Data Acquisition with NI-DAQmx

For reading voltages and capturing waveforms from an NI DAQ device. Assumes you can work with NumPy arrays — see Arrays and Plotting.

This page covers using Python to read voltages and capture waveforms from a National Instruments USB data-acquisition (DAQ) device via the `nidaqmx` package. Examples use the **NI USB-6009** — a common, low-cost multifunction device — as a concrete example; the patterns apply to other NI-DAQmx devices.

Prerequisites

The NI-DAQmx drivers (from National Instruments) and the Python package (`pip install nidaqmx`).

1 Example Device: USB-6009

Parameter	Value
Analog inputs	8 single-ended or 4 differential
Resolution	14 bits
Max sample rate	48 kS/s (single channel)
Input range	±10 V, ±5 V, ±1 V, ±0.2 V
Analog outputs	2 (12-bit, 0–5 V)
Digital I/O	12 lines

The max sample rate is shared across channels: with 4 channels, each gets ~12 kS/s.

2 Listing Connected Devices

Before acquiring, confirm the device is recognized and learn its name (e.g., `Dev1`):

```
import nidaqmx.system

system = nidaqmx.system.System.local()
for device in system.devices:
    print(f"Name: {device.name}  Type: {device.product_type}  Serial:
{device.serial_num}")
```

The device name (`Dev1` , `Dev2` , …) is what you use to address channels. If another device was connected first, yours might be `Dev2` .

3 Reading a Single Voltage

```
import nidaqmx

def read_voltage(device="Dev1", channel="ai0"):
    with nidaqmx.Task() as task:
        task.ai_channels.add_ai_voltage_chan(f"{device}/{channel}")
```

```
        return task.read()

print(f"Voltage: {read_voltage():.4f} V")
```

Channel naming: `Dev1/ai0` is the first analog input; `Dev1/ai0:3` is channels 0 through 3.

4 Single-Ended vs. Differential Inputs

How you wire the signal must match how you configure the channel, or you'll get garbage readings.

- **RSE (referenced single-ended):** measures the input pin relative to ground. Gives more channels; use it when the signal is already referenced to ground (most bench measurements).
- **Differential (DIFF):** measures the difference between two pins, rejecting common-mode noise. Fewer channels; use it for small signals, long cables, or noisy environments.

```
from nidaqmx.constants import TerminalConfiguration

task.ai_channels.add_ai_voltage_chan(
    "Dev1/ai0",
    terminal_config=TerminalConfiguration.RSE,   # or .DIFF
)
```

Mode	Channels	Best for
RSE	8	Bench signals referenced to ground
DIFF	4	Small signals, noisy environments, floating sources

Common gotcha: if you don't specify `terminal_config`, the device may default to differential. If a signal's negative terminal isn't properly connected, you'll get unexpected readings (often roughly half the expected value, or unstable). When readings look wrong, check that the configuration matches the wiring first.

5 Reading Multiple Samples

To capture a time-varying signal or characterize noise, set a sample rate and a number of samples:

```
import nidaqmx
import numpy as np
from nidaqmx.constants import AcquisitionType

def read_samples(num_samples=1000, sample_rate=1000, device="Dev1", channel="ai0"):
    """Return (times, voltages) for a finite acquisition."""
    with nidaqmx.Task() as task:
        task.ai_channels.add_ai_voltage_chan(f"{device}/{channel}")
        task.timing.cfg_samp_clk_timing(
            rate=sample_rate,
            sample_mode=AcquisitionType.FINITE,
            samps_per_chan=num_samples,
        )
        voltages = np.array(task.read(number_of_samples_per_channel=num_samples))
```

```
    times = np.arange(num_samples) / sample_rate
    return times, voltages

times, voltages = read_samples(num_samples=1000, sample_rate=1000)  # 1 s of data
print(f"mean = {np.mean(voltages):.4f} V,  std = {np.std(voltages):.4f} V")
```

This is the pattern you want whenever the acquisition has a **known duration** — including when it runs alongside something else, such as a stage moving through a scan. Start the motion, call `task.read()`, and it blocks until every sample has arrived. “Streaming data while the motor moves” in that sense is still a *finite* acquisition; Continuous Acquisition below is for runs whose length you don’t know in advance.

