lockin: user's manual

A four-demodulator lock-in amplifier with oscillators, control loops and phase-locked loops, on the Red Pitaya STEMlab 125-14

Rok Žitko
Jozef Stefan Institute, Ljubljana, Slovenia
rok.zitko@ijs.si

Red Pitaya `lockin` design, version 1.0 (build 0x2ae6d985)
October 27, 2026

Abstract

`lockin` turns the Red Pitaya into a dual-input lock-in amplifier with four demodulators, four oscillators, two outputs and four control loops. Each demodulator detects an input's component at a harmonic or a sideband of the oscillators with a sine or square reference, filtered by up to eight RC sections; readings are volts at the connectors and phases referred to the OUT1 connector, through the board's calibration. PID controllers inside the FPGA lock oscillators to inputs (phase-locked loops), hold amplitudes and steer offsets. A data stream, a triggered capture and a frequency counter deliver the data; software measures averages, sweeps and resonance fits, spectra, the noise around a reference, the Allan deviation, and chooses time constants from the signal; a second port speaks a subset of the SR860's commands. This manual is for the lab: the first measurement, every setting and how it behaves, readings and measurements, loops, the software, calibration, applications, specifications and the command reference. Figures marked "measured" were recorded on the board used here, in loopback; the others are computed. How the instrument works, with derivations, is in the theory manual shared with `feedback` (`theory_manual.pdf`).

Contents

1 What lockin is **3**
1.1 What makes it different . . . 3
1.2 How a measurement is made, in one paragraph . . . 4
1.3 The documents . . . 4

2 Before you connect equipment **4**

3 Getting started **5**
3.1 The board and the service . . . 5
3.2 A first measurement . . . 5
3.3 The control panel . . . 6
3.4 Presets . . . 6

4 The settings **6**
4.1 How a configuration works . . . 6
4.2 Oscillators . . . 7
4.3 Outputs . . . 8
4.4 Inputs . . . 8
4.5 Demodulators . . . 9
4.6 PIDs . . . 9

4.7 The stream, units, and the rest . . . 10
4.8 Dither . . . 10

5 Readings **11**

6 Measurements **11**
6.1 Auto-phase . . . 11
6.2 Choosing a time constant from the signal . . . 11
6.3 Sweeps and resonance fits . . . 11
6.4 Captures . . . 12
6.5 The frequency counter . . . 12
6.6 Spectra of an input . . . 13
6.7 The spectrum around the reference . . . 13
6.8 The noise density . . . 14
6.9 Harmonics . . . 14

7 The data stream **15**
7.1 Long-term stability . . . 15

8 Control loops **16**
8.1 Gains, limits and signs . . . 16
8.2 Amplitude loop . . . 16
8.3 Offset loop and monitors . . . 17
8.4 Phase-locked loops . . . 17
8.5 Following a resonance . . . 18

9 Software **18**
9.1 Python on the board . . . 18
9.2 Python on a PC . . . 19
9.3 TCP commands . . . 19
9.4 The command line . . . 19
9.5 The SR86x port . . . 19

10 Calibration **20**
10.1 The board calibration . . . 20
10.2 Frequency response and delay . . . 20

11 Applications **21**
11.1 A weak signal in noise . . . 21
11.2 Characterizing a resonator . . . 21
11.3 Harmonics: modulation spectroscopy . . . 22
11.4 Two tones: intermodulation . . . 22
11.5 A bridge or a differential signal . . . 22
11.6 Locking to an external reference . . . 23
11.7 Recording for hours . . . 23

12 Specifications **23**

A Command reference **24**
A.1 lockin's TCP commands (port 5025) . . . 24
A.2 Common commands . . . 25
A.3 Python . . . 26

B The data stream's rows **26**

1 What lockin is

`lockin` is an FPGA design for the Red Pitaya STEMlab 125-14 together with the software that controls it. With it the board becomes a dual-input, four-demodulator digital lock-in amplifier with its own oscillators, control loops and phase-locked loops, in the style of a benchtop instrument:

Four oscillators
: from 0 to 62.5 MHz in steps of 0.44 μHz, each with a phase; frequency changes are phase-continuous, or *coherent*: every phase relative to one zero time, so that two oscillators at one frequency keep the phase difference you set and returning to a frequency restores its phase.

Two outputs
: OUT1 and OUT2, each the sum of the four oscillators' sines with its own amplitudes, a DC offset, an optional monitor of any reading, optional dither, and voltage limits that the FPGA enforces on every sample (and that can be locked).

Four demodulators
: on IN1, IN2 or IN1 − IN2, each mixing the input at the full sample rate with a reference at a harmonic m of an oscillator (or a sideband $mf_A \pm f_B$ of two), a sine or a square wave, with its own phase.

Low-pass filters
: of 1 to 8 RC sections (6 to 48 dB/octave) with any time constant from 0.5 μs to 10^6 s, on X and Y of each demodulator.

Readings
: X, Y, R, θ, the unwrapped phase and the frequency deviation, in volts at the connectors (rms or peak) and degrees referred to the OUT1 connector, all of one frame.

Four PID controllers
: at the filter rate (976.5625 kHz), from any reading to an oscillator's frequency or phase, an output's amplitude or its offset: phase-locked loops, amplitude and offset loops.

Data
: a continuous stream of X and Y, timestamped by frame; a four-channel capture of signals and readings with triggers; a frequency counter.

Software measurements
: averaged readings, auto-phase, sweeps and resonance fits, spectra of the inputs, a zoom spectrum and the noise density around a reference, a time constant chosen from the signal, the Allan deviation of long recordings, the inputs' delay calibration.

An SR86x-compatible port
: programs written for a Stanford Research SR860 or SR865 can drive demodulator 1, oscillator 1 and OUT1 with a subset of that instrument's commands.

1.1 What makes it different

- **Real volts and phases at the connectors.** Readings are volts at the SMA connectors through the board calibration shared by every design, and phases are referred to the OUT1 connector: a sine at IN1 in phase with OUT1's sine reads $\theta = 0$ (section 10).
- **Exact arithmetic.** The references are exact to the 48-bit phase, the filters' time constants are realized within 2×10^{-5}, and the whole signal path is computed by integer arithmetic that a Python model reproduces bit for bit: the FPGA's simulation benches compare every sample against it.
- **Everything of one frame.** A reading is one frame's X, Y, R, θ, PID outputs and oscillator frequencies together; the stream and the capture carry exactly the values a reading would show.

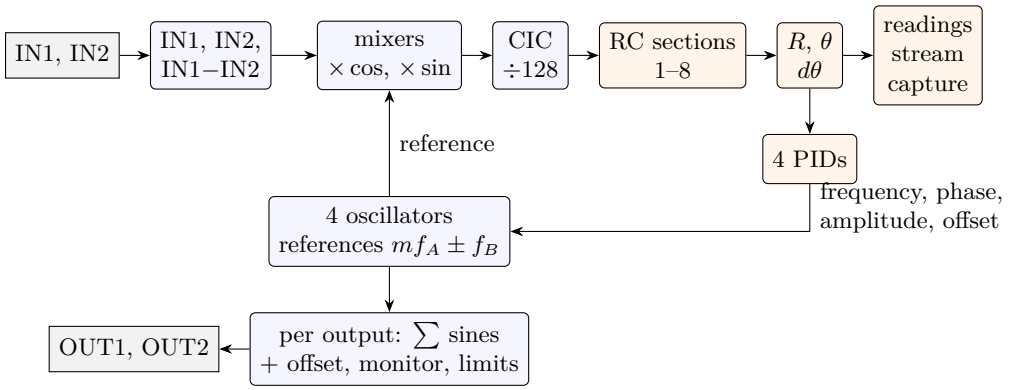


Figure 1: lockin's signal path. Blue: every sample (125 MSa/s); orange: every frame of 128 samples (976.5625 kHz).

- **Loops inside the instrument.** PID controllers act on the oscillators and outputs within four microseconds of a measurement, without a computer in the loop.

1.2 How a measurement is made, in one paragraph

A demodulator multiplies every sample of its input by the cosine and the sine of its reference phase. The products' slow part is the input's component at the reference frequency; the CIC filter averages the products over 256 samples into one value per frame of 128 samples, and the RC sections smooth these further with the time constant you set. X and Y are the result, scaled to volts; $R = \sqrt{X^2 + Y^2}$ and $\theta = \mathrm{atan2}(Y, X)$ follow. A sine $V \sin(2\pi f t + \theta)$ at the input reads $X = V \cos\theta$, $Y = V \sin\theta$ (rms volts by default). The *theory manual* (`lockin/docs/manual/theory_manual.pdf`, shared with `feedback`) derives all of this, with its noise, its resolution and its timing.

1.3 The documents

document	contents
this manual	operating lockin: settings, readings, measurements, loops, software, calibration, applications, specifications, the command reference
`docs/manual/theory_manual.pdf`	how lockin and feedback work, with derivations and computed figures
`docs/PROTOCOL.md`	every setting, method and command in brief
`RECIPES.md`	worked examples
`docs/THEORY.md`, `docs/REGISTERS.md`	the FPGA's internals and registers
`docs/VERIFICATION.md`	dated test records
`../common/CALIBRATION.md`	the board calibration

Figures marked *measured* were recorded on the board used throughout, through its own inputs in loopback (OUT1 → IN1, OUT2 → IN2, 50 Ω terminators, LV jumpers; dated in the captions); the others are computed with the design's model.

2 Before you connect equipment

Read the repository's section *Connecting to lab equipment: read first* (the root `README.md`). For lockin in particular:

- **Loading lockin leaves the outputs quiet**: both outputs off at 0 in converter units (a few millivolts from 0 V), the PIDs and the stream off. After a reboot, the service restores the last configuration.
- **PIDs act on the outputs.** A PID with an amplitude or offset target moves an output up to its `limits` (in the target's units). Set them for your load, and set the output's own voltage limits (`outputs.<o>.limits`): the FPGA clamps every sample to them, after all sums and

PID terms. `protect(o)` locks them until the FPGA is reloaded: then no configuration, preset, safe state or calibration can widen them.

- **Load.** Amplitudes are set into the load the board was calibrated with (50 Ω terminators at the inputs in the calibration's loopback); into 1 MΩ an output gives about twice the set voltage.
- **Inputs.** The LV jumpers give about ±1 V, the HV jumpers about ±20 V; tell the software which is set (`RANGE IN1 LV`), since the readings' scale depends on it. A clipped input invalidates every measurement on it: it is flagged in the readings, the stream and `hardware()`.
- **The trigger pin** DIO3_P on the extension connector is a 3.3 V LVCMOS input with a pull-down, not 5 V tolerant.
- **Safe state.** `SAFE` (Python `lockin.safe()`) switches both outputs off (0 V, or the nearest protected limit) and stops every PID and the stream.

