QuPyt GUI User Manual

Version 2

This document is a comprehensive user guide for the QuPyt-GUI (Version 2), a graphical interface for running quantum-sensing experiments with the QuPyt framework. It covers installation, setup, all GUI components, YAML experiment files, pulse-sequence generation, and advanced customization.

July 12, 2025

Contents

1 Introduction

QuPyt is a Python toolkit for controlling hardware in quantum sensing experiments (AWGs, DAQs, microwave sources, etc.) via terminal commands and YAML configuration files. The QuPyt-GUI eliminates much of the manual terminal work by providing a point-and-click interface for:

- Defining and editing experiment descriptors (parameters, pulse patterns, sequence logic)
- Generating pulse-sequence Python modules automatically
- Building the master experiment YAML and deploying it internally
- Launching and monitoring a “watcher” process that runs QuPyt in the background
- Live and post-run data visualization (ODMR spectra, fits, summaries)
- Managing presets and experiment definitions

Why a GUI? Without QuPyt-GUI, each experiment cycle requires:

1. Creating/activating a virtual environment
2. Installing/editing YAML descriptor files by hand
3. Writing or editing pulse-sequence Python modules
4. Running terminal commands and manually copying YAML file to the waiting room for every run of the experiment
5. Monitoring output in the console

This can be tedious, error-prone, and slows down iterative experiment design. QuPyt-GUI streamlines the entire workflow.

2 Installation

2.1 Download

Clone or download the GUI from GitHub:

1. Go to `https://github.com/Aman-Sunesh/QuPyT-GUI-for-Quantum-Sensing-Experiments`
2. Download the folder named `Version 2`
3. Rename it to `GUI` and place it inside your QuPyt root: `<base_path>\QuPyt-master\GUI`

2.2 Dependencies

Open a command-line shell (e.g. Windows PowerShell, Command Prompt, or the integrated terminal in VS Code) and run:

```
# 1. Create & activate venv
python -m venv venv
.\venv\Scripts\activate

# 2. Install QuPyt in editable mode
cd "<path-to-your-local-QuPyt-folder>"
pip install -e .

# 3. Install Python packages
pip install matplotlib nidaqmx numpy pulsestreamer pypylon \
            pyserial pyvisa pyvisa-py PyYAML termcolor \
            tqdm watchdog PyQt6 pyqtgraph pydantic \
            windfreak harvester harvesters jinja2 scipy
```

2.3 First Launch

You can start the QuPyt-GUI in two convenient ways:

1. **From the terminal:**

   ```
   cd /d "C:/path/to/QuPyt-master"
   .\venv\Scripts\Activate.ps1    # in PowerShell
   python -m GUI.main
   ```

2. **Via a desktop shortcut (.bat file):**
   Create a new text file, paste the following, and save it as e.g. `Launch_QuPyt_GUI.bat`:

   ```
   @echo off
   cd /d "C:/path/to/QuPyt-master"
   set PYTHONPATH=%CD%
   "%CD%\venv\Scripts\pythonw.exe" -m GUI.main
   ```

   Double-clicking this shortcut will automatically activate the venv, set the PYTHONPATH, and launch the GUI.

3. **Modified QuPyt Source Files for GUI Compatibility**
   To ensure seamless integration with our custom GUI, three files from the original QuPyt repository by Karl Briegel were modified:

   (a) `SequenceDesigner.py`

   (b) `sensors.py`

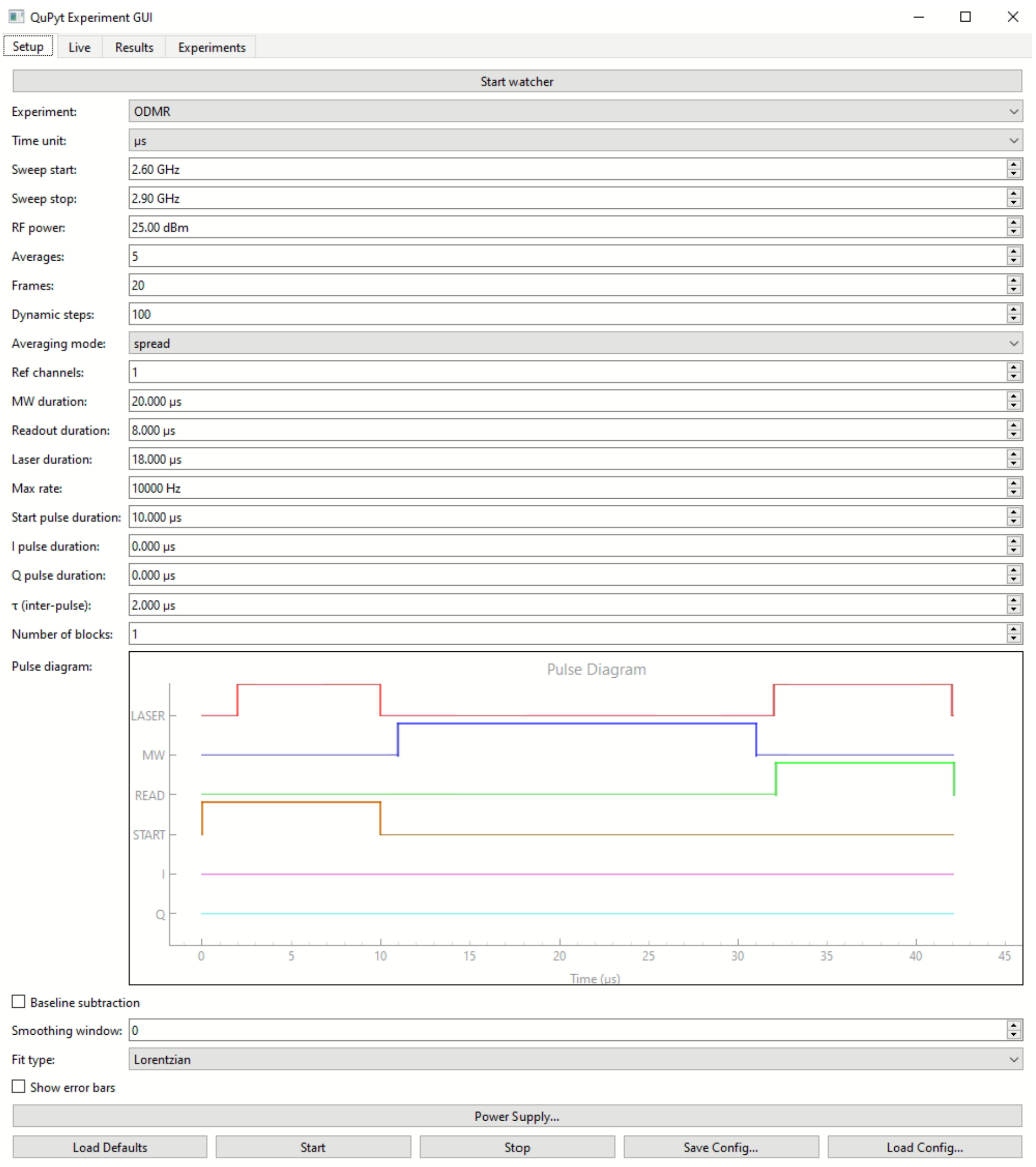
   (c) `yaml_sequence.py`

   All modified files are available at:

   `https://github.com/Aman-Sunesh/QuPyT-GUI-for-Quantum-Sensing-Experiments/tree/main/Modified%20QuPyt%20Files`

   Please ensure that these versions replace the original ones in your working directory for full GUI functionality. These changes preserve core logic but enable GUI-based parameter handling, live execution, and compatibility with additional hardware controls.

3 Overview of the GUI



QuPyt Experiment GUI

Setup | Live | Results | Experiments

Start watcher

Experiment: ODMR
Time unit: μs
Sweep start: 2.60 GHz
Sweep stop: 2.90 GHz
RF power: 25.00 dBm
Averages: 5
Frames: 20
Dynamic steps: 100
Averaging mode: spread
Ref channels: 1
MW duration: 20.000 μs
Readout duration: 8.000 μs
Laser duration: 18.000 μs
Max rate: 10000 Hz
Start pulse duration: 10.000 μs
I pulse duration: 0.000 μs
Q pulse duration: 0.000 μs
τ (inter-pulse): 2.000 μs
Number of blocks: 1
Pulse diagram:

☐ Baseline subtraction
Smoothing window: 0
Fit type: Lorentzian
☐ Show error bars

Power Supply...

Load Defaults | Start | Stop | Save Config... | Load Config...