Choosing a sample rate: to capture a signal faithfully, sample at least twice its highest frequency (the Nyquist criterion); in practice 5–10× is more comfortable. Sampling too slowly produces *aliasing* — the signal folds down to a false lower frequency. The standard deviation of a steady signal measured this way is the device’s noise floor (e.g., ~5 mV RMS in RSE ±10 V mode on the USB-6009), and you can’t resolve anything smaller than that.

6 Reading Several Channels at Once

Add more channels to the same task and every channel is sampled together on the same clock, so the readings line up in time:

```
import nidaqmx
import numpy as np
from nidaqmx.constants import AcquisitionType

def read_channels(channels=("ai0", "ai1"), num_samples=1000,
                  sample_rate=1000, device="Dev1"):
    """Return (times, data) where data has shape (n_channels, num_samples)."""
    chan_str = ",".join(f"{device}/{ch}" for ch in channels)
    with nidaqmx.Task() as task:
        task.ai_channels.add_ai_voltage_chan(chan_str)
        task.timing.cfg_samp_clk_timing(
            rate=sample_rate,
            sample_mode=AcquisitionType.FINITE,
            samps_per_chan=num_samples,
        )
        data = np.array(task.read(number_of_samples_per_channel=num_samples))
    return np.arange(num_samples) / sample_rate, data

times, data = read_channels(("ai0", "ai1"))
print(f"ai0 mean = {data[0].mean():.4f} V,  ai1 mean = {data[1].mean():.4f} V")
```

Two things to watch. The **sample rate is shared** — the device’s maximum is a total across channels, so two channels at 48 kS/s total gives 24 kS/s each. And on many devices, including the USB-6009, channels are **multiplexed** rather than sampled simultaneously: a single converter switches between them, so readings are offset by a small interchannel delay. That’s irrelevant for slow signals and matters if you’re comparing phase between channels.

7 Continuous Acquisition

Everything above acquires a fixed number of samples and stops. Setting `sample_mode=AcquisitionType.CONTINUOUS` instead starts the device streaming into a buffer that you drain as it fills.

Reach for it when you **don't know the duration in advance** — watching a signal until you decide to stop, waiting on an event, or a run longer than the device's buffer can hold. If you know how long you want to record, even if the acquisition runs alongside a moving stage, the finite pattern above is simpler and safer: `task.read()` blocks until the samples arrive and there is no buffer to fall behind.

The rule that governs all of this: **the device keeps writing into its buffer whether or not you read it.** Fall behind and the buffer wraps, you get a `DaqError` about samples being overwritten, and the run is lost. So read everything available each time round the loop rather than a fixed block.

7.1 Recording every sample

Use this when the run is open-ended but you still need every sample for later analysis:

```
import nidaqmx
import numpy as np
from nidaqmx.constants import AcquisitionType

sample_rate = 10_000
duration = 20.0            # seconds

chunks = []
with nidaqmx.Task() as task:
    task.ai_channels.add_ai_voltage_chan("Dev1/ai0")
    task.timing.cfg_samp_clk_timing(
        rate=sample_rate,
        sample_mode=AcquisitionType.CONTINUOUS,
    )
    task.start()
    # ---- start your motion / stimulus here, so it runs while data streams ----
    try:
        collected = 0
        target = int(sample_rate * duration)
        while collected < target:
            available = task.in_stream.avail_samp_per_chan
            if available:
                chunk = task.read(number_of_samples_per_channel=available)
                chunks.append(np.asarray(chunk))
                collected += len(chunk)
    except KeyboardInterrupt:
        print("Stopped early by user.")

voltages = np.concatenate(chunks)
times = np.arange(len(voltages)) / sample_rate
print(f"{len(voltages)} samples over {times[-1]:.2f} s")
```

Collecting into a list and concatenating once at the end is deliberate: growing a NumPy array inside the loop reallocates it every iteration, which is exactly the work you can't afford while the buffer is filling.