3 Getting started

3.1 The board and the service

The board is set up as described in `common/SETUP.md` and reachable on the network (here at 192.168.0.149). A control service, `rp-overlay`, runs on it: it loads the last active design at boot, restores its configuration, and answers commands on TCP port 5025 (and, for lockin, the SR86x-compatible port 23). Everything below goes through it, or through the Python driver on the board itself. Switch the board to lockin with

```
python3 -m rpfpga.cli --host 192.168.0.149 --load lockin      # from a PC (common/python on
    ↪ PYTHONPATH)
echo "LOAD lockin" | nc 192.168.0.149 5025                     # or with netcat
```

Loading reprograms the FPGA with every setting off; the board's calibration and input-range settings are kept (they belong to the board, not to a design).

3.2 A first measurement

Cable OUT1 to IN1 with a 50 Ω terminator at IN1, LV jumpers. The preset `loopback` puts a 10 kHz sine of 0.2 V (peak) on OUT1 from oscillator 1 and sets demodulator 1 on IN1 at that frequency, with a 10 ms time constant and 24 dB/octave:

```
lockin.load_preset("loopback")
d = lockin.readings()["demodulators"]["1"]
print(d["x"], d["y"], d["r"], d["phase"])          # V rms, V rms, V rms, degrees
analysis.auto_phase(lockin, 1)                      # turn the reference: the signal on +X
```

R reads $0.2/\sqrt{2} = 0.141$ V rms times the board's analog loopback response at 10 kHz, and θ the phase of IN1's sine against OUT1's (near zero once the inputs' delay is calibrated). `auto_phase` turns demodulator 1's reference phase by the measured θ, so that the signal lies on $+X$ and Y reads zero. Figure 2 shows the same measurement on the board: the input and output captured, and two seconds of X and Y from the data stream after the auto-phase.

The same from a PC, through the service (standard library only):

```
import sys; sys.path[:0] = ["common/python", "lockin/control"]   # from the repository's root
from lockin import LockinClient
with LockinClient("192.168.0.149") as li:
    li.load_preset("loopback")
    print(li.readings()["demodulators"]["1"])
    print(li.auto_phase(1))
```

and from anything that can open a TCP connection (each reply is one line, `OK <value>` or `ERR <message>`):

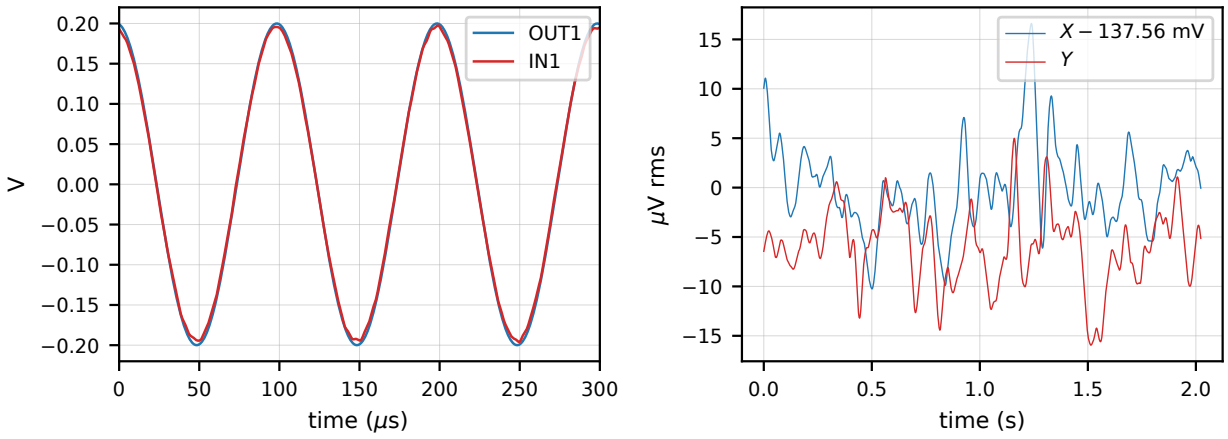


Figure 2: The `loopback` preset on the board (measured, 2026-10-27). Left: OUT1 and IN1 captured at 1.95 MSa/s. Right: two seconds of X and Y at 1000 rows/s after `auto_phase`: X = 137.559 mV rms, Y =-6.0 μV, with 4.37 μV rms of scatter in X (ENBW 7.8 Hz).

```
$ nc 192.168.0.149 5025
PRESET loopback
R? 1
AUTOPHASE 1
READ?
```

3.3 The control panel

On the board, the Jupyter notebook `lockin/lockin.ipynb` (at `http://192.168.0.149:9090`, password `xilinx`) opens a control panel with four tabs: **Settings** (oscillators, demodulators, outputs, PIDs, the stream: edit, then one **Apply**), **Meters** (the readings, live), **Presets** and **Calibration**. The panel's buttons **Safe**, **Zero time** and **Refresh** act at once. The same panel runs on a PC in Jupyter with the client: `make_gui(LockinClient(host))`.

3.4 Presets

name	settings
`off`	everything off, 0 V (the power-up state)
`loopback`	OUT1: 10 kHz, 0.2 V; demodulator 1 on IN1 at 10 kHz, τ = 10 ms, 24 dB/oct
`harmonics`	OUT1: 10 kHz, 0.2 V; demodulators 1–4 on IN1 at harmonics 1–4
`pll`	PID 1 locks oscillator 2 to a signal near 10 kHz on IN2 (100 Hz bandwidth, ±500 Hz, held below 10 mV); OUT2 repeats oscillator 2 at 0.1 V

Your own configurations can be saved under a name and loaded later (`save_preset(name)`, `PRESET SAVE <name>`); they live on the board's SD card.

4 The settings

4.1 How a configuration works

The whole instrument is described by one configuration, a JSON-compatible dictionary. `config()` returns it; `apply(patch)` merges a partial one into it and returns the *applied* values: the realized ones, which can differ from the requested ones (a frequency on the 0.44 μHz grid, a time constant or a gain quantized by the hardware). Units are numbered from 1 in keys and references.

```
applied = lockin.apply({"oscillators": {"1": {"frequency": 32768.0}},
                        "demodulators": {"1": {"time_constant": 3e-3, "order": 4}}})
print(applied["oscillators"]["1"]["frequency"])      # 32768.0000000xx: the realized frequency
print(lockin.config()["demodulators"]["1"])
```

- **Atomic.** An apply is validated completely before anything is written: a refused setting changes nothing (`ValueError`, `ERR <message>` over TCP). The FPGA then makes all the changes live together on one frame boundary, so that, for example, a new frequency and a new phase of two oscillators and a new filter of a demodulator act from the same sample.
- **Shared.** The service, the notebook, scripts on the board and clients on a PC share one configuration and a board-wide lock: an edit from one does not undo another's unrelated settings. A long measurement (a sweep, a capture, an average) is cancelled by another client's edit, so that it never reports a result for settings that changed under it (`ERR CANCELLED`).
- **Remembered.** The service keeps the configuration and restores it after a reboot.
- **Units.** Everything is physical: volts at the SMA connectors, hertz, degrees, seconds. Output amplitudes are peak volts; X, Y, R are rms volts, or peak volts with `units peak`.

4.2 Oscillators

key	default	meaning
`oscillators.k.frequency`	10^6	Hz, 0 to 62.5 MHz, in steps of $f_s/2^{48} = 0.44\,\mu$Hz
`oscillators.k.phase`	0	degrees: an offset of the oscillator's phase, which its sines on the outputs and every reference it drives carry
`phase_mode`	`continuous`	or `coherent`
`t0`	0	the zero time (a sample index) of coherent phases

Phase modes. In `continuous` mode a frequency change continues from the phase the oscillator has: no phase jump, as in most signal generators. In `coherent` mode every oscillator's phase is $360°\,f(t - t_0) +$ phase, relative to one zero time t_0: two oscillators at the same frequency keep the phase difference you set, and returning to a frequency restores its phase, whatever happened in between. `zero_time()` (`ZERO`) sets t_0 to now. A coherent apply takes effect about 2 ms after the request (the time the driver needs to prepare the phases).

```
lockin.apply({"phase_mode": "coherent",
              "oscillators": {"1": {"frequency": 1e6, "phase": 0.0}, "2": {"frequency": 1e6, "phase":
    ↪ 90.0}},
              "outputs": {"1": {"enabled": True, "amplitudes": {"1": 0.2}},
                          "2": {"enabled": True, "amplitudes": {"2": 0.2}}}})
```

4.3 Outputs

Each output is the sum of the four oscillators' sines, each with its own amplitude, plus an offset and a monitor, clamped to the output's limits:

$$\mathrm{OUT}o = \mathrm{clamp}\Big(\mathrm{offset} + \mathrm{monitor} + \sum_{k=1}^{4} A_{o,k} \sin \phi_k + \text{PID terms},\ \mathrm{limits}\Big). \tag{1}$$

key	default	meaning
`outputs.o.enabled`	false	off: 0 V (amplitudes and offset ignored)
`outputs.o.amplitudes`	0 each	V peak of each oscillator's sine on this output (may be negative: the sine inverted)
`outputs.o.offset`	0	V
`outputs.o.dither`	false	triangular dither of ±1 DAC step (section 4.8)
`outputs.o.limits`	$[-1.2, 1.2]$	V: every sample is clamped; clipped samples are counted
`outputs.o.protect`	false	read only: the limits are locked until the FPGA is reloaded (`protect(o)`)
`outputs.o.monitor`	off	a reading as a voltage: `source` (`x`, `y`, `r`, `phase`, `frequency` of demodulator `index`, `pid` output or `error` of PID `index`), `scale` in V per unit

Amplitudes are corrected for the DAC's zero-order-hold droop $\mathrm{sinc}(f/f_\mathrm{s})$ and, if stored, for the calibration's output response, so that the set amplitude is the one at the connector at every frequency. Keep frequencies below about 50 MHz; above, the DAC's images near 125 MHz $- f$ grow and the analog filters fall off. Add a low-pass filter after the output to remove the images when they matter.

Limits act on every sample after all sums, PID terms and the monitor: a loop that runs away cannot exceed them. The clipped samples are counted (`hardware()`) and flagged in the stream's rows.

Monitors. An output can carry a reading as a DC voltage, updated once per frame: for example `{"source": "r", "index": 1, "scale": 1000}` puts demodulator 1's R on the output at 1 V per mV. The scale is volts per volt, per degree, per hertz, or per unit of the PID's target.

4.4 Inputs

key	default	meaning
`inputs.in1.offset`	0	V subtracted from the input before the demodulators
`inputs.in1.ac`	false	digital AC coupling: the input's running DC estimate subtracted from every sample
`inputs.in1.ac_corner`	10	its corner frequency (Hz), 10^{-6} to 10^{5}

A DC level at the input does no harm to a demodulator far from DC (it lands at the reference frequency after mixing, where the filters remove it), but a large one uses up the converter's range: AC coupling or an offset removes it before the mixers. The input ranges (LV, HV) are the board's jumpers, declared to the software with `set_input_range` or `RANGE`.