Figure 1: Main QuPyt-GUI window with **Setup**, **Live**, **Results**, and **Experiments** tabs.

The GUI is divided into four main tabs:

- **Setup**: select experiment type, set parameters, preview pulse diagram, launch the experiment watcher.
- **Live**: real-time log output, progress bars, and live ODMR spectrum plotting.
- **Results**: view metadata, processed spectrum, fitted parameters, summary statistics, and export.
- **Experiments**: add/edit/delete YAML descriptors via a sub-dialog editor.

4 Setup Tab

4.1 Watcher Control

- **Start watcher**: Manually launches only the QuPyt watcher process (equivalent to running `python -m qupyt.main`). The watcher will then monitor the `~/.qupyt/waiting_room` directory for newly deployed experiment YAML files and stream log/output into the Live tab. *Note:* this does *not* regenerate any pulse-sequence code or write new YAMLs — it simply starts (or restarts) the watcher. You only need to click this once before your first run (the **Start** button below will also invoke it automatically).

4.2 Experiment Selection

- **Experiment**: choose from your own descriptors (loaded from `GUI/experiments/*.yaml`).

4.3 Sweep & Acquisition Parameters

`Sweep start` (GHz): RF source start frequency. *Type:* float spin-box; range auto-scaled to hardware limits—sets the lower bound of the frequency sweep.

`Sweep stop` (GHz): RF source stop frequency. *Type:* float spin-box; range auto-scaled—sets the upper bound of the frequency sweep.

`RF power` (dBm): output power level for the WindFreak (or other RF) source. *Type:* integer slider; adjusts microwave drive strength during each MW pulse.

`Averages` : total number of repeats per sweep point. *Type:* integer spin-box; higher values reduce noise at the cost of longer acquisition time.

`Frames` : number of detector readout frames collected per pulse block. *Type:* integer spin-box; sets the DAQ's internal buffer depth / batching.

`Dynamic steps` : number of discrete frequency points in the sweep. *Type:* integer spin-box; controls resolution of your ODMR scan.

`Averaging mode` : choice of how multi-frame data is combined. *Options:* "sum" (accumulate counts) or "spread" (store individual frames).

`Ref channels` : number of APD reference inputs recorded alongside the signal. *Type:* integer spin-box; used for common-mode noise subtraction.

4.4 Pulse Timing Controls

`MW duration` (µs): length of the microwave drive pulse within each block.

`Readout duration` (µs): length of the "read" pulse that gates the APD count. *Notes:* During this time the DAQ's "READ" line is high; the photon counts arriving at the APD are actively digitized.

`Laser duration` (µs): total optical repolarization time following readout. *Breakdown:* Internally this is split into two back-to-back "LASER" pulses:

- A *readout sub-pulse* (as discussed above) of length = `Readout duration`
- A *repolarization sub-pulse* of length = `Laser duration` − `Readout duration`

Spacing: these two sub-pulses are separated by a fixed "dead time" ($\sim 2\,\mu s$) for charge relaxation. This setting can be changed in the Experiments tab; we'll discuss it shortly in the following sections.

`Start pulse duration` (µs): an initial TTL "START" trigger pulse sent at time zero. *Purpose:* Arms the DAQ / other downstream hardware so that subsequent "READ" pulses actually result in recorded data.

`I pulse duration` (µs): duration of the auxiliary "I" microwave pulse (for tomography or correlation sequences).

`Q pulse duration` (µs): duration of the auxiliary "Q" microwave pulse.

τ (µs) Inter-pulse delay: delay between MW and readout within each block, or between dynamical decoupling pulses. *Type:* float; used for spin-echo, XY8, or other multi-pulse sequences.

`Max Rate` (Hz): upper limit on how many full pulse-sequence cycles per second the system will attempt. *Details:* Internally enforces

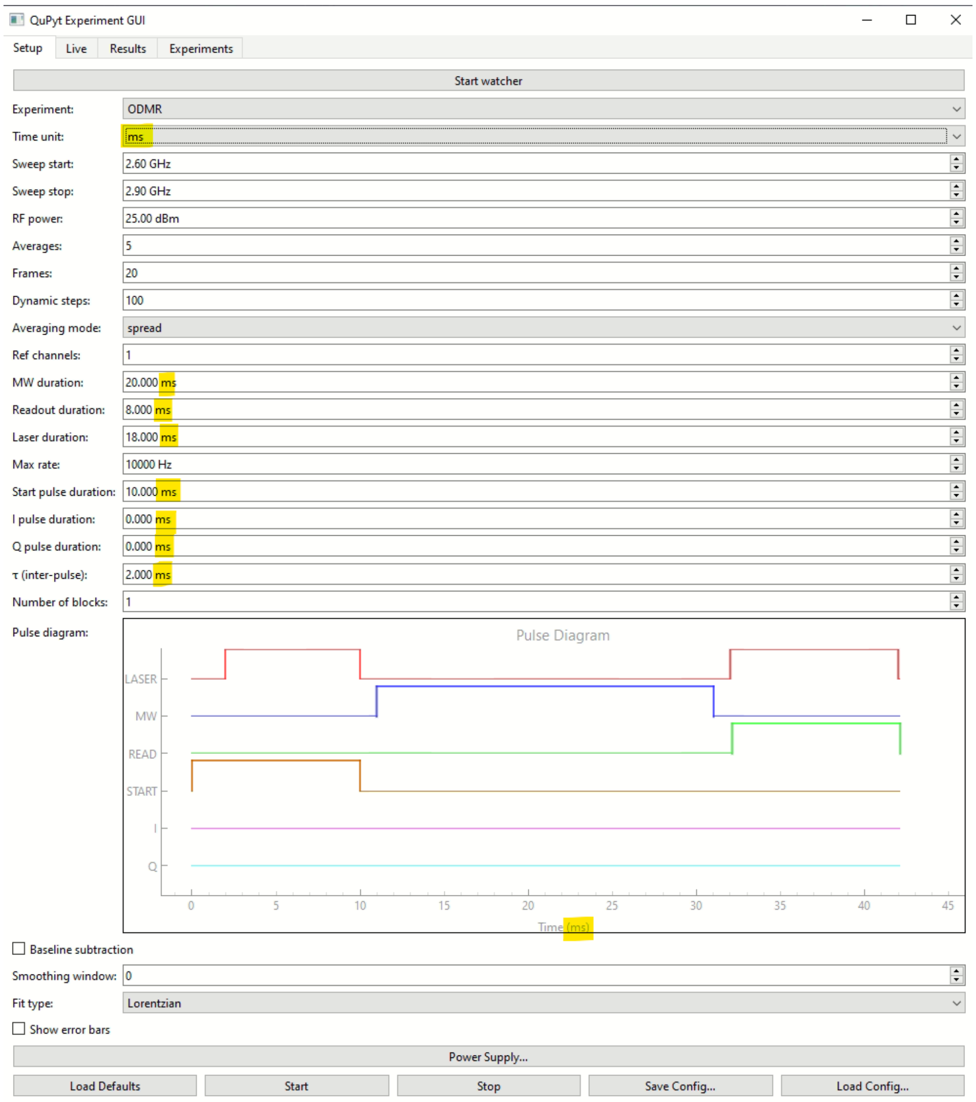
$$\text{Total Period} \geq \frac{1}{\texttt{Max rate}}$$

so that you never exceed hardware timing limits (laser driver, DAQ throughput, etc.).

`Number of blocks` : How many distinct sequence segments (e.g., "wait_loop" vs. "block_0") you'll run per measurement.
Type: Integer; for simple ODMR this is usually 2 (read + reference), but you may add more for advanced protocols.

4.5 Time–Unit Selector

- **Time unit**: choose among `ns`, `µs`, or `ms`. Although QuPyt internally operates in microseconds (µs), this selector lets you enter times in nanoseconds or milliseconds and automatically converts them to µs.
  - `ns` → multiplied by 1×10^{-3} to yield µs
  - `µs` → no conversion
  - `ms` → multiplied by 1×10^{3} to yield µs
- All spin-box inputs (e.g. MW duration, readout duration, inter-pulse delay) and the pulse diagram's time axis update immediately when you change the unit.


QuPyt Experiment GUI

Setup | Live | Results | Experiments

Start watcher

Experiment:	ODMR
Time unit:	ms
Sweep start:	2.60 GHz
Sweep stop:	2.90 GHz
RF power:	25.00 dBm
Averages:	5
Frames:	20
Dynamic steps:	100
Averaging mode:	spread
Ref channels:	1
MW duration:	20.000 ms
Readout duration:	8.000 ms
Laser duration:	18.000 ms
Max rate:	10000 Hz
Start pulse duration:	10.000 ms
I pulse duration:	0.000 ms
Q pulse duration:	0.000 ms
τ (inter-pulse):	2.000 ms
Number of blocks:	1

Pulse diagram:

☐ Baseline subtraction

Smoothing window:	0
Fit type:	Lorentzian

☐ Show error bars

Power Supply...