7.2 Watching a signal live

Use this to check a connection or see what a signal looks like before committing to a careful run. It shows only the most recent samples and **does not keep the rest** — the older data is discarded as it arrives:

```
import nidaqmx
import numpy as np
import matplotlib.pyplot as plt
from nidaqmx.constants import AcquisitionType
from IPython.display import display, clear_output

sample_rate = 10_000
window = 1000              # most recent samples to display

fig, ax = plt.subplots(figsize=(9, 4))
line, = ax.plot([], [])
ax.set_xlabel("Sample")
ax.set_ylabel("Voltage (V)")

with nidaqmx.Task() as task:
    task.ai_channels.add_ai_voltage_chan("Dev1/ai0")
    task.timing.cfg_samp_clk_timing(
        rate=sample_rate,
        sample_mode=AcquisitionType.CONTINUOUS,
    )
    task.start()
    try:
        while True:
            available = task.in_stream.avail_samp_per_chan
            if available:
                data = np.asarray(task.read(number_of_samples_per_channel=available))
                recent = data[-window:]
                line.set_data(np.arange(len(recent)), recent)
                ax.set_xlim(0, len(recent))
                ax.set_ylim(recent.min() - 0.1, recent.max() + 0.1)
                clear_output(wait=True)
                display(fig)
    except KeyboardInterrupt:
        pass

plt.close(fig)
```

Interrupt the kernel to stop it. Redrawing is far slower than the acquisition, so draining the whole buffer each pass — and plotting only the tail — is what keeps the device from overrunning while the figure catches up.

> If you get “samples overwritten”
>
> `DaqError: ... Attempted to read samples that are no longer available` means the loop fell behind. Lower the sample rate, do less work inside the loop (defer plotting and analysis until after), or enlarge the buffer with `task.in_stream.input_buf_size`.

8 Voltage Range and Noise

Narrower input ranges have lower noise floors but clip larger signals. Set the range explicitly with `min_val` / `max_val`:

```
from nidaqmx.constants import TerminalConfiguration

task.ai_channels.add_ai_voltage_chan(
    "Dev1/ai0",
    terminal_config=TerminalConfiguration.RSE,
    min_val=-10.0, max_val=10.0,
)
```

On the USB-6009, the ±1 V range has a markedly lower noise floor than ±10 V — but it saturates above 1 V. Pick the narrowest range your signal never exceeds.

9 Analog Output

The USB-6009 can also generate voltages (0–5 V only — specify the range or you'll get an error):

```
import nidaqmx
import time

with nidaqmx.Task() as task:
    task.ao_channels.add_ao_voltage_chan("Dev1/ao0", min_val=0.0, max_val=5.0)
    task.write(2.5, auto_start=True)   # output 2.5 V
    time.sleep(5)
    # output returns to 0 V when the task closes
```

Loopback test: wire `ao0` to `ai0`, write a voltage, and read it back to confirm both work.

10 When the Device Can't Do What You Need

Low-cost DAQ hardware leaves things out, and the thing it most often leaves out is **triggering on the signal itself**. The USB-6009 has a digital trigger input but no analog one: you cannot tell it "start acquiring when the input crosses 2 V."

That is usually not fatal. A DAQ hands you an array, and most hardware features you're missing can be recovered by doing the work on the array after it arrives. The two patterns below cover most of it, and the habit generalises past NI hardware — when the device can't do it, look at whether the data can.

10.1 Catching an event you can't trigger on

If you need to capture something that happens *because* you caused it — a discharge, a step response, a switch closing — you don't need a trigger at all. Start the acquisition first, cause the event second, and record enough extra data at the front that the event is guaranteed to land inside the buffer.

The extra data is the important part. Between "start the task" and "cause the event" sits an unpredictable delay: the OS may deschedule your thread, and the driver takes time to arm. So capture a **headroom** of

samples ahead of where you expect the event, sized generously against that jitter — a few milliseconds comfortably covers typical Windows scheduling delays.

```
import time

import nidaqmx
import numpy as np
from nidaqmx.constants import AcquisitionType, TerminalConfiguration

SAMPLE_RATE = 100_000
KEEP = 300                 # samples of the event itself that you want
HEADROOM_MS = 5.0          # generous cover for scheduling jitter

headroom = int(HEADROOM_MS / 1000 * SAMPLE_RATE)
total = KEEP + headroom

with nidaqmx.Task() as ai, nidaqmx.Task() as ao:
    ai.ai_channels.add_ai_voltage_chan(
        "Dev1/ai0", terminal_config=TerminalConfiguration.RSE,
        min_val=-10.0, max_val=10.0,
    )
    ai.timing.cfg_samp_clk_timing(
        SAMPLE_RATE, sample_mode=AcquisitionType.FINITE, samps_per_chan=total,
    )
    ao.ao_channels.add_ao_voltage_chan("Dev1/ao0", min_val=0.0, max_val=5.0)

    ao.write(5.0, auto_start=True)      # charge / set up
    time.sleep(0.01)

    ai.start()                          # arm FIRST
    ao.write(0.0, auto_start=True)      # then cause the event
    raw = np.asarray(ai.read(number_of_samples_per_channel=total, timeout=5.0))
```