4.5 Demodulators

key	default	meaning
`demodulators.d.input`	`in1`	`in1`, `in2` or `diff` (IN1 − IN2)
`demodulators.d.oscillator`	d	the reference's oscillator A (1–4)
`demodulators.d.harmonic`	1	$m = 0 \dots 1023$: the reference is mf_A (plus the sideband)
`demodulators.d.sideband`	none	nf_B with oscillator B, `sign` −1, 0 or +1
`demodulators.d.phase`	0	degrees subtracted from θ (the reference's phase)
`demodulators.d.reference`	`sine`	or `square` (±1; readings scaled to the fundamental)
`demodulators.d.time_constant`	10^{-3}	τ of each RC section, 0.5 µs to 10^6 s
`demodulators.d.order`	4	sections 0 to 8 (6 dB/octave each; 0: the CIC's output alone)

The reference. A demodulator detects the input's component at its reference frequency mf_A+nf_B: $m = 1$, $n = 0$ is the ordinary lock-in; $m > 1$ detects a harmonic (the second harmonic of a modulated absorption, a nonlinearity's distortion); $n = \pm 1$ an intermodulation product of two drive frequencies; $m = 0$ with $n = 0$ the input's DC level (through the same filter). The reference is exact: its phase is computed from the oscillators' 48-bit phases, so a harmonic or sideband never drifts against its oscillators.

A square reference multiplies by ±1 instead of a sine, as an analog switching lock-in does. It also detects the odd harmonics of the input (1/3 of the third, 1/5 of the fifth, ...); the readings are scaled so that a sine at the fundamental reads its true amplitude. Prefer the sine reference unless you need to reproduce a square-wave lock-in.

The filter. The time constant and the order set the measurement's bandwidth and its speed: the equivalent noise bandwidth (ENBW) sets the noise in X and Y, the settling time how long to wait after a change. `status()` reports both for every demodulator, with its reference frequency:

```
lockin.apply({"demodulators": {"1": {"time_constant": 0.1, "order": 4}}})
s = lockin.status()["demodulators"]["1"]
print(s["frequency"], s["enbw"], s["settle"])        # Hz, Hz, s (to 1e-4 of a step)
```

order n	1	2	3	4	5	6	7	8
slope (dB/oct)	6	12	18	24	30	36	42	48
ENBW $\times\tau$	0.250	0.125	0.094	0.078	0.068	0.062	0.056	0.052
settling to 10^{-4} (τ)	9.3	12.0	14.0	16.0	18.0	19.8	21.5	23.0
3 dB corner $\times\tau$	0.159	0.102	0.081	0.069	0.061	0.056	0.051	0.048

Table 1: The filters of n equal RC sections (computed, for $\tau \gg 1\,\mu$s): the equivalent noise bandwidth $\Gamma(n-\frac{1}{2})/(4\sqrt{\pi}\,\Gamma(n)\,\tau)$, the time a step takes to come within 10^{-4}, and the half-power corner.

Higher orders reject interferers near the reference better and settle to a given precision faster for a given ENBW (figure 3); 4 is a good default. Without dither or a reason for speed, choose the time constant from the precision you need: the noise in X is $e_n\sqrt{\text{ENBW}}$ for an input noise density e_n, so `auto_tau` chooses τ from the measured noise and signal (section 6.2).

4.6 PIDs

Four PID controllers run at the frame rate. Each reads one demodulator's quantity and drives one target; the outputs of PIDs with the same target add.

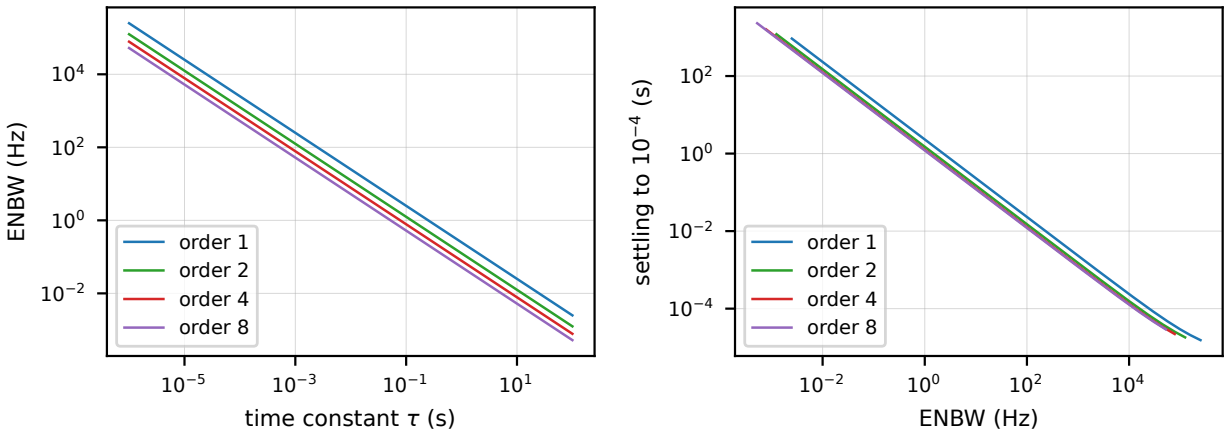


Figure 3: Choosing the filter (computed). Left: the equivalent noise bandwidth against the time constant for orders 1, 2, 4 and 8. Right: the settling time to 10^{-4} against the ENBW: for a given noise, a higher order settles faster.

key	default	meaning
`pids.p.enabled`	false	off: output 0, state cleared
`pids.p.input`	`x` of 1	`demodulator` and `quantity`: `x`, `y`, `r` (V), `phase` (degrees; the error wraps to ±180°), `frequency` (Hz: $d\theta/dt$), `phase_unwrapped` (degrees, every cycle since the FPGA's reset)
`pids.p.setpoint`	0	the input's units; error = setpoint − input
`pids.p.target`	none	`kind` `frequency` or `phase` of `oscillator`; `amplitude` of `oscillator` on `output`; `offset` of `output`
`pids.p.p`	0	target units per input unit
`pids.p.i`	0	target units per input unit per second
`pids.p.d`	0	target units per (input unit per second)
`pids.p.d_filter`	0	s: the D term's low-pass (2^k frames; 0 none)
`pids.p.limits`	$[-1, 1]$	the output's range in target units (anti-windup)
`pids.p.hold`	false	freeze the integrator, zero P and D
`pids.p.hold_below`	none	V: hold while R of the input's demodulator is below it
`pids.p.lock_tolerance`	0	input units: locked while $\lvert e\rvert \le$ it for `lock_time`
`pids.p.lock_time`	10^{-3}	s (2^k frames)

The applied configuration reports the realized gains (the hardware's gains are 18-bit mantissas with exponents: a gain beyond the range is refused with the limit in your units). Section 8 describes the loops in practice.

4.7 The stream, units, and the rest

key	default	meaning
`stream.rate`	0	rows per second, rounded to $976\,562.5/2^j$; 0 off
`stream.demodulators`	[1]	the demodulators whose X and Y each row carries
`units`	`rms`	or `peak`: the scale of X, Y, R

4.8 Dither

The DAC resolves 14 bits (0.12 mV on a ±1 V range). A small output amplitude or a slow offset change is then a staircase whose rounding error repeats with the signal: harmonics and dead zones. Dither (`dither`) adds a triangular random value of ±1 DAC step to every sample before the rounding, which turns the error into white noise of 0.5 DAC step rms (about 60 μV on a ±1 V range, spread over the 62.5 MHz band: about 8 nV/$\sqrt{\text{Hz}}$) and makes the output's average exact to a small fraction of a step. Use it for small amplitudes (below a few millivolts) and for slow offset loops.

5 Readings

`readings()` (`READ?`) returns one frame's results, all of the same frame:

field	meaning
`frame`	the frame's number since the FPGA was loaded (one per 1.024 μs)
`demodulators.d`	`x`, `y`, `r` (V rms or peak), `phase` (degrees, ±180), `phase_unwrapped` (degrees, all cycles), `frequency_offset` (Hz: $d\theta/dt$, the input's frequency against the reference)
`pids.p`	`output` (target units), `error` (input units), `integral`, `locked`, `saturated` (at a limit), `held`
`oscillators.k`	`frequency`: the live frequency, with the PIDs' corrections
`inputs_dc`	IN1's and IN2's DC estimates (V)

A reading takes about 3.4 ms through Python on the board (most of it the framework's checks), longer over the network. For many readings use the data stream (section 7); for a mean and a standard deviation, `average`:

```
a = analysis.average(lockin, ("1",), count=16)["1"]
print(a["x"], a["x_std"], a["r"], a["phase"])
```

`hardware()` (`HW?`) reports the counters and flags: clipped samples of each input and output since the last `clear_counts()`, the FIFO's level and dropped rows, the PIDs' flags, the protected outputs, the trigger pin's level and the sample counter.

Magnitude and phase in noise. At low signal-to-noise R is biased upward: it averages to $\sqrt{R_0^2+\sigma^2}$ rather than R_0, and reads $\sigma\sqrt{\pi/2}$ with no signal at all (σ: the noise in X and in Y). For weak signals average X and Y after an auto-phase and take X, or average X and Y separately and take their magnitude; the theory manual's chapter on noise has the details.

6 Measurements

6.1 Auto-phase

`auto_phase(device, d)` averages demodulator d's readings and sets its phase so that the signal lies on $+X$ ($Y = 0$); it returns the new phase. Afterwards X is the in-phase amplitude with all of the filter's noise rejection, and Y shows changes of the phase.

6.2 Choosing a time constant from the signal

`auto_tau(device, d, precision)` measures the input's noise density e_n near the reference and the signal R, and sets the shortest time constant whose noise $e_n\sqrt{\text{ENBW}}$ is at most `precision`$\cdot R$ (for the demodulator's order, or one you give). On the board, a 2 mV signal at 10 kHz with a precision of 10^{-3} got $\tau = 25.1$ ms (order 4, ENBW 3.11 Hz); its R then scattered by 0.88×10^{-3} of itself over 20 s (measured, 2026-10-27).

```
t = analysis.auto_tau(lockin, 1, precision=1e-3)     # applies the time constant found
print(t["time_constant"], t["order"], t["enbw"], t["noise_density"], t["r"])
```

6.3 Sweeps and resonance fits

`sweep(device, parameter, target, values)` steps one setting through a list of values, waits for the demodulators to settle after each step (their settling time to 10^{-4}, or `settle` seconds, times `settle_factor`), and averages `count` readings of each demodulator. The parameter is an oscillator's `frequency` or `phase` (target `{"oscillator": k}`), an output's `amplitude` of an oscillator (`{"output": o, "oscillator": k}`) or an output's `offset` (`{"output": o}`). Sweeps nest (`outer`: an outer sweep around each inner one), restore the swept settings afterwards, and on the board run as an acquisition: another client's edit cancels them. Over TCP:

```
SWEEP {"parameter": "frequency", "target": {"oscillator": 1}, "start": 1e3, "stop": 1e6,
       "points": 31, "scale": "log", "demodulators": [1, 2], "fit": false}
```

Figure 4 is such a sweep on the board: the analog loopback response OUT1 → IN1 and OUT2 → IN2 from 1 kHz to 60 MHz. The amplitude rises by about 7 % at a pole-zero near 21 kHz, rises slowly to about 10.0 % above its low-frequency value at 10 MHz, and falls off above 50 MHz; the phase, with the inputs' delays calibrated, stays within a few degrees of zero up to about 10 MHz. A calibration can store this response and correct for it (section 10).