Load Defaults | Start | Stop | Save Config... | Load Config...

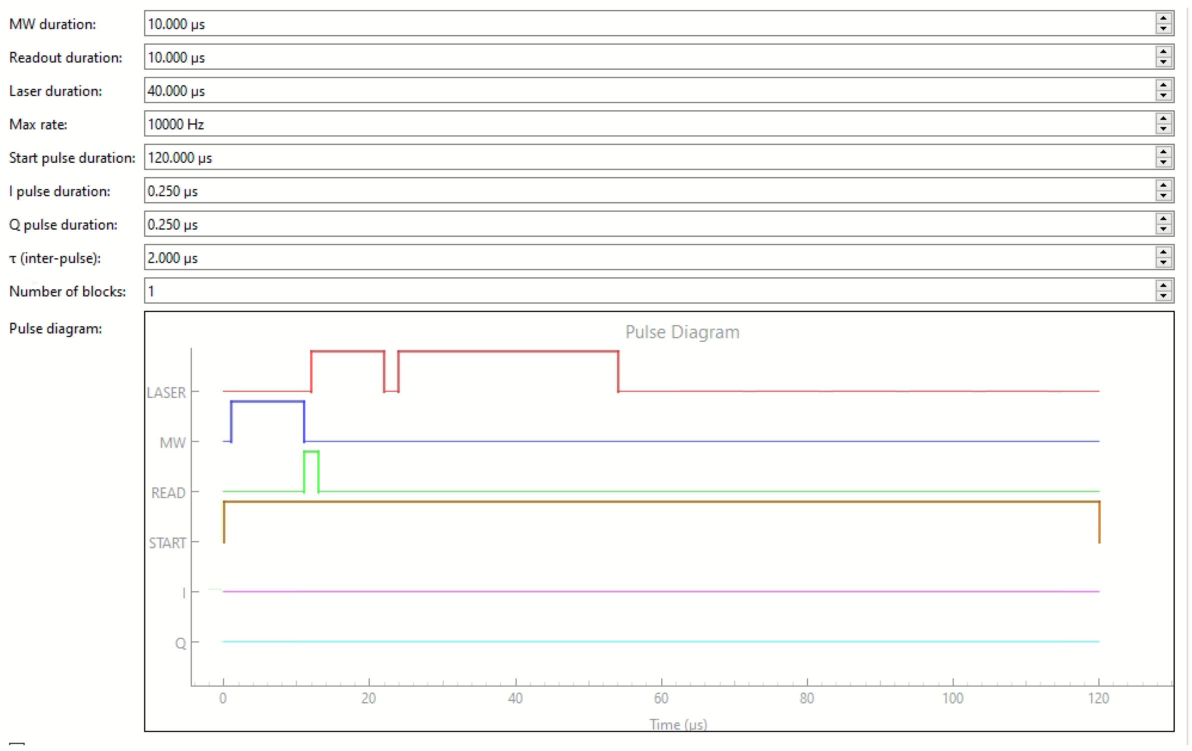
Figure 2: Selecting time units converts all timing values and axes to the internal µs scale.

4.6 Pulse Diagram Preview

The Pulse Diagram provides an immediate, graphical representation of every TTL channel's timing, letting you instantly catch overlaps, incorrect delays, or missing gaps before you run the experiment. As soon as you adjust any timing spin-box (MW, LASER, READ, I, Q, START, τ, blocks), the plot redraws—no need to deploy or run.

The diagram shows:

- *Baselines* (flat segments) during idle periods on each channel.
- *Pulses* (up–over–down boxes) colored by channel (LASER, MW, READ, START, I, Q).
- *Left-axis ticks* labelled with channel names for quick mapping to your hardware lines.
- *Bottom-axis units* (ns/µs/ms) matching the Time–Unit selector for consistent entry and display.



MW duration:	10.000 µs
Readout duration:	10.000 µs
Laser duration:	40.000 µs
Max rate:	10000 Hz
Start pulse duration:	120.000 µs
I pulse duration:	0.250 µs
Q pulse duration:	0.250 µs
τ (inter-pulse):	2.000 µs
Number of blocks:	1

Pulse diagram:

Figure 3: Real-time preview of the pulse sequence on each channel.

4.7 Buttons

- **Load Defaults**: Resets timing parameters to YAML defaults. These defaults can be changed in the Experiments tab.
- **Start**:

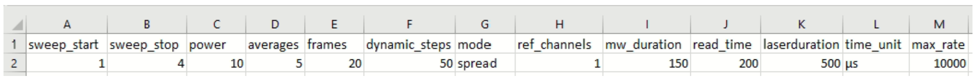
  Performs the complete "prepare & run" workflow:

  1. Generates the low-level pulse-sequence Python module (`user_pulse_seq.py`).
  2. Writes your experiment YAML file (`<Experiment>.yaml`) to the Desktop.
  3. Snapshots and saves the current GUI settings.
  4. Launches the QuPyt watcher process (same action as clicking **Start watcher**), then switches to the Live tab.

  Note: This differs from **'Start watcher'** in that **Start watcher** should be used only when you already have a valid YAML (e.g. written by hand or from another GUI session) and simply want to launch or restart the watcher. In contrast, use **Start** for the normal workflow: pick your parameters in the GUI, click "Start," and the GUI handles code generation, YAML deployment, and live-data streaming all in one step.

- **Stop**: Stops the running watcher process and ensures the PulseBlaster stops sending pulses to all channels. Used to force-stop experiments.

- **Save Config. . . / Load Config. . .** : If there are specific values for an experiment (e.g., ODMR) that are used repeatedly, they may be saved by clicking **Save Config**. The GUI saves them as a CSV file.



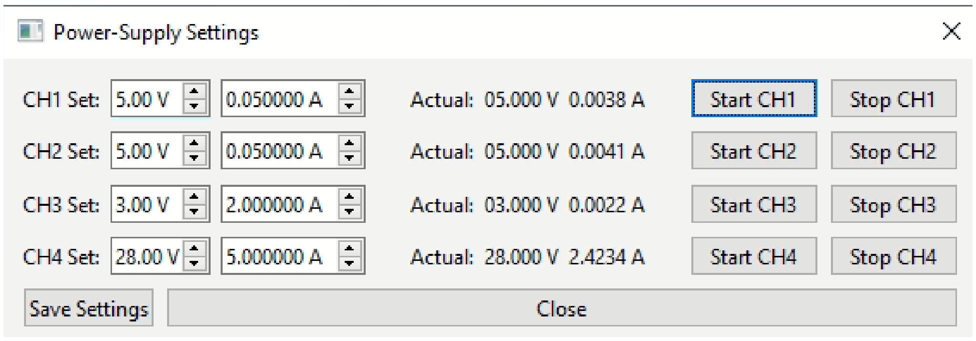
	A	B	C	D	E	F	G	H	I	J	K	L	M
1	sweep_start	sweep_stop	power	averages	frames	dynamic_steps	mode	ref_channels	mw_duration	read_time	laserduration	time_unit	max_rate
2	1	4	10	5	20	50	spread	1	150	200	500	µs	10000

Figure 4: Sample saved CSV file generated.

Any saved CSV may then be loaded as needed by clicking on the **Load Config...** button.

Note: each time you click the **Start** button, the GUI snapshots your setup parameters into `<QuPyt-root>/.qupyt/last_config.json`. On subsequent launches, these values are automatically restored, so you return to the same configuration you used last time.

- **Power Supply**: Opens the Korad KC3405 control dialog for configuring power output on up to four channels (CH1–CH4). Ensure the KC3405 is powered on and connected via USB or serial before attempting to adjust any voltage or current setpoints..



Power-Supply Settings

CH1 Set:	5.00 V	0.050000 A	Actual: 05.000 V 0.0038 A	Start CH1	Stop CH1
CH2 Set:	5.00 V	0.050000 A	Actual: 05.000 V 0.0041 A	Start CH2	Stop CH2
CH3 Set:	3.00 V	2.000000 A	Actual: 03.000 V 0.0022 A	Start CH3	Stop CH3
CH4 Set:	28.00 V	5.000000 A	Actual: 28.000 V 2.4234 A	Start CH4	Stop CH4

Figure 5: Korad power supply control panel for CH1–CH4 configuration.

