `python3 src/workshop.py --headless` again.

   You should see `Filter cutoff fc = 1061 Hz`, then three `wrote …` lines for `results/workshop-waveform.png`, `results/workshop-spectrum.png` and `results/workshop-before-after.png`. The run then stops at `NotImplementedError: TODO: export a WAV file — see the docstring above`.
3. Open the three PNGs and check them:
   - Waveform: a dense, harmonic-rich wiggle, not a clean sine, because each note is built from several harmonics.
   - Spectrum: three clear peaks near 262 Hz (C4), 330 Hz (E4) and 392 Hz (G4), plus smaller peaks at their odd harmonics.
   - Before/after: the three fundamental peaks survive filtering, and the higher-harmonic peaks are visibly shorter after filtering than before.

Part 4: Export and listen

Can you hear what the filter did?

1. Fill in `export_wav`. Its docstring gives the normalization to 16-bit integers and the standard-library `wave` calls.
2. Fill in `play_audio`. Try `sounddevice` first, with the imports and the playback all inside one `try/except Exception`. If it fails, print the file's path and tell the user to open it from their host's file manager. Playback failing must never crash the script.

   Neither `sounddevice` nor `soundfile` (which the docstring uses to read the WAV back) is installed in the course image. Both are listed as optional in this lab's `requirements.txt`; install them with `pip install --user sounddevice soundfile` if you want to try live playback. That install goes into the container, not your `workspace` folder, so you'd have to repeat it if you remove the container. Without them your fallback runs instead, and that's fine, because the noVNC desktop may not have a sound device anyway; see Python in the workbench.
3. Run the headless version:

   ```
   python3 src/workshop.py --headless
   ```

   You should see the same output as Part 3, then `wrote …/results/chord-unfiltered.wav` and `wrote …/results/chord-filtered.wav`, with no traceback.
4. Run the full version:

   ```
   python3 src/workshop.py
   ```

   The three plots open first. Close them, and the script plays the unfiltered chord and then the filtered one, or prints your fallback message for each if live playback isn't available.
5. Listen to `chord-unfiltered.wav` and `chord-filtered.wav`. If playback doesn't work over your noVNC session, open them from your host machine: they're in `workspace/modules/ad101/labs/lab-05-signal-workshop/results/`.

   You should hear the same notes in both files. The filtered one sounds duller and more muffled, but its pitch is the same.

   Why does the filter change the chord's timbre but not its pitch?

Answer: A1, in Answers at the end.

Finishing

You're done when `python3 src/workshop.py --headless`, running your own seven functions, finishes without a traceback and writes three PNGs and two WAVs into `results/`, and your before/after spectrum shows the three fundamentals surviving and the higher harmonics shorter.

Questions

1. Why does the filter change the chord's timbre but not its pitch? (Answered in Part 4, step 5.)

Submit

Nothing to submit.

To compare your work with the reference, run the solution with its own output folder so it doesn't overwrite your files:

```
python3 solutions/workshop_solution.py --headless --out results/solution
```

Next on the Course path: this was the last step of AD101. Carry on with AD102: Linear Circuits & Fabrication.

Answers

A1. Part 4: Export and listen

Pitch is set by the fundamentals, 261.6, 329.6 and 392.0 Hz, all well below f_c = 1061 Hz, so they come through almost unchanged; that's the three peaks surviving in your before/after spectrum from Part 3. Timbre is the recipe of higher harmonics on top of each note, and the low-pass turns those down.

Appendices

A. Reading plots

This page is a cheat sheet for the three kinds of plot that analog engineers read every day, along with the mistakes people usually make when reading each one.

1. Waveform (time domain)

On a waveform the horizontal axis is time, and the vertical axis is voltage (or current).

Mark this	How
Period T	The time between matching points on consecutive cycles
Frequency f	$1/T$
Peak-to-peak	The top of the swing minus the bottom
Amplitude A	For a sine, half the peak-to-peak value, measured from the center line
DC offset	The vertical position of the center line
Phase / delay	The horizontal shift between two traces

The usual misreads are taking the peak value for the peak-to-peak value, which puts you off by a factor of 2, and reading the frequency from a zoomed-in window that shows only a fraction of a cycle.

2. Spectrum (frequency domain)

On a spectrum the horizontal axis is frequency (in Hz), and the vertical axis is magnitude, on either a linear or a dB scale.