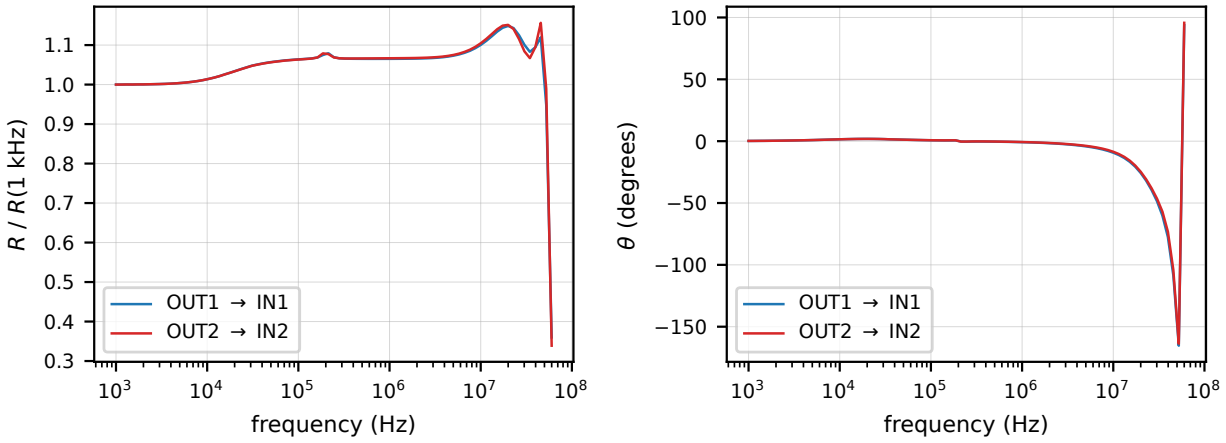


Figure 4: The board's analog loopback response, OUT1 → IN1 and OUT2 → IN2 through 50 cm cables and 50 Ω terminators (measured, 2026-10-27): R against the set amplitude (relative to its value at 1 kHz) and θ, from a frequency sweep of 81 points.

A resonance seen through the lock-in is a circle in the X–Y plane. `fit_resonance` fits a complex Lorentzian $a/(1+2\mathrm{i}Q(f-f_0)/f_0)+c$ to a frequency sweep's X and Y and returns f_0, Q, the FWHM, the amplitude and phase and the background; `SWEEP` with `fit` does it on the board. The theory manual shows the fit on a simulated tuning fork.

6.4 Captures

`capture(sources, rate, trigger, pretrigger, level)` records 8192 words of up to four sources at 125 MHz/2^k:

- full-rate sources `in1`, `in2`, `out1`, `out2`, `xin<d>` (demodulator d's input), each word the mean over 2^k samples, in volts;
- frame-rate sources `x<d>`, `y<d>`, `r<d>`, `phase<d>`, `frequency<d>`, `pid<p>`, `error<p>`: each word the reading of one frame (at rates up to 976.5625 kHz);
- triggers `immediate`, `rising` or `falling` (the DIO3_P pin), `in1_up`, `in1_down`, `in2_up`, `in2_down` (Schmitt levels `level` = (high, low) volts), `software`; `pretrigger` words before the trigger;
- `words`: an immediate capture stopped after its first words (a short record, quickly).

The result has the times `t` (seconds from the trigger), the realized `rate` and each source's values. Figure 5 shows a triggered capture.

6.5 The frequency counter

`count_frequency(channel, gate_s)` counts the rising edges of IN1 or IN2 through Schmitt levels (±`hysteresis` around `level`) in a gate of about `gate_s` (a power of two samples) and returns the frequency from the first and last edges' samples (a reciprocal count); `None` without two edges. Its

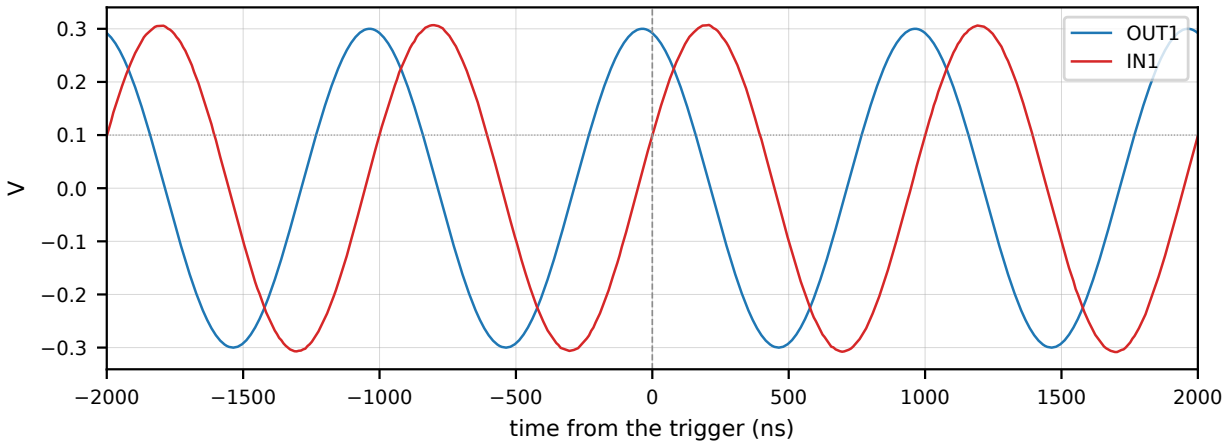


Figure 5: A capture at 125 MSa/s of OUT1 (a 1 MHz sine of 0.3 V) and IN1, triggered by IN1 rising through 0.1 V, with 1024 words before the trigger (measured, 2026-10-27). The input follows the captured output word 235 ns later: the core's output latency (14 samples, 112 ns) and the converters' loop delay (about 120 ns).

resolution is the 8 ns of the edges' timing over the gate, not one count in the gate: figure 6 shows the scatter of repeated counts of a 1.234568 MHz sine.

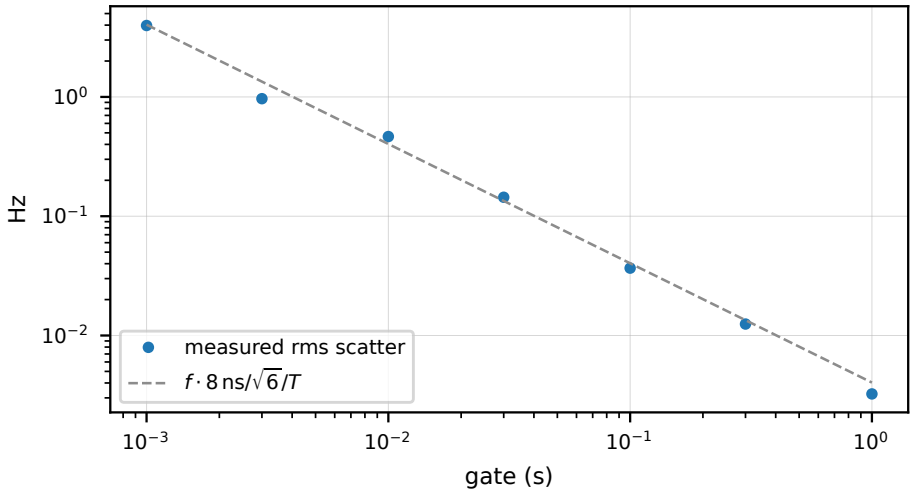


Figure 6: The counter on IN2 (OUT2: 1.234568 MHz, 0.2 V): the rms scatter of 8 counts per gate, against the timing limit $f \cdot (8\,\mathrm{ns}/\sqrt{6})/T_{\mathrm{gate}}$ (measured, 2026-10-27). The counter uses the board's own clock: its readings have no offset against the oscillators.

6.6 Spectra of an input

`spectrum(device, channel, averages, rate)` computes an input's spectrum from captures (8192 samples at $125\,\mathrm{MHz}/2^k$, a Blackman–Harris window, several captures power-averaged): per bin its power in dBV rms and its density in $\mathrm{V}/\sqrt{\mathrm{Hz}}$. A decimated capture is a mean over 2^k samples, which folds some of what lies above the decimated Nyquist frequency back into the band: trust the top part of a decimated spectrum less.

6.7 The spectrum around the reference

`zoom_spectrum(device, d, span, points, averages)` records $z = X + \mathrm{i}Y$ of demodulator d at a frame rate divided by a power of two and takes its spectrum. With rms outputs, the two-sided density of z at an offset δ equals the input's one-sided density at $f_{\mathrm{ref}} + \delta$, times the filter's response,

which the function divides out: the result is the input's spectrum around the reference, with a resolution set by the record's length, not by the reference frequency (a zoom FFT). Figure 7 shows a 1 mV tone 300 Hz above a 10 kHz reference.

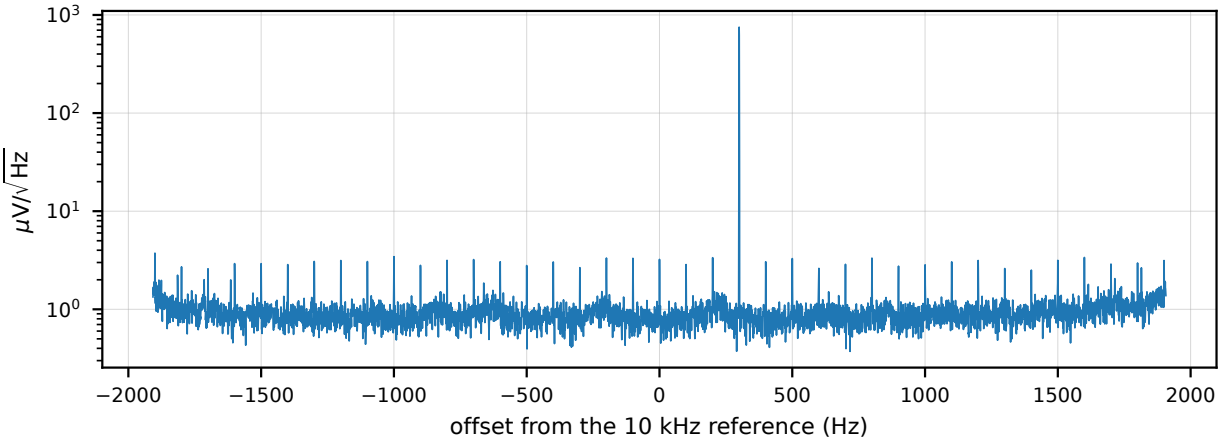


Figure 7: The zoom spectrum around 10 kHz (span 2 kHz, 8 averages): a 1 mV (peak) tone from OUT1 300 Hz above the reference reads 0.7257 mV rms, against 0.7472 mV rms from a fit to the captured input (a tone between two bins loses up to 0.8 dB in the window); the floor is IN1's noise density near 10 kHz, 0.90 $\mu V/\sqrt{Hz}$ (measured, 2026-10-27).

6.8 The noise density

`noise_density(device, d)` returns the input's noise density at the reference: the zoom spectrum's median within the filter's band, corrected for the median's bias, and, independently, from the variance of X and Y ($\sigma_X = e_n\sqrt{\mathrm{ENBW}}$). A tone at the reference is excluded (it would dominate the variance; the median ignores it). Figure 8 is the board's input noise from 100 Hz to 20 MHz with the outputs off.