This dialog allows users to:

1. Enter desired voltage in the "V Set" spin box (0–30 V).
2. Enter desired current in the "I Set" spin box (10 µA–10 A).
3. Click **Start CH** to enable output on that specific channel.
4. Click **Stop CH** to disable output.
5. Monitor real-time voltage and current via "Actual" labels, updated every 500 ms.
6. Click **Save Settings** to save values in `~/.qupyt/power_supply_config.json`. The saved values are automatically reloaded on next launch.

Note: This dialog is specifically designed for the Korad KC3405 programmable DC power supply (it may also work with other Korad models). The rest of the GUI will function normally even if no PSU is connected. To support a different power supply, edit `power_supply.py`.

5 Live Tab

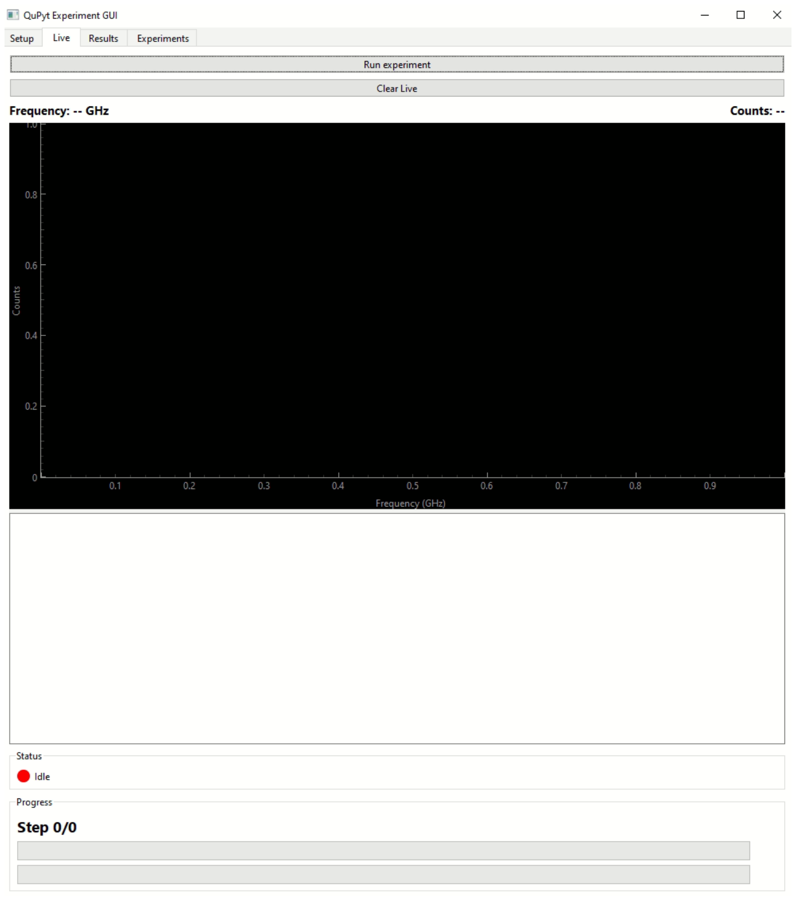


Figure 6: Live monitoring of QuPyt process: console log, progress bars, and live ODMR plot.

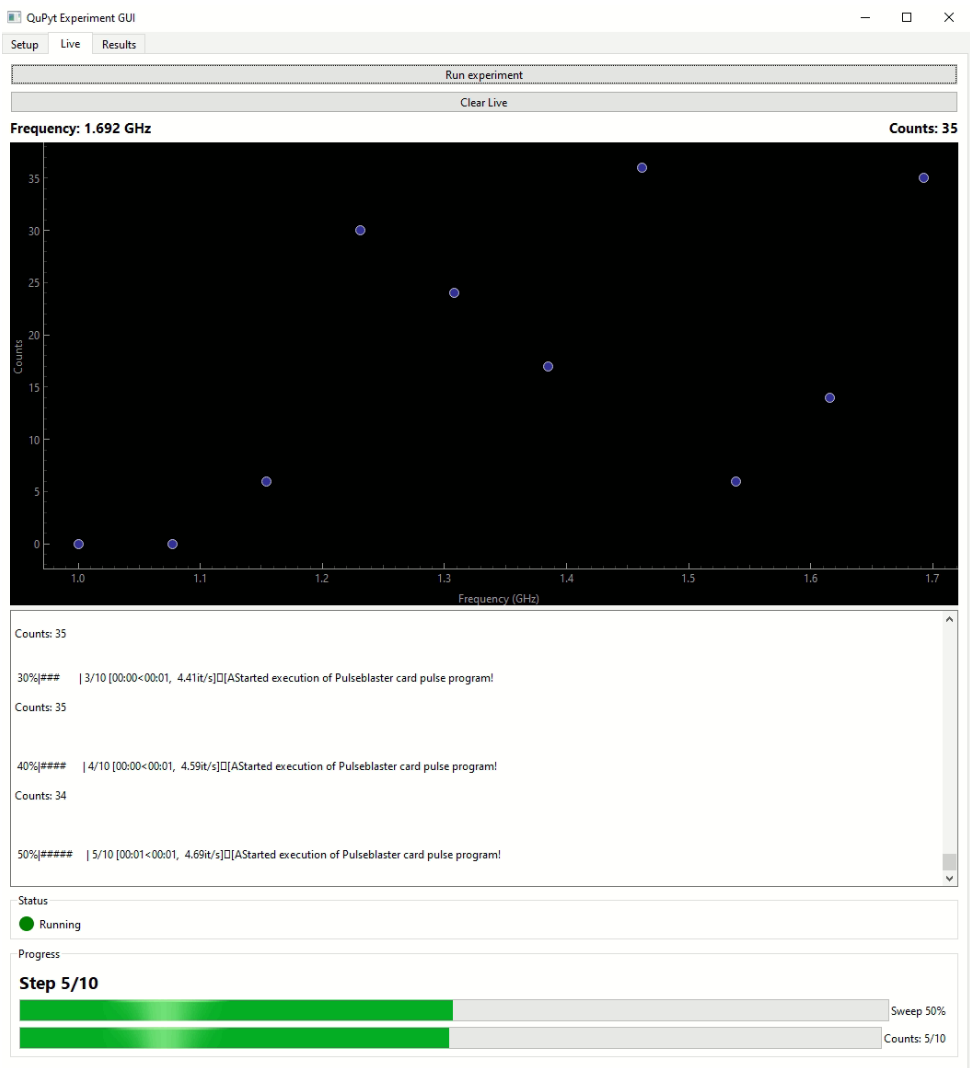
The Live tab displays:

- **Run experiment**: Deploys the currently configured experiment YAML into the waiting room ($\tilde{/}$`.qupyt/waiting_room`) and starts the QuPyt watcher process to execute the run.
- **Clear Live**: resets the plot and log.
- **Live ODMR Spectrum**: A real-time scatter plot of photon counts vs. RF frequency (GHz), updated continuously as the experiment progresses to give immediate visual feedback on your ODMR scan.
- **Current values**: most recent frequency and count readout.
- **Status LED / label**: shows *Idle* (red) or *Running* (green).
- **Progress**:
  - Step n/m label
  - Sweep progress bar (%)
  - Counts gauge (step count out of total)

5.1 Live ODMR Spectrum (Realtime Plot)

As soon as the QuPyt watcher begins streaming data, the Live ODMR Spectrum pane updates *in real time*. Each new frequency–count pair immediately appears as a blue marker, letting you:

- *Watch the resonance dip form* as the frequency sweeps through the NV center's ESR transition.
- *Detect anomalies on the fly*—any unexpected spikes or noise become instantly visible.
- *Gauge data quality* by eye before the run completes, so you can abort or adjust parameters mid-experiment.