Feature	Meaning
Spike / stem	A tone (or a harmonic) at that frequency
Height	How strong that tone is relative to the others
Noise floor	Sets the smallest feature you can see
Comb of odd harmonics	A square-ish or clipped waveform
Sidebands around a carrier	Modulation (AM, for example)

The usual misread is expecting the clean, thin sticks of a textbook spectrum. A real FFT has a finite bin width and leakage, so each spike has some width and spreads a little into the bins around it. When you compare large and small tones on one plot, use dB, so that the small tones are still visible next to the large ones.

dB conversion table

Linear ratio	dB ($20 \log_{10}$)
×10	+20

Linear ratio	dB ($20 \log_{10}$)
×2	≈ +6
×1	0
×0.707 ($1/\sqrt{2}$)	−3
×0.5	≈ −6
×0.1	−20

3. Bode plot (system response)

A Bode plot is two graphs stacked one above the other, both plotted against log frequency:

1. Magnitude in dB
2. Phase in degrees

Feature	Meaning
Flat magnitude	The passband
−3 dB crossing	The cutoff frequency f_c
−20 dB/decade	A first-order roll-off
−45° at f_c	The signature of a single RC low-pass
Phase ⟶ −90°	The high-frequency limit of a single RC

The usual misreads are treating a plot with a linear frequency axis as a Bode plot (a Bode plot always uses log frequency), and confusing gain in dB with gain as a plain ratio.

Quick mental model

```
signal itself     →  waveform + spectrum
what a circuit
does to signals   →  Bode plot
```

Labs: Lab 01 · Lab 02 · Lab 03 · Lab 04 · Lab 05

B. Music and signals

Every idea in this course has a musical counterpart, and this page collects them in one place. Use it as a translation table when the analogy on a guide page doesn't click the first time you read it.

Glossary

Musical term	Signal-processing term	Where it's covered
Pitch	Frequency (f)	Pitch and loudness
Loudness	Amplitude (A)	Pitch and loudness
Beat length	Period ($T = 1/f$)	Pitch and loudness
Two voices out of sync	Phase / delay (ϕ)	Two voices in time
Chord	Superposition (signals added)	Adding signals
Timbre (why instruments sound different)	Harmonic content / spectrum	Timbre and harmonics
Rhythm's "on" fraction	Duty cycle	Rhythm and duty cycle
Equalizer display	FFT / spectrum	The FFT as an equalizer
Distorted / crunchy tone	Harmonic distortion (THD)	Distortion and THD
Wagon-wheel effect	Aliasing	Sampling fast enough
Bass/treble knob	Filter (low-pass / high-pass)	Filters are tone knobs
EQ curve	Bode plot	Reading a Bode plot
Muffled sound through a wall	Low-pass-filtered signal	Filtering a beat

Octaves and doubling

Going up one **octave** doubles the frequency. A3 (220 Hz) and A4 (440 Hz) sound like "the same note" an octave apart, because 440 is exactly double 220. Equal steps in music are equal ratios of frequency rather than equal differences in Hz, so the frequency axes in this course, such as the one on a Bode plot, are often drawn on a log scale.

Note-name to frequency table

Standard concert pitch is A4 = 440 Hz. The table includes every note that the lab snippets and Lab 05 use, along with a few others.

Note	Frequency (Hz)
A3	220.0
C4	261.6
D4	293.7

Note	Frequency (Hz)
E4	329.6
F4	349.2
G4	392.0
A4	440.0
B4	493.9
C5	523.3
E5	659.3
A5	880.0

A C-major triad (C4, E4 and G4) is the chord you'll find in Adding signals, Lab 02 and Lab 05.

Related

- Reading plots: how to read the three plots themselves
- Getting started: how the course is organized

C. Python in the workbench

This page explains how to run the AD101 explorers and scripts inside the IIC-OSIC-TOOLS noVNC desktop you set up in IC101, and what to do when something goes wrong. The first time you load the course, Lab 01, Part 1 walks you through it.

One-time setup

Start noVNC the way you did in IC101 (`./scripts/start_vnc.sh` from your `workspace` folder), open a terminal in the desktop, and run:

```
. /foss/designs/common/.designinit
mod add ad101               # first time only: clones into modules/ad101
mod ad101                   # → /foss/designs/modules/ad101
python3 scripts/check_env.py
```

If you'd rather not use `mod add`, you can clone the repo on the host instead:

```
# on the host, inside your workspace clone
cd workspace/modules
git clone https://github.com/uoftasic/ad101.git
```

`check_env.py` should print `OK` for numpy, for matplotlib and for the `adsig` smoke line, and finish with `=== All checks passed ===`.