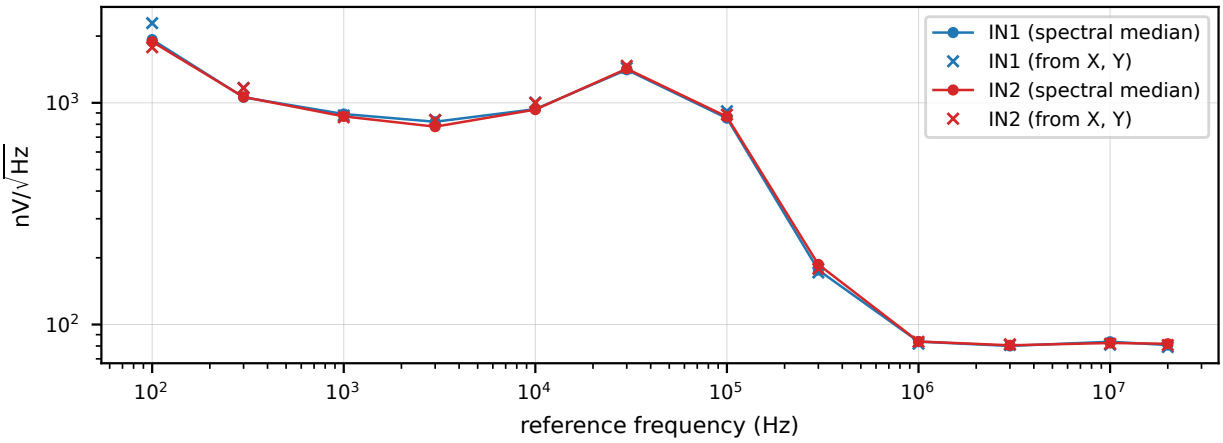


Figure 8: The input noise density of IN1 and IN2 (LV jumpers, terminated with 50 Ω through the loopback cables, outputs off), from `noise_density` at references from 100 Hz to 20 MHz (measured, 2026-10-27): 0.89 $\mu V/\sqrt{Hz}$ at 1 kHz, 0.85 $\mu V/\sqrt{Hz}$ at 100 kHz, 84 $nV/\sqrt{Hz}$ at 1 MHz. The rise toward low frequencies is the converters' flicker noise; the broad bump near 30 kHz is the board's (its source not identified).

6.9 Harmonics

With several demodulators at harmonics of one oscillator, a nonlinearity's distortion is measured at once. Figure 9 shows the board's own: a 10 kHz sine of 0.8 V through the converters' loopback,

harmonics 2 to 8 at most −64 dBc.

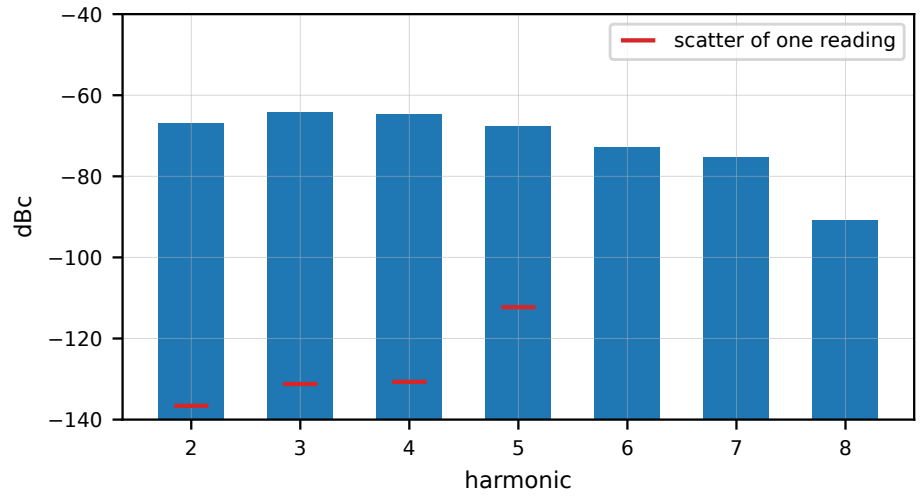


Figure 9: Harmonics of a 10 kHz, 0.8 V sine through the loopback OUT1 → IN1 (measured, 2026-10-27; demodulators at harmonics 1–4, then 5–8, $\tau = 0.1$ s, 24 dB/oct), relative to the fundamental; the bars' tops are the averages of 16 readings, the floor the readings' scatter.

7 The data stream

The stream delivers every 2^j-th frame's X and Y of chosen demodulators, without gaps, while it is on: `apply({"stream": {"rate": 1000, "demodulators": [1, 2]}})` (the rate is rounded to $976\,562.5/2^j$ rows/s). The board's service drains the FPGA's FIFO and sends the rows to every connected client (`DATA STREAM`); on a PC:

```
from lockin import DataStream, LockinClient
with LockinClient("192.168.0.149") as li:
    li.set_stream(1000, demodulators=(1, 2))
    with DataStream("192.168.0.149", interval=0.1) as s:
        for _ in range(10):
            rows, lost = s.read()          # [(frame, t_s, x1, y1, x2, y2, x3, y3, x4, y4, locked,
    ↪ clipped, apply, flags, units_peak)]
    rec = li.record(60, rate=976.5625, demodulators=(1,))    # 60 s: {"t", "x1", "y1", "rate", "lost
    ↪ ", "gaps"}
```

Every row carries its frame number and time, the PIDs' locked flags, the frame's clip flags, the apply count (so that a recording shows where the configuration changed) and a flag after a gap. The service sustains about 8000 rows/s on this board; faster streams drop rows at the FIFO, counted and flagged. `record(seconds, rate, demodulators)` collects the stream into columns.

7.1 Long-term stability

For drifts over minutes and hours, record the stream and take its Allan deviation (`allan_deviation(values, rate)`): the averaging time where it is smallest is the longest worth averaging, beyond it drifts dominate. Figure 10 is five minutes of a 1 MHz loopback.

`RunningStats()` keeps the count, mean, standard deviation and extremes of values as they arrive (`.add(values)`, `.summary()`), for recordings too long to keep.

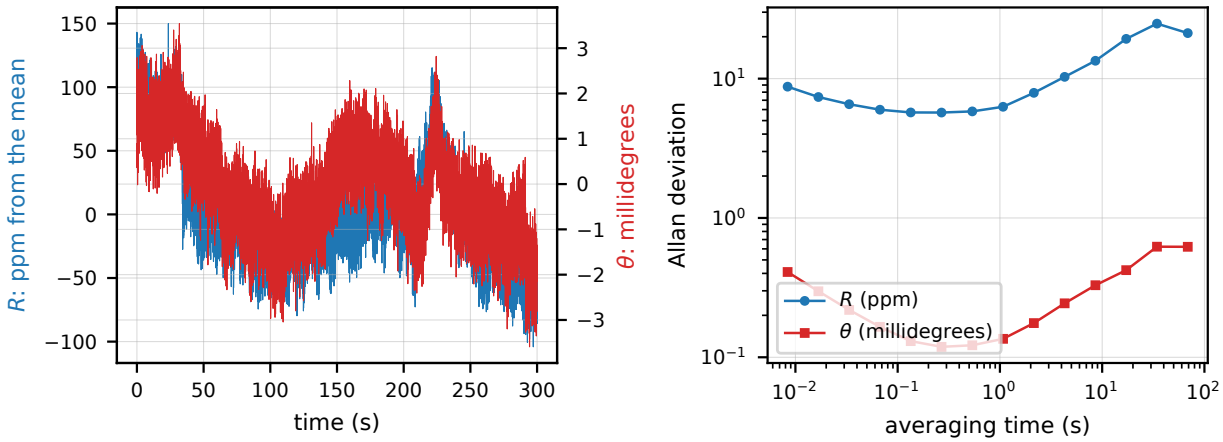


Figure 10: Five minutes of R and θ of a 1 MHz, 0.2 V loopback (OUT1 → IN1, τ = 1 ms, 24 dB/oct, 119 rows/s; measured, 2026-10-27). Left: R's relative deviation from its mean and θ against time. Right: their Allan deviations: R reaches 5.7 $\times 10^{-6}$ at 0.268 s.

8 Control loops

A PID closes a loop around the instrument's own measurement, inside the FPGA: it reads a demodulator's quantity once per frame and changes an oscillator's frequency or phase, an output's amplitude or its offset, four frames (4.1 μs) later. No computer is in the loop, so loops of kilohertz bandwidth run as steadily as slow ones.

8.1 Gains, limits and signs

The gains are physical: p in target units per input unit, i per second, d in seconds. A loop on a plant of static gain K (target to input) with an integral gain alone crosses over near $f_c = |Ki|/2\pi$; the demodulator's filter must be faster than that (its corner well above f_c), since the loop cannot be faster than the measurement it acts on. The theory manual's chapter on the slow PIDs gives the largest crossover each filter allows.

The sign of the gains depends on how the target moves the input: the error is setpoint − input, so a target that raises the input needs positive gains. A loop with the wrong sign runs to a limit: the `limits` (in the target's units) stop it there, and the integrator stops at the limit as well (anti-windup), so the loop recovers as soon as the error changes sign.

Hold, reset, lock. `hold` freezes the integrator and zeroes P and D; `hold_below` does so automatically while the demodulator's R is below a level (a lost signal: the loop keeps its last output rather than chasing noise). `reset_pid(p)` clears the integrator. A PID is *locked* while its error has stayed within `lock_tolerance` for `lock_time`; the flag is in the readings, the stream's rows and `hardware()`.