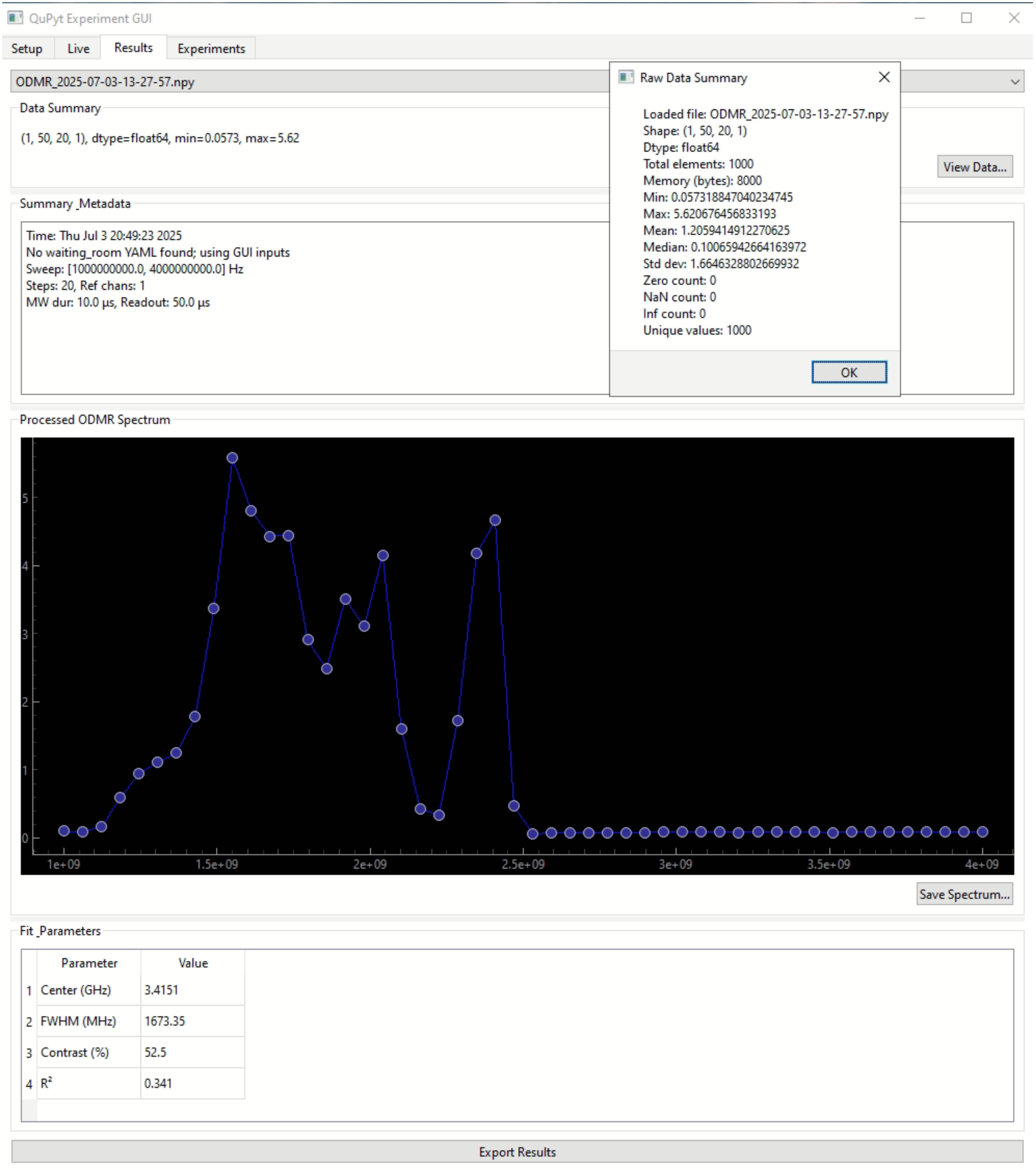
Frequency: 1.692 GHz
Counts: 35

```
Counts: 35

 30%|###     | 3/10 [00:00<00:01,  4.41it/s]□[AStarted execution of Pulseblaster card pulse program!
Counts: 35

 40%|####    | 4/10 [00:00<00:01,  4.59it/s]□[AStarted execution of Pulseblaster card pulse program!
Counts: 34

 50%|#####   | 5/10 [00:01<00:01,  4.69it/s]□[AStarted execution of Pulseblaster card pulse program!
```

Step 5/10
Sweep 50%
Counts: 5/10

Figure 7: Real-time scatter plot of counts vs. frequency, updating continuously during the experiment.

6 Results Tab



ODMR_2025-07-03-13-27-57.npy

Data Summary

(1, 50, 20, 1), dtype=float64, min=0.0573, max=5.62

Summary Metadata

```
Time: Thu Jul 3 20:49:23 2025
No waiting_room YAML found; using GUI inputs
Sweep: [1000000000.0, 4000000000.0] Hz
Steps: 20, Ref chans: 1
MW dur: 10.0 µs, Readout: 50.0 µs
```

Raw Data Summary

```
Loaded file: ODMR_2025-07-03-13-27-57.npy
Shape: (1, 50, 20, 1)
Dtype: float64
Total elements: 1000
Memory (bytes): 8000
Min: 0.057318847040234745
Max: 5.620676456833193
Mean: 1.2059414912270625
Median: 0.10065942664163972
Std dev: 1.6646328802669932
Zero count: 0
NaN count: 0
Inf count: 0
Unique values: 1000
```

Fit Parameters

	Parameter	Value
1	Center (GHz)	3.4151
2	FWHM (MHz)	1673.35
3	Contrast (%)	52.5
4	R^2	0.341

Figure 8: Results tab: file selector, metadata, processed spectrum, fit table, and data summary.

6.1 File Selector

Choose among all `ODMR_*.npy` files in the working directory. By default, the dropdown automatically selects the most recently modified file, so you immediately see the latest completed run when you switch to the Results tab.

6.2 View Data

Clicking **'View Data. . . '** opens a summary dialog that displays basic statistics of the currently loaded NumPy array, including:

- Array *shape* and *data type*
- *Min*, *max*, *mean*, *median*, and *standard deviation*
- Counts of `0`, `NaN`, and `Inf` values
- Number of *unique* values

6.3 Summary & Metadata

Displays:

- Timestamp of run
- Sweep range, steps, averages
- Pulse durations (MW, readout)

6.4 Processed ODMR Spectrum

Plots mean counts (and optionally reference-corrected difference) vs. frequency. Includes **Save Spectrum. . .** button to export PNG.

6.5 Fit & Parameter Table

Fits the spectrum to either a Lorentzian or Gaussian:

- *Center (GHz)*
- *FWHM (MHz)*
- *Contrast (%)*
- R^2

6.6 Export Results

The **Export Results** button lets you save the processed experiment data as a NumPy (`.npy`) file for downstream analysis or plotting in other tools.