Running an explorer

```
cd labs/lab-01-signal-explorer
python3 src/explore.py              # interactive windows
python3 src/explore.py --figure f1  # one figure
python3 src/explore.py --headless   # PNGs into results/ (no GUI)
```

The terminal stays busy until you close the plot windows, so you won't get a prompt back while they're open. You can close a window with its × button, or by pressing `q` while the figure has focus.

Running the snippets

The short Python snippets in Labs 01–04 import only `numpy` and `matplotlib`. Unlike the explorers, they don't need `_path.py` or `mod ad101`, so they run in any Python 3 that has those two packages, whether that's the noVNC desktop, your host machine or your own laptop. Save each snippet as its own `.py` file and run it with `python3 <file>.py`. In the desktop, anything you save under `/foss/designs` is stored in your `workspace` folder on the host, so it survives a container restart.

GUI backend

The interactive explorers need a matplotlib GUI backend (Tk or Qt) and a display to draw on, and the noVNC desktop provides both. On a host machine without a display, `check_env.py` reports the `Agg` backend instead. That's fine for `--headless` and for the maintainers' `build_figures.py` (described further down), because they only write PNG files and never open a window. If you get a backend warning inside IIC-OSIC-TOOLS and noVNC, run `check_env.py` again, and if it still warns, use one of the fixes below or run with `--headless`.

If the GUI won't open

`check_env.py` may report `WARN no interactive GUI backend`, which means the explorers have nothing to open their windows with. You have three ways round it:

1. Use headless mode. You lose the interactive windows, but the exported PNGs are the same figures you see on the guide pages, so you can still learn from them:

   ```
   python3 src/explore.py --headless
   ```

2. Install a GUI backend inside the container:

   ```
   pip install --user PyQt5
   # or make sure Tk is available; then re-run check_env.py
   ```

3. Check that you're typing in a terminal inside the noVNC desktop. A terminal on the host has no display for the windows to open on.

Saving a figure

- To save the figure in an explorer or snippet window, click the save icon (the floppy disk) in the window's toolbar.
- To save every figure from an explorer at once, run `python3 src/explore.py --headless`, which writes PNGs into the lab's `results/` folder.
- To save from your own snippet, add `plt.savefig("my-capture.png", dpi=140)` just before `plt.show()`.

Listening to a filtered signal (Lab 05)

Lab 05 exports and plays audio so that you can hear what a filter does to a chord. Live playback needs a sound device, and the noVNC desktop may not pass one through from the host, so the lab has you write a fallback, in the same spirit as `--headless` when there's no GUI backend:

1. The `play_audio` function you write tries live playback with `sounddevice`. Neither `sounddevice` nor `soundfile` is installed in the course image; both are listed as optional in `labs/lab-05-signal-workshop/requirements.txt`, and you can install them with `pip install --user sounddevice soundfile`.
2. If playback fails, your fallback prints the path of each `.wav` file, which the script has already written into `results/`, and you can open them from your host machine's file manager or media player.

Either way, you can compare the filtered chord with the unfiltered one.

Regenerating the docs figures (maintainers)

From the AD101 repo root:

```
python3 scripts/build_figures.py
# → docs/assets/img/f01-….png … f15-….png
```

The noise figures use fixed random seeds (`RNG_SEED = 42` in `labs/common/adsig.py`, and `HIDDEN_SEED = 7` for F8 in Lab 03's `src/explore.py`), so a rebuild produces the same PNGs and doesn't leave a pile of changed images in git.

Common errors

Symptom	Fix
`ModuleNotFoundError: common`	Run it from the lab folder as `python3 src/explore.py`, which

Symptom	Fix
	uses `_path.py` to find `common`, or run `mod ad101` first
`ModuleNotFoundError: numpy`	Install the packages with `pip install --user numpy matplotlib`
Blank or frozen window over noVNC	Open fewer windows (`--figure f1` opens just one), close any extras, and check your VNC bandwidth
`check.py` says `Not quite`	Read the spectrum peak again; your answer has to be within ±5 Hz
`NotImplementedError: TODO: …` in Lab 05	This is expected. Fill in the function the message names, then re-run

Related

- IC101: Launch noVNC
- Getting started