8.2 Amplitude loop

Hold a response's amplitude constant by the drive: PID input R of demodulator 1, target the amplitude of oscillator 1 on OUT1. Figure 11 shows such a loop on the loopback (the plant's gain K is the loopback's, 0.722 V rms of R per volt of peak amplitude): with $i = 1000\,\mathrm{s}^{-1}$ the setpoint steps settle exponentially with a time constant of 1.52 ms (a single pole at $1/(Ki)$: 1.39 ms predicted).

```
lockin.apply({"oscillators": {"1": {"frequency": 1953125.0}},
              "outputs": {"1": {"enabled": True, "amplitudes": {"1": 0.07}}},
              "demodulators": {"1": {"input": "in1", "oscillator": 1, "time_constant": 2e-5, "order":
    ↪  2}},
              "pids": {"1": {"enabled": True, "input": {"demodulator": 1, "quantity": "r"}, "
    ↪ setpoint": 0.05,
                             "target": {"kind": "amplitude", "oscillator": 1, "output": 1},
                             "i": 1000.0, "limits": [-0.07, 0.3]}}})
```

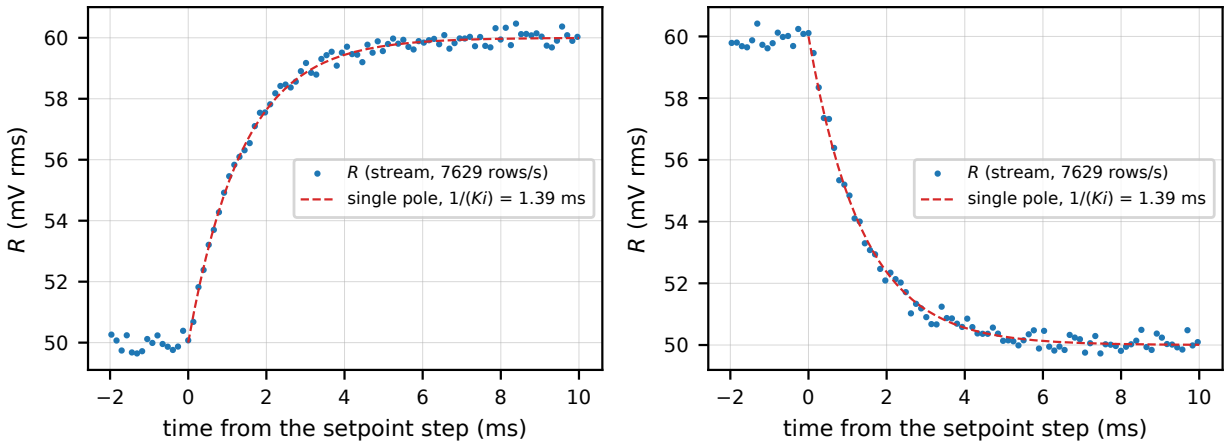


Figure 11: An amplitude loop on the loopback: demodulator 1's R at 100 kHz held by oscillator 1's amplitude on OUT1 ($i = 1000\,\mathrm{s}^{-1}$), its setpoint stepped from 50 to 60 mV and back, from the data stream at 7629 rows/s (measured, 2026-10-27). Dashed: the single pole e^{-tKi} with the plant's gain K measured from a reading.

8.3 Offset loop and monitors

An offset target steers an output's DC level: holding X of a dispersive signal at zero through a DC actuator, centring a resonance, or a slow side-of-fringe lock. The signs of p and i depend on the actuator. A monitor puts any reading on an output as a voltage, for an oscillograph, a chart recorder or another instrument's input (section 4.3).

8.4 Phase-locked loops

A PID from a demodulator's phase to its oscillator's frequency makes the oscillator follow the input: a PLL. `lock_pll` sets it up from its bandwidth:

```
lockin.apply({"oscillators": {"2": {"frequency": 976562.5}, "3": {"frequency": 976600.0}},
              "outputs": {"2": {"enabled": True, "amplitudes": {"2": 0.2}}},
              "demodulators": {"2": {"input": "in2", "oscillator": 3}}})
analysis.lock_pll(lockin, pid=1, bandwidth=1000.0, oscillator=3, demodulator=2)
print(lockin.readings()["oscillators"]["3"]["frequency"])
```

- **The loop:** a type-2 loop (P and I) crossing over at the bandwidth, with the demodulator's filter set to order 2 at eight times the bandwidth (`filter` false keeps your filter). Bandwidths up to 19 kHz are possible (the integral gain's range); 10 kHz keeps about 48° of phase margin, 1 kHz 60°.
- **Pull-in:** from up to 7–8 bandwidths away (3 at 10 kHz). `seed in1` or `in2` first measures the input's frequency with the counter and sets the oscillator there; `acquire` (Hz) first locks the frequency with the same PID on $d\theta/dt$ within that range and then switches to the phase, for inputs the counter cannot measure (harmonics, sidebands, IN1 − IN2, noisy signals).

- **Limits:** ±5 bandwidths around the oscillator's set frequency (`frequency_range`): inside the pull-in range, so that a loop that lost its input pulls back in when it returns.
- **Hold:** `hold_below` keeps the frequency while R is below a level.

The oscillator's live frequency (`readings()["oscillators"]`) is then the input's frequency, measured every frame; other demodulators referenced to this oscillator (or its harmonics) measure other inputs against the external signal, and an output can repeat it. Figure 12 shows a 100 Hz loop relocking after its oscillator's set frequency jumped by one and by three bandwidths.

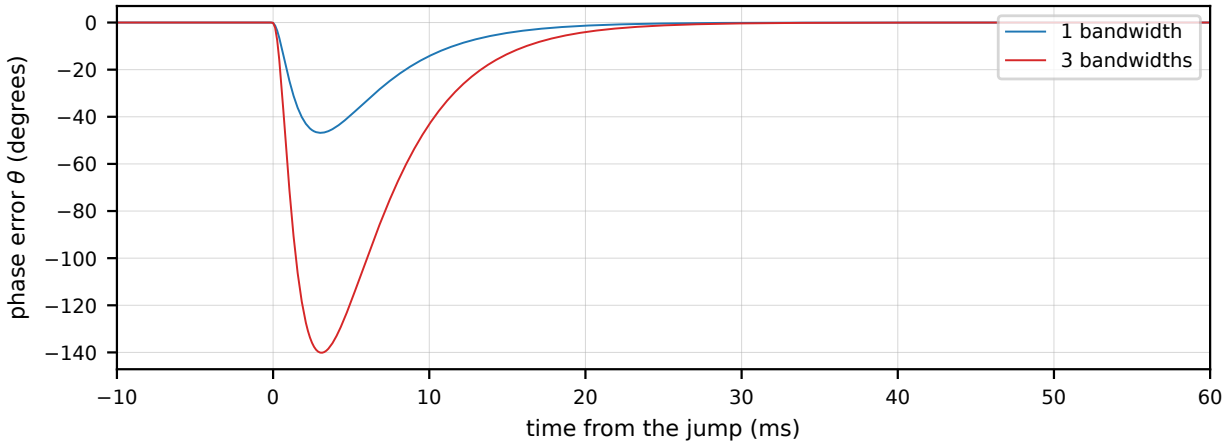


Figure 12: A 100 Hz PLL (PID 1, oscillator 3, demodulator 2 on IN2) locked to a 1 MHz sine from OUT2: its phase error after the oscillator's set frequency jumped by 100 Hz and 300 Hz (from the data stream; measured, 2026-10-27). The loop absorbs the jump in its integrator within 14.4 ms (one bandwidth) and 19.0 ms (three).

8.5 Following a resonance

After a sweep and a fit, a PID can keep an oscillator on a resonance: input the resonator's phase, setpoint the phase at f_0, target the oscillator's frequency. Near f_0 the phase falls by $360°Q/(\pi f_0)$ per hertz, so an integral gain $i = -2\pi$ bw/slope gives a loop of bandwidth bw; keep it well below the resonance's half width $f_0/2Q$ and the filter's bandwidth (`RECIPES.md`, "Follow the resonance").

9 Software

Four ways to control lockin share one configuration: the Python driver on the board, the Python client on a PC, plain TCP commands, and the command line. The notebook's control panel uses the driver (or the client). Appendix A lists every command.

9.1 Python on the board

```
import rpfpga                                    # first: it prepares the environment PYNQ needs
from pynq.overlays.lockin import DRIVER, analysis
lockin = DRIVER()                                # attaches to lockin, or loads it if another design
    ↪ runs
lockin.apply({"oscillators": {"1": {"frequency": 10e3}}})
```

The driver is the instrument: `config`, `apply`, `readings`, `status`, `hardware`, `capture`, `count_frequency`, the presets, the calibration, and the analysis functions taking it as their first argument. It never touches the FPGA unless the board runs lockin with the expected build (a register access to an unconfigured FPGA would hang the board): loading and attaching are checked against the design's identity and build.

9.2 Python on a PC

`LockinClient(host)` talks to the board's service with the standard library alone (put `common/python` and `lockin/control` on the path). It has the driver's methods, plus convenience ones (`set_frequency`, `set_output`, `set_demodulator`, `set_pid`, `set_stream`, `value`, ...); the long ones (`sweep`, `average`, `auto_phase`, `zoom_spectrum`, `noise_density`, `auto_tau`, `lock_pll`, `capture`) run on the board. `DataStream(host)` reads the data stream, `record()` collects it.

- **One request at a time per connection.** Use a second client (`client.clone()`) to read meters while another controls.
- **Acquisitions.** A long measurement claims the board's single acquisition slot: a second one fails at once (`Busy`), and another client's edit cancels it (`Cancelled`).
- **Reloading** the FPGA starts a new session: old handles raise `Stale`; reconnect.
- **Compare-and-set.** `snapshot()` returns the configuration with its revision; `apply_snapshot(patch, expected_revision=...)` refuses (`Conflict`) if someone changed the configuration in between.

9.3 TCP commands

The service listens on port 5025; one command per line, case-insensitive keywords, one reply line each: `OK <value>` (JSON for structures) or `ERR <message>`. The common commands (`*IDN?`, `CONFIG`, `PRESET`, `SAFE`, `CAL`, `RANGE`, ...) work for every design; lockin adds its own (appendix A).

```
FREQ 1 10000
AMP 1 1 0.2
OUT 1 ON
DEMOD 1 {"input": "in1", "oscillator": 1, "time_constant": 0.01, "order": 4}
R? 1
PLL 1 {"bandwidth": 100, "oscillator": 2, "demodulator": 2, "seed": "in2"}
SWEEP {"parameter": "frequency", "target": {"oscillator": 1}, "start": 1e3, "stop": 1e6, "points":
    ↪ 31, "scale": "log"}
SAFE
```

The data stream is a command of its own connection: `DATA STREAM [interval]` replies with a header (`OK {"fields": ..., "format": ..., "rate": ...}`) and then one line every interval, `DATA {"rows": n, "lost": ..., "data": base64}`, each row 15 little-endian float64 fields (appendix B). `STOP` ends it.

9.4 The command line

```
python3 -m rpfpga.cli --host 192.168.0.149 --osc 1=10e3 --amp 1=1,0.2 --demod 1=in1,0.01,4 --
    ↪ autophase 1
```

prints the description, every demodulator's readings, the active PIDs and the counters. Options: `-osc K=HZ[,DEG]`, `-demod D=INPUT[,TAU[,ORDER]]`, `-amp O=K,V`, `-offset1 V`, `-offset2 V`, `-units rms|peak`, `-pid P JSON`, `-autophase D`, and the common `-load`, `-preset`, `-safe`, `-range`, `-calibration`, `-json`.

9.5 The SR86x port

Programs written for a Stanford Research SR860 or SR865 lock-in amplifier can drive lockin on the service's second port, 23. The port maps a subset of the SR860's commands onto demodulator 1, oscillator 1 and OUT1, with the SR860's conventions: values in V rms, Hz and degrees, `SLVL` the sine's rms amplitude, time constants from the SR860's table, sensitivities, status bytes. For example with pymeasure:

```python
from pymeasure.instruments.srs import SR860
lia = SR860("TCPIP::192.168.0.149::23::SOCKET")      # line terminator: "\n"
lia.frequency = 10e3
lia.sine_voltage = 0.1                                # V rms on OUT1
lia.time_constant = 0.01
print(lia.x, lia.y, lia.magnitude, lia.theta)
```

SR860 command	lockin
`FREQ`, `FREQINT`, `FREQDET?`	oscillator 1's frequency; the detection frequency (harmonic $\times f$)
`PHAS`, `HARM`	demodulator 1's reference phase and harmonic (1–1023)
`SLVL`, `SOFF`	oscillator 1's rms amplitude on OUT1 (OUT1 switched on); OUT1's offset
`ISRC`, `ICPL`	the input: A (IN1), A−B, B (IN2); AC or DC coupling
`OFLT`, `OFSL`, `ENBW?`	the time constant (SR860's table, nearest), the slope (orders 1–4), the noise bandwidth
`SCAL`, `ASCL`, `ILVL?`	the sensitivity (for `ASCL` and the status bits only), auto-scale, the input level
`APHS`	auto-phase
`OUTP?`, `SNAP?`, `SNAPD?`	X, Y, R, θ and others, of one frame
`*IDN?`, `*RST`, `*CLS`, `*ESR?`, `*STB?`, `LIAS?`, …	identity, reset, the IEEE 488.2 status and the lock-in status word

Unknown commands and bad values set the standard event status bits and answer nothing, as an SR860 does; data capture, aux I/O, the display and dual reference are not provided. The full table is in `docs/PROTOCOL.md`.