7 Experiments Tab

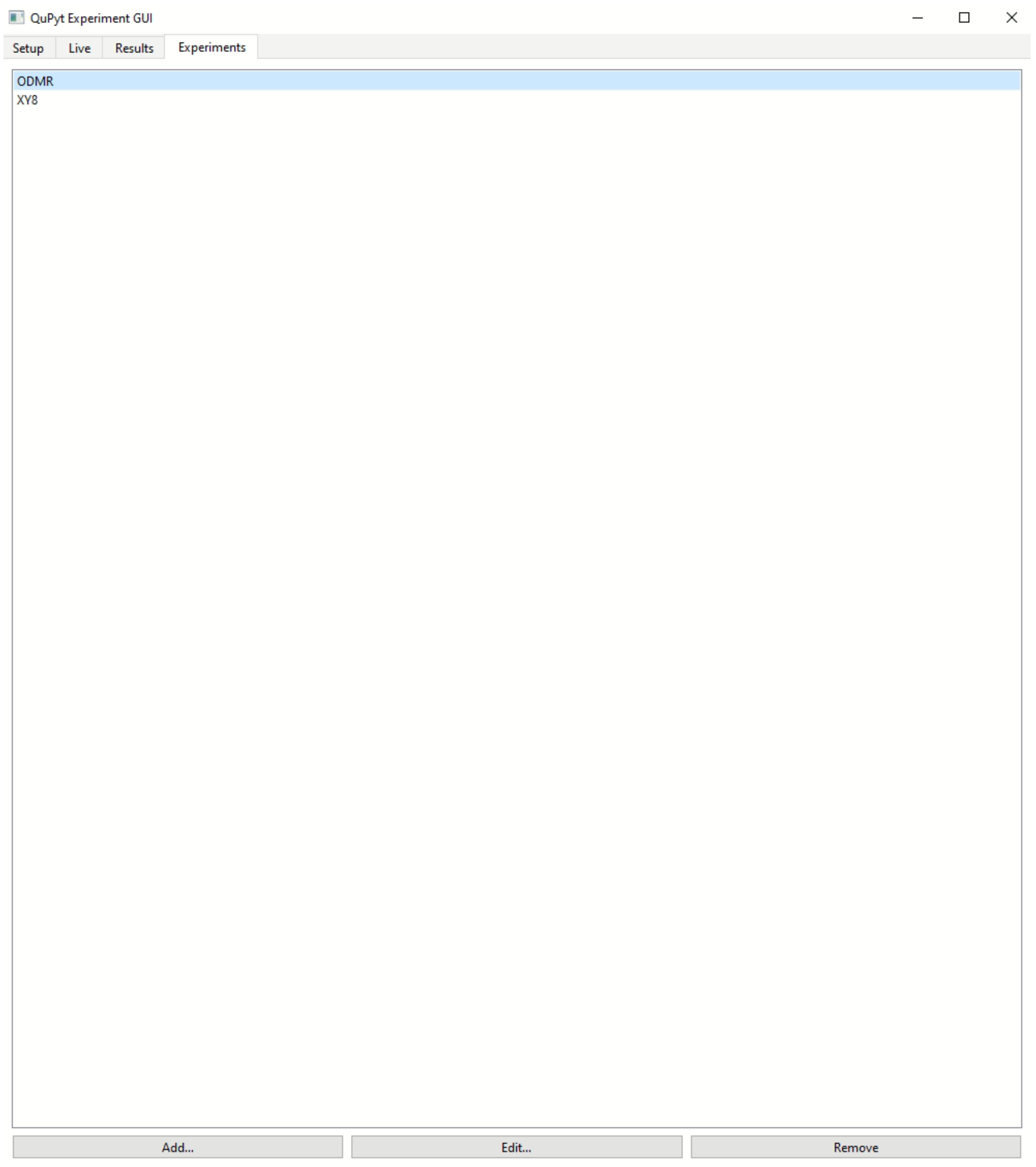


Figure 9: Manage your experiment descriptors: Add, Edit, Remove.

The Experiments tab lists all `*.yaml` descriptors in `GUI/experiments`. You can:

- **Add...**: launch the `ExperimentEditor` dialog to create a new descriptor.
- **Edit...**: modify an existing one.
- **Remove**: delete a descriptor.

7.1 ExperimentEditor Dialog

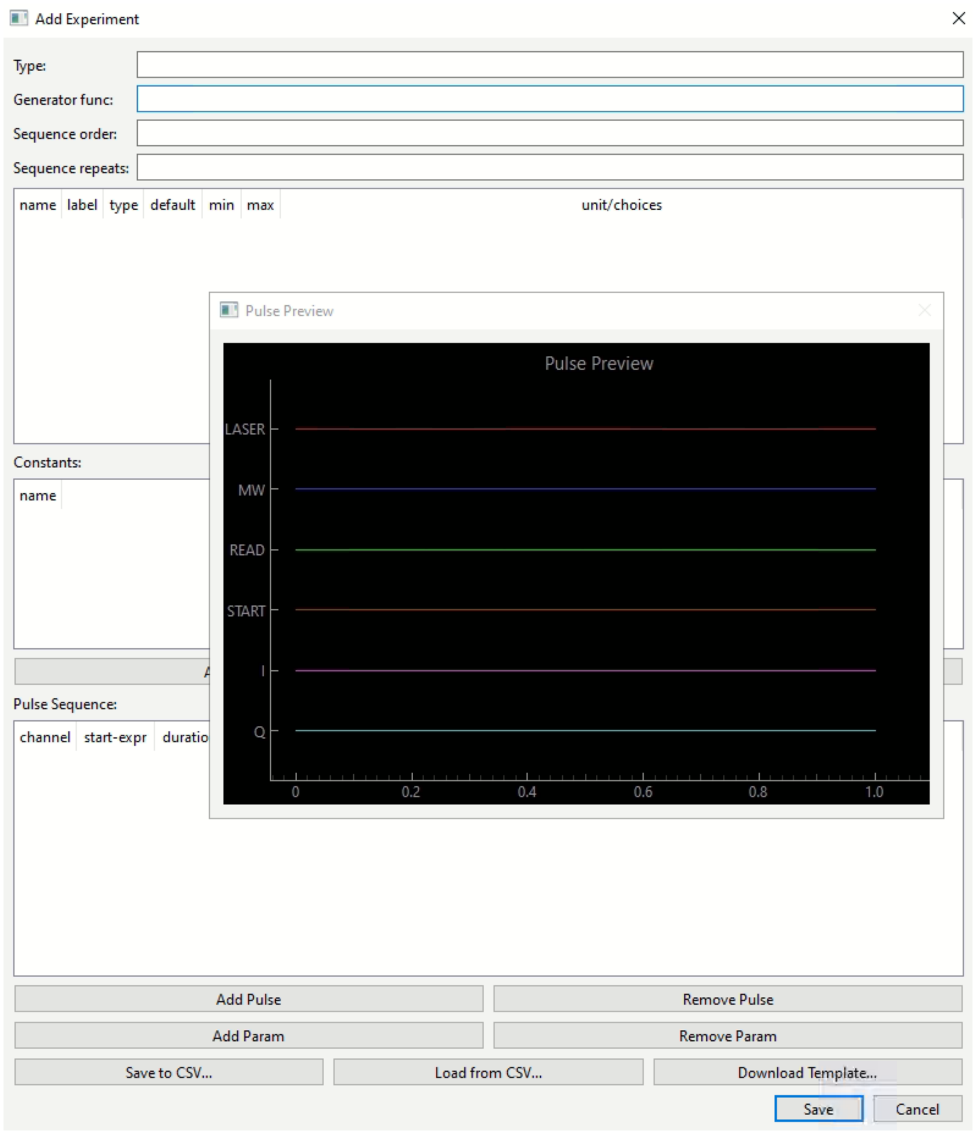


Figure 10: ExperimentEditor: define parameters, constants, and pulse tables.

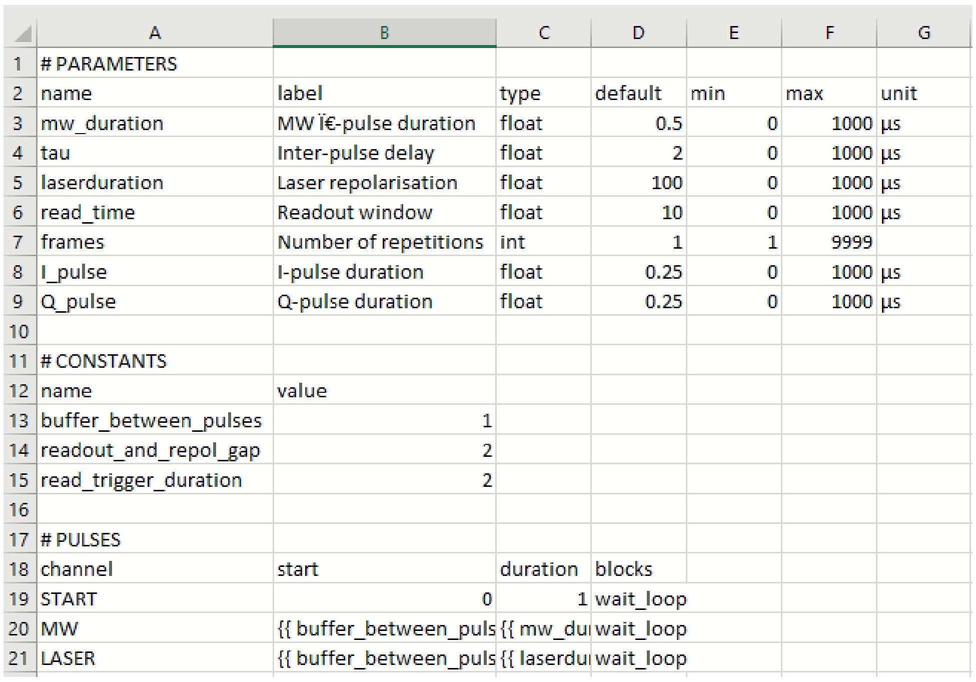
The ExperimentEditor dialog lets you create or modify experiment descriptors, and the underlying pulse sequence definitions, without hand-editing any files. You can:

- Set the **Experiment type** (must be unique).
- Optionally specify a **Pulse-generator function** path.
- Populate the **Parameters** table: each row defines one parameter with columns for `name`, `label`, `type` (`int`, `float`, or `choice`), `default`, `min`, `max`, and `unit` (or comma-separated choices for a `choice` parameter).
- Populate the **Constants** table: name/value pairs for any fixed constants used in your expressions.
- Populate the **Pulse Sequence** table: define each pulse by specifying `channel`, Jinja2 `start` expression, Jinja2 `duration` expression, and a comma-list of sequence `blocks`.

- Fill in **Sequencing order** and **Sequencing repeats** as comma-separated lists.
- Import/export all tables to/from CSV, or download a scaffold template.