Now find the event in the array. For a step down, the event is the single largest drop between consecutive samples, which `np.diff` and `argmin` locate directly:

```
def trim_to_event(raw, keep):
    """Return `keep` samples starting at the sample before the steepest drop."""
    if np.ptp(raw) < 0.5 * max(abs(raw.max()), 1.0):
        return raw[:keep]               # nothing happened — return the head so you can
debug
    edge = int(np.argmin(np.diff(raw)))
    return raw[edge:edge + keep]

voltage = trim_to_event(raw, KEEP)
times = np.arange(len(voltage)) / SAMPLE_RATE      # t = 0 at the event
```

Two details worth copying. The **sanity check** compares the range of the whole buffer against the peak, not one sample against its neighbour — for an exponential decay the first step can be a few percent of the amplitude, so a per-sample threshold would reject perfectly good data. And returning the head of the buffer on failure, rather than raising, leaves you something to plot when you are trying to work out why nothing happened.

10.2 A software trigger for a stable display

Streaming with `AcquisitionType.CONTINUOUS` gives you a fresh window of samples each pass, starting wherever the buffer happened to be. Plot those windows in sequence and a perfectly steady sine wave will crawl across the screen, because nothing aligns one frame to the next.

An oscilloscope solves this with a trigger. You can do the same in a few lines: acquire **twice** the window you intend to display, find the first upward crossing of the signal's own mean, and display the window that starts there.

```
def apply_trigger(raw, display_n, edge="rising"):
    """Align a buffer to its first mean-level crossing."""
    if len(raw) < display_n + 1:
        return raw[:display_n]

    level = raw.mean()
    last_start = len(raw) - display_n       # leave room for a full window
    prev, nxt = raw[:last_start], raw[1:last_start + 1]

    if edge == "falling":
        crossings = np.nonzero((prev >= level) & (nxt < level))[0]
    else:
        crossings = np.nonzero((prev < level) & (nxt >= level))[0]

    start = int(crossings[0]) if crossings.size else 0
    return raw[start:start + display_n]
```

Read `2 * display_n` samples each pass and hand them to this, and a periodic signal sits still. Using the **mean** as the trigger level rather than a fixed voltage means it works without configuration on any signal that crosses its own average, which is most of them. The `crossings.size` fallback matters: a DC or flat signal never crosses, and free-running from the start of the buffer is better than an exception in a display loop.

Oversampling by 2× is what makes it work — you need slack in the buffer for the trigger point to move around in. Without it there is no window left after the crossing.

11 Saving Data to CSV

```
import numpy as np

def save_acquisition(times, voltages, filename, metadata=None):
    header = []
    if metadata:
        header += [f"# {k}: {v}" for k, v in metadata.items()]
    header.append("time_s,voltage_V")
    np.savetxt(filename, np.column_stack((times, voltages)),
               delimiter=',', header="\n".join(header), comments='')

save_acquisition(times, voltages, "measurement.csv",
                 metadata={"device": "USB-6009", "channel": "ai0", "sample_rate": 1000})
```

Including the acquisition settings in the header makes the file self-documenting — your future self will thank you.

12 Troubleshooting

`DaqError: Device identifier is invalid` — check the USB connection; run the device-listing snippet to see the real name; the device may be `Dev2`; confirm drivers are installed.

`DaqError: Sample rate is too high` — the device max is shared across channels; with multiple channels, divide by the channel count, or lower the rate.

Data looks wrong — check the signal is on the right terminal, ground is connected, the signal is within the input range, and the sample rate clears Nyquist.

Wrap acquisitions in error handling so failures are legible:

```
import nidaqmx
from nidaqmx.errors import DaqError

try:
    with nidaqmx.Task() as task:
        task.ai_channels.add_ai_voltage_chan("Dev1/ai0")
        print(f"{task.read():.4f} V")
except DaqError as e:
    print(f"DAQ error: {e}")
```

Back to Python resources