10 Calibration

10.1 The board calibration

The converters have gain and offset errors of up to about 14 % and 15 mV on the board used here. One calibration, stored on the board and shared by every design, turns the converters' codes into volts at the connectors (`common/CALIBRATION.md`); lockin uses it for every amplitude, offset, limit and reading. lockin can also measure it:

```
CAL OFFSETS           # inputs terminated or shorted: the ADC offsets
CAL HOLD              # both outputs at a reference code: read OUT1, OUT2 with a DMM ...
CAL RELEASE
CAL LOOPBACK {"out1": 0.5631, "out2": 0.5582}    # OUT1 -> IN1, OUT2 -> IN2: the gains and offsets
```

In Python the same steps are `measure_input_offsets()`, `hold_reference()`, `release_reference()` and `calibrate(dmm)`; the numbers above are an example of DMM readings. The outputs change during these steps: disconnect equipment that must not see them. Without a calibration the nominal constants are used: about 14 % off in amplitude on this board, though ratios and loopback measurements stay right.

The input ranges. The LV/HV jumpers are read by nothing: declare them (`set_input_range("in1", "lv")`, `RANGE IN1 LV`); the calibration keeps constants for each setting.

10.2 Frequency response and delay

The board calibration is a DC calibration. Two optional fields per input and range refine lockin's AC readings:

- **the loop delay** (`delay_s`): the time from OUT1's DAC port through connectors joined directly to the input's ADC port, about 120 ns. lockin adds its own latencies and refers every

phase to the OUT1 connector with it: a sine at IN1 in phase with OUT1's sine reads $\theta = 0$. Without it, θ includes the converters' delay (43° at 1 MHz).

- **the response** (`response`): the input's amplitude gain against frequency, `[[Hz, gain], ...]`; readings are divided by it.

Measure the delay in loopback, with cables of known delay (here 0.5 ns each):

```
analysis.measure_input_delay(lockin, "in1", output=1, cable_delay_s=0.5e-9, store=True)
analysis.measure_input_delay(lockin, "in2", output=2, cable_delay_s=0.5e-9, store=True)
```

It drives the output into the input at five multiples of the frame rate (where the mixers' $2f$ products vanish), fits the phase's slope against frequency, and stores the delay the slope shows (OUT2's calibrated skew against OUT1 is taken into account). The response needs a calibrated source, since a loopback measures the output and the input together (figure 4); store it with `set_input_response("in1", points)` or `CAL RESPONSE IN1 [[...]]`, and the outputs' with `set_output_response` (`CAL RESPONSE OUT1 ...`).

What is calibrated, what is not

DC gains and offsets (after `CAL LOOPBACK`, with a DMM for absolute accuracy); the inputs' loop delays (after `measure_input_delay`). Not calibrated unless you store tables: the analog front ends' frequency responses (a few percent between 20 kHz and 20 MHz on this board) and the IN1/IN2 match at AC. Drift with temperature is not compensated: recalibrate after warm-up and after changing jumpers or cables.

11 Applications

Worked examples from the lab. `lockin` is the driver on the board; with `LockinClient` on a PC the same calls work (`li.sweep(...)`, `li.auto_phase(1)`, ..., run on the board). `RECIPES.md` has more.

11.1 A weak signal in noise

A sample excited from OUT1 at 1.234 kHz responds with microvolts on IN1. Choose the time constant from the precision you need, put the signal on X, and average:

```
lockin.apply({"oscillators": {"1": {"frequency": 1234.0}},
              "outputs": {"1": {"enabled": True, "amplitudes": {"1": 0.5}}},
              "demodulators": {"1": {"input": "in1", "oscillator": 1, "time_constant": 0.1, "order":
    ↪ 4}}})
print(lockin.status()["demodulators"]["1"])          # frequency, ENBW (Hz), settling time (s)
analysis.auto_phase(lockin, 1)                       # the signal on +X
r = analysis.average(lockin, ("1",), count=100)["1"]
print(r["x"], r["x_std"])
```

With $\tau = 0.1$ s and 24 dB/oct the ENBW is 0.78 Hz; with the input's noise of $0.89\,\mu\text{V}/\sqrt{\text{Hz}}$ near 1 kHz on this board (figure 8), X scatters by about 0.79 μV rms per reading. Readings closer together than the settling time are correlated: the standard error of the mean of n readings is $\sigma_X/\sqrt{n}$ only for readings a few time constants apart. A large DC level at the input is harmless to the demodulation but uses up the converter's range: remove it with AC coupling (`{"inputs": {"in1": {"ac": True, "ac_corner": 1.0}}}`).

Two refinements: the input's noise is lower above a few kilohertz (move the modulation up if the experiment allows), and `noise_density` measures the actual noise at your reference, with your cabling and interference.

11.2 Characterizing a resonator

A quartz tuning fork, a cantilever's piezo, an LC circuit between OUT1 and IN1: sweep, fit, then follow the resonance.

```
values = analysis.sweep_values(32.6e3, 32.9e3, 151)
lockin.apply({"demodulators": {"1": {"time_constant": 10e-3, "order": 4}}})
result = analysis.sweep(lockin, "frequency", {"oscillator": 1}, values, count=4)
d = result["demodulators"]["1"]
fit = analysis.fit_resonance(result["values"], d["x"], d["y"])
print("f0 %.3f Hz, Q %.0f, FWHM %.3f Hz" % (fit["f0"], fit["q"], fit["fwhm"]))
```

The sweep waits each point's settling time, set by the filter; a sharp resonance needs its own ring-down time $\approx Q/(\pi f_0)$ as well, which `settle` or `settle_factor` provides. Then a PID on the resonator's phase keeps oscillator 1 on the resonance (section 8):

```
import math
slope = 360 * fit["q"] / (math.pi * fit["f0"])          # degrees per Hz at f0
bw = 1.0                                                # Hz: well below f0 / (2 Q)
lockin.apply({"oscillators": {"1": {"frequency": fit["f0"]}},
              "pids": {"1": {"enabled": True, "input": {"demodulator": 1, "quantity": "phase"},
                             "setpoint": fit["phase"], "i": -2 * math.pi * bw / slope,
                             "target": {"kind": "frequency", "oscillator": 1, "output": 1},
                             "limits": [-50.0, 50.0]}}})
```

The oscillator's live frequency then tracks f_0 (a mass sensor, a temperature sensor, a force-gradient microscope's frequency shift), and the drive amplitude can be held by a second PID on R (section 8) to measure the dissipation as the drive needed.

11.3 Harmonics: modulation spectroscopy

Wavelength-modulation spectroscopy dithers a laser at f and detects the photodiode's signal at $2f$, whose zero crossing marks the line's centre. Demodulators at harmonics 1 and 2 of one oscillator give both at once:

```
lockin.apply({"oscillators": {"1": {"frequency": 50e3}},
              "outputs": {"1": {"enabled": True, "amplitudes": {"1": 0.1}}},
              "demodulators": {"1": {"input": "in1", "oscillator": 1, "harmonic": 1, "time_constant":
    ↪ 1e-3},
                               "2": {"input": "in1", "oscillator": 1, "harmonic": 2, "time_constant":
    ↪ 1e-3}}})
```

The harmonic's reference is exact, so its phase is fixed against the modulation. Set each demodulator's phase by `auto_phase` on a strong line, then keep it.

11.4 Two tones: intermodulation

A mixer, a nonlinear sample, or a frequency converter driven by two tones f_A and f_B produces products $mf_A \pm f_B$. A demodulator with a sideband detects one directly:

```
lockin.apply({"oscillators": {"1": {"frequency": 100e3}, "2": {"frequency": 37e3}},
              "outputs": {"1": {"enabled": True, "amplitudes": {"1": 0.2}},
                          "2": {"enabled": True, "amplitudes": {"2": 0.2}}},
              "demodulators": {"3": {"input": "in1", "oscillator": 1, "harmonic": 1,
                                     "sideband": {"oscillator": 2, "sign": 1}}}})
```

11.5 A bridge or a differential signal

Demodulating IN1 − IN2 (`input diff`) subtracts the two inputs sample by sample before the mixers: a bridge's two arms, a balanced detector's two photodiodes, a reference arm subtracted from a signal arm. The subtraction happens on the converter codes, so the common-mode rejection is the inputs' match: about 0.75 % at DC without a DMM calibration on this board, better after one, and not calibrated at AC.

11.6 Locking to an external reference

An external signal near 10 kHz on IN2 (a chopper's reference, a function generator driving the experiment): oscillator 2 follows it, and the demodulators referenced to oscillator 2 measure IN1 against it.

```
lockin.apply({"outputs": {"2": {"enabled": True, "amplitudes": {"2": 0.1}}},
              "demodulators": {"2": {"input": "in2", "oscillator": 2}}})
analysis.lock_pll(lockin, pid=1, bandwidth=100, oscillator=2, demodulator=2, hold_below=0.01, seed="
    ↪ in2")
lockin.apply({"demodulators": {"1": {"input": "in1", "oscillator": 2, "time_constant": 0.1}}})
```

`hold_below` keeps the frequency when the reference disappears. For a reference the counter cannot measure (weak, noisy, a harmonic), use `acquire` instead of `seed` (section 8.4). The loop's bandwidth decides what the oscillator follows: the reference's slow drifts within it, not its fast phase noise.

11.7 Recording for hours

```
with LockinClient("192.168.0.149") as li:
    rec = li.record(3600, rate=119.2, demodulators=(1, 2))          # one hour at about 119 rows/s
adev = analysis.allan_deviation(rec["x1"], rec["rate"])
```

The stream's rows carry the frame numbers, so a recording's time base is the board's clock; the apply counts in the rows show where someone changed the configuration, and flags show gaps.