Using the CSV template

1. Click **Download Template. . .** to save a commented CSV scaffold.
2. Open the CSV in your editor, fill in your `PARAMETERS`, `CONSTANTS`, and `PULSES` sections.
3. In ExperimentEditor, click **Load from CSV. . .** to import your entries.



	A	B	C	D	E	F	G
1	# PARAMETERS						
2	name	label	type	default	min	max	unit
3	mw_duration	MW Ï€-pulse duration	float	0.5	0	1000	µs
4	tau	Inter-pulse delay	float	2	0	1000	µs
5	laserduration	Laser repolarisation	float	100	0	1000	µs
6	read_time	Readout window	float	10	0	1000	µs
7	frames	Number of repetitions	int	1	1	9999	
8	I_pulse	I-pulse duration	float	0.25	0	1000	µs
9	Q_pulse	Q-pulse duration	float	0.25	0	1000	µs
10							
11	# CONSTANTS						
12	name	value					
13	buffer_between_pulses	1					
14	readout_and_repol_gap	2					
15	read_trigger_duration	2					
16							
17	# PULSES						
18	channel	start	duration	blocks			
19	START	0	1	wait_loop			
20	MW	{{ buffer_between_puls	{{ mw_du	wait_loop			
21	LASER	{{ buffer_between_puls	{{ laserdu	wait_loop			

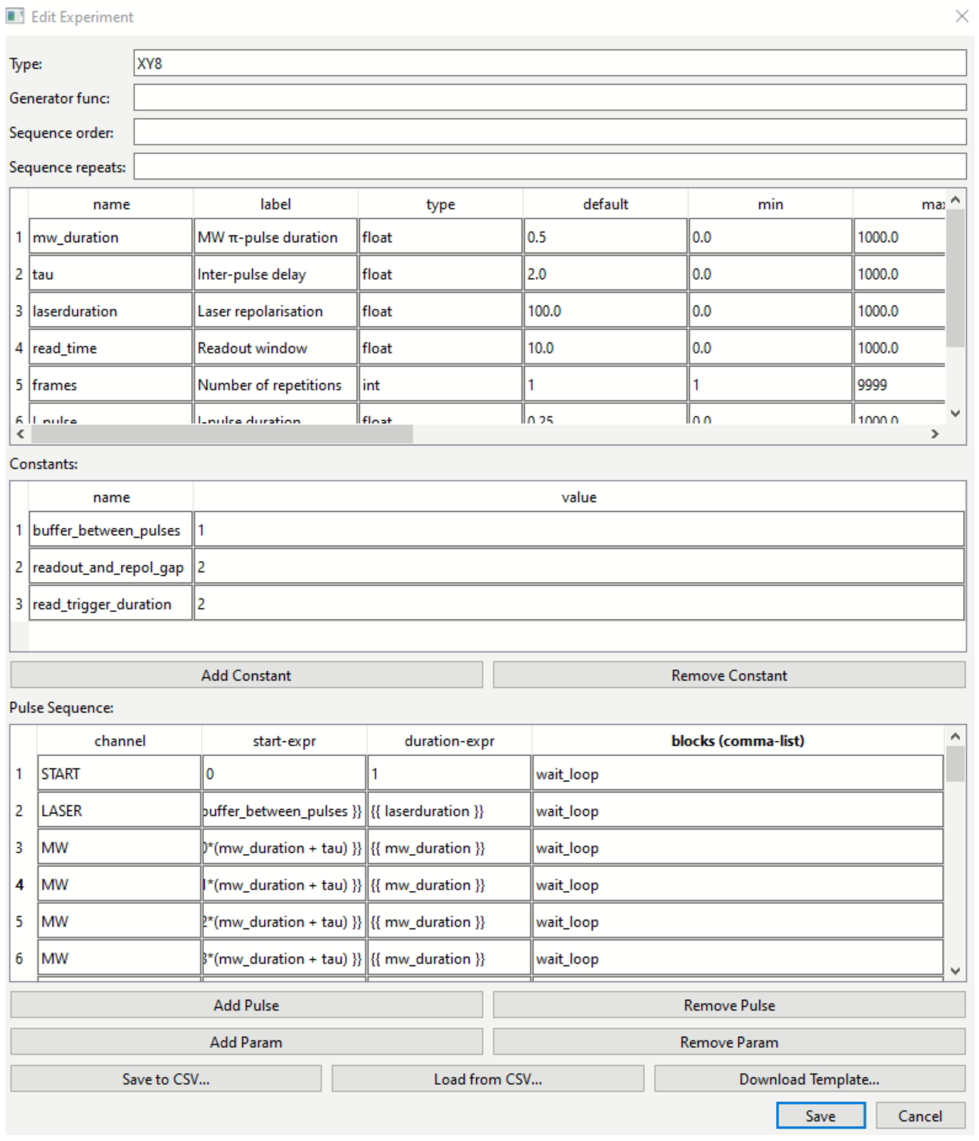
Figure 11: A blank CSV template for creating new experiments.

Quick-start from an existing experiment If you'd like to create a new experiment by modifying an existing one (for example, ODMR):

1. In the Experiments tab, select "ODMR" (or another descriptor), select "Edit..." Then, click **Save to CSV. . . .** This exports the full ODMR descriptor (parameters, constants, pulses) to a CSV file.
2. Open the CSV in your favorite editor, make your adjustments (e.g. tweak pulse durations, add new parameters).
3. Back in the Experiments tab, click **Add. . .** to open a fresh ExperimentEditor.
4. In the dialog, choose **Load from CSV. . .** and select your edited CSV. All tables will be populated with your modified values.
5. Give your new experiment a distinct *Experiment type* name, then click **Save**. Your new YAML descriptor is written and immediately available in the GUI.

Manual entry alternative

- Use the **Add Param / Remove Param** buttons to edit the Parameters table directly.
- Use the **Add Constant / Remove Constant** buttons to manage constants.
- Use **Add Pulse / Remove Pulse** to construct your pulse sequence entries one row at a time.



Type:	XY8
Generator func:	
Sequence order:	
Sequence repeats:	

	name	label	type	default	min	ma
1	mw_duration	MW π-pulse duration	float	0.5	0.0	1000.0
2	tau	Inter-pulse delay	float	2.0	0.0	1000.0
3	laserduration	Laser repolarisation	float	100.0	0.0	1000.0
4	read_time	Readout window	float	10.0	0.0	1000.0
5	frames	Number of repetitions	int	1	1	9999
6	I_pulse	I-pulse duration	float	0.25	0.0	1000.0

Constants:

	name	value
1	buffer_between_pulses	1
2	readout_and_repol_gap	2
3	read_trigger_duration	2

Pulse Sequence:

	channel	start-expr	duration-expr	blocks (comma-list)
1	START	0	1	wait_loop
2	LASER	buffer_between_pulses }}	{{ laserduration }}	wait_loop
3	MW	0*(mw_duration + tau) }}	{{ mw_duration }}	wait_loop
4	MW	1*(mw_duration + tau) }}	{{ mw_duration }}	wait_loop
5	MW	2*(mw_duration + tau) }}	{{ mw_duration }}	wait_loop
6	MW	3*(mw_duration + tau) }}	{{ mw_duration }}	wait_loop

Figure 12: Example ExperimentEditor filled in for an XY8 pulse sequence.

Pulse Preview Window Along with the ExperimentEditor dialog, a separate **Pulse Preview** window automatically appears. This live plot updates in real time as you modify parameters, constants, or pulse-sequence entries, allowing you to visually verify timing, overlap, and overall sequence structure before saving.