12 Specifications

Computed values come from the bit-exact model; measured ones from the board used here (loopback OUT1 → IN1, OUT2 → IN2 with 50 Ω terminators, LV jumpers; the dates are in `docs/VERIFICATION.md` and the figures' captions). Board-wide figures (converters, noise, clocks) are in the root `README.md`.

quantity	value
oscillators	4; 0 to 62.5 MHz in steps of 0.44 μHz; phase in steps of 2^{-32} cycle; continuous or coherent (relative to t_0) switching on a frame boundary
outputs	2: sum of the 4 oscillators' sines (24-bit amplitude words, about 0.07 μV steps; corrected for the DAC's droop and the calibration's output response) + offset (2^{-16} DAC step) + PID terms + a monitor; triangular dither; limits clamp every sample (measured: 0.150009 V for 0.15 V); clipped samples counted
demodulators	4; input IN1, IN2 or IN1 − IN2 (offsets, digital AC coupling of any corner); reference $mf_A + nf_B$, $m = 0 \ldots 1023$, $n \in \{-1, 0, 1\}$, exact to the 48-bit phase; sine (error $\leq$ 0.57 LSB of 2^{-17} before rounding, computed) or square reference
mixing and decimation	products at 125 MSa/s; CIC of order 2, decimation 128: frame rate 976.5625 kHz; the $2f$ product falls on the CIC's nulls when f is a multiple of 488.28125 kHz
filters	1 to 8 RC sections (6 to 48 dB/oct) on X and Y, $\tau = 0.5\,\mu$s to 10^6 s (realized within 2×10^{-5}, computed); ENBW and settling time reported; measured: $\tau = 0.99977$ of a set 0.1 s from a step
readings	X, Y, R (V rms or peak), θ (2^{-32} cycle), unwrapped θ (48 bits: ±32768 cycles), $d\theta/dt$ (0.23 mHz steps, ±488 kHz); one frame read consistently (about 3.4 ms per reading through Python on the board, measured)

quantity	value
accuracy	R against the amplitude of the same captured input samples: 0.99984 to 1.00005 from 1 kHz to 10 MHz; harmonic 3: 1.00000; sideband: 0.49999 for 0.5; square reference: 0.99944; IN1 − IN2: 1.50089 for 1.5 (measured). Absolute: the board calibration (DC) times the analog response
analog loopback response	relative to the set amplitude (calibration of 2026-10-04): 0.960 from DC to 1 kHz, +6.8 % at a pole-zero near 21 kHz, 1.022 at 1 MHz, 1.056 at 10 MHz, 1.00 at 50 MHz, 0.34 at 60 MHz (measured; figure 4); correct with response tables
linearity	the converter loop with dither: within 0.34 % from 0.8 V to 10 mV, 2.5 % at 1 mV (measured)
noise	IN1, IN2 (LV, terminated, outputs off): 0.89 μV/$\sqrt{\text{Hz}}$ at 1 kHz, 0.85 at 100 kHz, 84 nV/$\sqrt{\text{Hz}}$ at 1 MHz, rising below 1 kHz (measured, 2026-10-27; figure 8)
stability	2 h at 1 MHz in loopback: R 56–59 ppm rms, Allan deviation 10 ppm at 1 and 10 s, 20 ppm at 100 s; θ 0.0014° rms at 1 MHz, following the temperature as a 15 ps/K delay (measured)
crosstalk	OUT2 → IN1: −54 to −55 dB from 100 kHz to 10 MHz (measured)
phase	referred to the OUT1 connector through the core's 14 samples and the inputs' loop delays (this board: 119.91 ns IN1, 119.99 ns IN2, measured; a pure delay from 1 to 10 MHz within 0.3°)
PIDs	4 at the frame rate; inputs X, Y, R, θ (wrapped error), $d\theta/dt$, unwrapped phase; targets an oscillator's frequency (0.44 μHz steps) or phase, an output's amplitude of an oscillator, an output's offset; a type-2 PLL up to about 19 kHz bandwidth (computed); D low-pass; limits with anti-windup; hold, hold on low R; lock detection; targets act 3.07 μs after the frame's end plus the output's 14 samples
measured loops	PLL (200 Hz, seeded): oscillator within 1 mHz of the input, phase error 0.0004°, 0.04 Hz rms jitter per reading; relocking from one bandwidth in 14.5, 1.46, 0.145 ms (100 Hz, 1 kHz, 10 kHz); pull-in 7 bandwidths (3 at 10 kHz); with `acquire` from 100 bandwidths in about 0.3 s; amplitude loop: time constants 4.60, 1.35, 0.42 ms for i = 300, 1000, 3000 s^{-1} (predicted 4.62, 1.39, 0.46)
stream	rows of X, Y every 2^j frames (976 562.5/2^j rows/s), timestamped, flagged after a gap; through the service about 7600–9500 rows/s (measured)
capture	4 × 8192 words; full-rate sources (means over 2^k samples, $k = 0 \ldots 24$) and frame-rate readings; triggers immediate, DIO3_P, IN1/IN2 Schmitt levels, software; pretrigger 0 to 8191
frequency counter	IN1 or IN2, Schmitt levels, gate 2^g samples; within 0.1 ppm of the board's clock (measured, 0.1 s gate)
after loading	everything off, outputs at 0 in converter units

A Command reference

A.1 lockin's TCP commands (port 5025)

Replies are `OK <value>` or `ERR <message>`; a refused setting changes nothing. Units are numbered from 1.

command	reply, action
`OSC <k> <json>`, `OSC <k>?`	patch, query oscillator k
`FREQ <k> <Hz>`, `FREQ <k>?`	oscillator k's frequency (the realized one)

command	reply, action
`DEMOD <d> <json>`, `DEMOD <d>?`	patch, query demodulator d
`TC`, `ORDER`, `HARM`, `INPUT`, `REF`, `DPHASE <d> <value>`	one demodulator setting: time constant, order, harmonic, input (`IN1`, `IN2`, `DIFF`), reference (`SINE`, `SQUARE`), phase
`INPUTS <json>`, `INPUTS?`	the inputs' offsets and AC coupling
`OUT <o> <json>`, `OUT <o>?`, `OUT <o> ON|OFF`	output o
`AMP <o> <k> <V>`, `OFFS <o> <V>`	oscillator k's amplitude on output o; output o's offset
`PROTECT <o>`	lock output o's limits until the FPGA is reloaded
`PID <p> <json>`, `PID <p>?`, `PIDRESET <p>`	PID p
`PLL <p> <json>`	`{"bandwidth", "oscillator", "demodulator"[, "range", "hold_below", "seed", "gate", "filter", "acquire", "acquire_timeout"]}: lock_pll`
`STREAM <json>`, `STREAM?`	the stream's rate and demodulators
`UNITS RMS|PEAK`, `UNITS?`	the readings' units
`ZERO`	t_0 := now; replies t_0
`READ?`	`readings()` (JSON)
`X? <d>`, `Y? <d>`, `R? <d>`, `THETA? <d>`	one reading
`AVG? <count> [<d> ...]`	`average`
`AUTOPHASE <d> [<count>]`	auto-phase; replies the new phase
`HW?`, `CLEAR`	`hardware()`; `clear_counts()`
`CAPTURE? <json>`	`{"sources", "rate", "trigger", "pretrigger", "level", "timeout", "words"}`: the capture (without `t`: rebuild it from `rate` and `pretrigger`)
`COUNT? [IN1|IN2] [<gate s>]`	the frequency, or NaN
`SWEEP <json>`	`{"parameter", "target", "values"` or `"start", "stop", "points", "scale", "demodulators", "settle", "settle_factor", "count", "outer", "restore", "fit"}`; at most 40 000 readings per reply
`SPECTRUM? [IN1|IN2] [<averages>] [<rate>]`	`spectrum`
`ZOOM? <json>`	`{"demodulator"[, "span", "points", "averages", "compensate", "remove_mean"]}: zoom_spectrum`
`NOISE? <json>`	`{"demodulator"[, "averages", "points", "exclude"]}: noise_density`
`AUTOTAU <json>`	`{"demodulator"[, "precision", "order", "max_tau", "averages", "apply"]}: auto_tau`
`DATA STREAM [<interval s>]`	the data stream, on its own connection (section B)

A.2 Common commands

command	reply, action
`*IDN?`, `HELP`	identity; the commands
`DESIGNS?`, `DESIGN?`, `LOAD [design]`	installed designs, the active one; switch or reload
`CONFIG?`, `CONFIG <json>`, `CONFIG REPLACE <json>`	read, patch, replace the configuration
`SNAPSHOT?`, `SESSION?`, `REQUEST <json>`	a consistent snapshot; the session; a session-guarded request (the Python client's)

command	reply, action
`DESCRIBE?`, `STATUS?`	descriptions; status (with every demodulator's frequency, ENBW and settling time)
`PRESETS?`, `PRESET <name>`, `PRESET SAVE|DELETE <name>`	presets
`SAFE`	the safe state
`RANGE?`, `RANGE <IN1|IN2> <LV|HV>`	the input jumpers
`CAL?`, `CAL RESET`, `CAL OFFSETS`, `CAL HOLD`, `CAL RELEASE`, `CAL LOOPBACK [<dmm>]`	the board calibration and its measuring steps
`CAL RESPONSE <ch> <[[Hz, gain], ...]>`, `CAL DELAY <ch> <s>`	an output's or input's response or delay

A.3 Python

method	
`config()`, `apply(patch, replace)`, `snapshot()`, `apply_snapshot(...)`	the configuration
`describe()`, `status()`, `safe()`, `presets()`, `load_preset(n)`, `save_preset(n)`	common
`readings()`, `hardware()`, `clear_counts()`, `reset_pid(p)`, `protect(o)`, `zero_time()`	readings and controls
`capture(...)`, `count_frequency(...)`	acquisitions
`calibration()`, `ranges()`, `set_input_range(...)`, `set_input_delay(...)`, `set_input_response(...)`, `calibrate(dmm)`	the board calibration

`analysis` (with the driver or the client): `average`, `auto_phase`, `pll_settings`, `pll_filter`, `fll_settings`, `lock_pll`, `sweep`, `sweep_values`, `fit_resonance`, `spectrum`, `zoom_spectrum`, `noise_density`, `auto_tau`, `filter_response`, `demod_z`, `tau_for_enbw`, `allan_deviation`, `RunningStats`, `measure_input_delay`, `enbw`, `settle_time`. The client adds `set_frequency`, `set_oscillator`, `set_demodulator`, `set_inputs`, `set_output`, `enable`, `set_amplitude`, `set_offset`, `set_pid`, `set_stream`, `set_units`, `value`, `record`; `DataStream(host)` reads the stream.

B The data stream's rows

`DATA STREAM [interval]` (0.05 to 5 s, default 0.2) replies with one header line, `OK {"fields": [...], "format": "base64 of little-endian float64 rows", "interval_s", "rate", "demodulators", "queue_rows", "units", "session", "utc"}`, then one line every interval (also when empty): `DATA {"rows": n, "lost": rows dropped for this client so far, "data": base64}`. Each row has 15 float64 fields:

```
frame t_s x1 y1 x2 y2 x3 y3 x4 y4 locked clipped apply flags units_peak
```

`frame` counts frames since the FPGA's reset, `t_s` is its end in seconds; X and Y are NaN for demodulators not streamed; `locked` the PIDs' flags (bits 0–3); `clipped` 1 IN1, 2 IN2, 4 an output, during the frame; `apply` the apply count (modulo 256); `flags` 1: rows were lost before this one, so the series is continuous only between flags; `units_peak` 1 for peak units. `STOP` ends the stream.