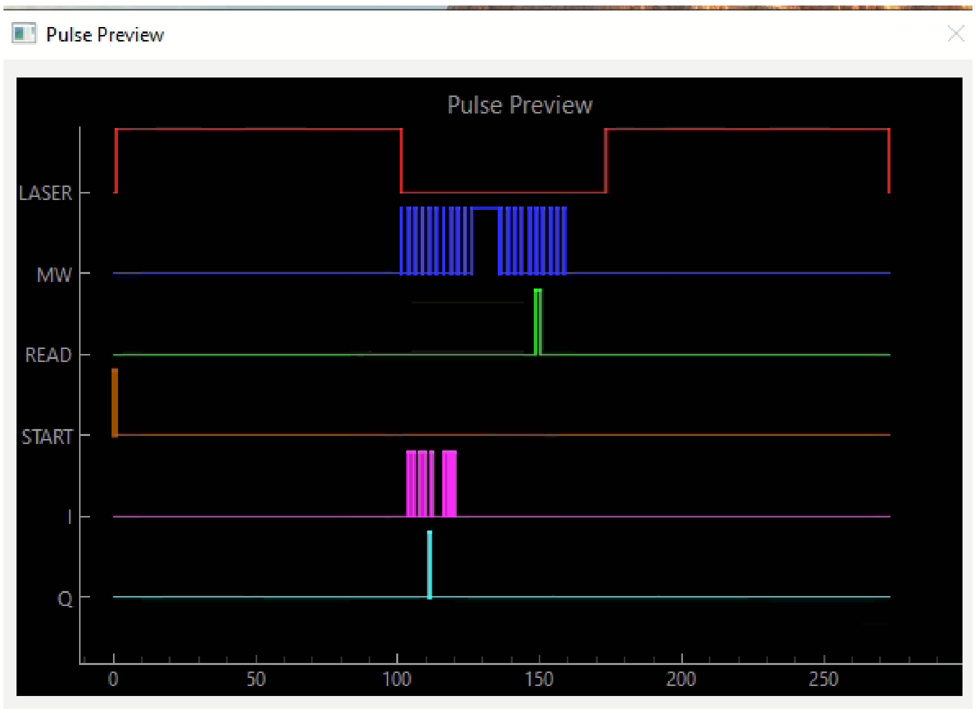


Figure 13: Dynamic pulse-sequence preview—updates instantly as you edit the descriptor.

8 Experiment YAML Explained

After you press **Start**, the GUI writes a YAML file (e.g., `<path-to-output>\ODMR.yaml`) with this structure:

```
experiment_type: ODMR
averages: 5

sensor:
  type: DAQ
  config:
    number_measurements: &nframes 20
    apd_input: "Dev1/ai0"

synchroniser:
  type: PulseBlaster
  channel_mapping:
    MW:    3
    LASER: 2
    READ:  0
    START: 1

dynamic_steps: &n_dynamic_steps 30

dynamic_devices:
  mw_source:
    device_type: WindFreak
    address: COM3
    config:
      frequency: [2800000000.0, 2950000000.0]
      amplitude: [["channel_0", [10.0,10.0]]]

static_devices: {}
```

```
data:
  averaging_mode: spread
  dynamic_steps: *n_dynamic_steps
  compress: false
  reference_channels: 1

ps_path: 'C:\...}\user_pulse_seq.py'

pulse_sequence:
  mw_duration: 550.0
  laserduration: 150.0
  readout_time: 8.0
  referenced_measurements: *nframes
  max_framerate: 10000
```

Line by line (customizable to your hardware):

`experiment_type` Identifier matching your YAML descriptor; appears in the GUI's Experiment dropdown.

`averages` Number of APD readout frames per dynamic step (your DAQ's "batch" size).

`sensor` *type*: your photon-counting or voltage-readout device (e.g. DAQ, oscilloscope, camera).
config: parameters such as `number_measurements` (frames) and input channel(s).

`synchroniser` *type*: your pulse-sequencer hardware (e.g. PulseBlaster, PulseStreamer).
config: device-specific settings.
channel_mapping: maps logical channels (MW, LASER, READ, START, I, Q, etc.) to device lanes.

`dynamic_steps` Number of discrete sweep points (e.g. frequency or delay values) in your scan.

`dynamic_devices` Defines each time-varying instrument (e.g. RF source, laser driver): `device_type`, connection `address`, and `config` block (e.g. frequency range, power/amplitude).

`static_devices` Defines any fixed-parameter hardware (e.g. digital I/O lines, temperature controllers); empty if unused.

`data` Acquisition settings: `averaging_mode` ("sum" vs. "spread"), `reference_channels`, and any compression flags.

`ps_path` Filesystem path to the generated Python pulse-sequence module (e.g. `user_pulse_seq.py`).

`pulse_sequence` Low-level timing parameters for your AWG/PB sequence: pulse durations (MW, LASER, READ, I/Q, etc.), gaps, and internal repetition settings. These values drive the actual TTL timing on your synchroniser.

You must edit APD input, WindFreak COM port, and `ps_path` if your hardware or file locations differ.

9 Pulse-Sequence Module

9.1 Generated Pulse-Sequence Module (user_pulse_seq.py)

When you click **Start**, the GUI writes a Python module called `user_pulse_seq.py`. Below is an example for an ODMR sequence; you can compare it with other experiments (e.g. XY8) by generating their modules via the GUI.

```python
import logging
from qupyt.pulse_sequences.yaml_sequence import YamlSequence
from qupyt import set_up

logging.basicConfig(
    level=logging.DEBUG,
    format="%(asctime)s - %(levelname)s - %(message)s"
)

def generate_sequence(params: dict) -> dict:
    logging.debug("Generating pulse sequence with %s", params)

    # 1) Create a sequence object with total duration in microseconds
    seq = YamlSequence(duration=1000000.0)

    # 2) Add pulses one at a time:
    seq.add_pulse("START",  0.0,    1000.0, sequence_blocks=["wait_loop"
       ↪ ])
    seq.add_pulse("MW",     1000.0, 1000.0, sequence_blocks=["wait_loop"
       ↪ , "block_0"])
    seq.add_pulse("LASER",  3000.0, 5000.0, sequence_blocks=["wait_loop"
       ↪ , "block_0"])
    seq.add_pulse("LASER", 10000.0, 5000.0, sequence_blocks=["wait_loop"
       ↪ , "block_0"])
    seq.add_pulse("READ",   2000.0, 2000.0, sequence_blocks=["block_0"])

    # 3) Define sequencing order & repeats:
    seq.sequencing_order   = ["wait_loop", "block_0"]
    seq.sequencing_repeats = [1, 50]

    # 4) Write out a YAML file
    seq_dir  = set_up.get_seq_dir()
    seq_file = seq_dir / "sequence.yaml"
    seq.write(seq_file)

    logging.info("Pulse sequence generated.")
    return {}
```

Line by line:

`def generate_sequence(params: dict) -> dict:` Entry point called by `qupyt.main`. `params` contains all GUI-collected values.

`seq = YamlSequence(duration=1000000.0)` Instantiates a sequence with a fixed total duration (in microseconds). For ODMR, this might cover all pulses and gaps.

`seq.add_pulse(...)` Adds one TTL pulse:

- Channel name (e.g., “START”, “MW”, “LASER”, “READ”)
- Start time offset (microseconds)
- Pulse duration (microseconds)
- `sequence_blocks`: labels grouping pulses into logical playback blocks

`seq.sequencing_order` Order in which blocks are played back (e.g., first the `wait_loop`, then `block_0`).

`seq.sequencing_repeats` How many times each block is repeated (e.g., 1 repetition of `wait_loop`, then 50 of `block_0`).

`seq_dir = set_up.get_seq_dir()` Locates QuPyt’s pulse-sequence output directory (e.g., `~/.qupyt/sequences`).

`seq.write(seq_file)` Serializes the sequence into `sequence.yaml` for the hardware driver to consume.

`return {}` Optionally returns a dictionary to augment the main experiment YAML; empty here.

Note: This example is tailored to an ODMR experiment. If you switch to XY8 (or any other sequence) and click **Start** in the GUI, you will get a different `user_pulse_seq.py` with channel timings, block names, order, and repeats adjusted to that protocol—yet following the exact same structure.

10 Troubleshooting

- **UnboundLocalError: t_min** Ensure your QuPyt code defines `t_min` at the top of the function. You may check this under `synchronisers.py`.

- **DAQ (NI-DAQ) Configuration & Connectivity Symptoms:** Progress bar stuck at 0%, “could not read DAQ,” timeouts.
  - START trigger not arriving:
    - Verify TTL wiring from your pulse-generator START line to the DAQ’s `start_trig` PFI.
    - In your YAML, confirm `sensor.config.start_trig` matches that terminal.
  - No sample clock ticks / flat output:
    - Check `daq.config.sample_clock_source` and wiring to PFI or onboard clock.
    - Ensure `daq.config.max_samp_rate` is within hardware spec.
    - Confirm `number_measurements` equals triggers per shot.

- **Synchroniser (PulseBlaster / PulseStreamer)**
  Symptoms: No pulses on scope; "Error programming inst1: –91"; triangular markers.
  - Board init failed:
    - Check PB/PS card is seated, powered, drivers installed; `pb_count_boards()` should return ≥ 1.
    - Ensure no other process holds the card.
  - Sequence never runs or parsing errors:
    - Inspect logs from `load_sequence()` for YAML errors.
    - After `write()`, open `~/.qupyt/.../sequence.yaml` to verify content.

- **Pulse-Sequence Parameters & Code Symptoms:** Overlapping-pulse errors; unexpected dead-time; wrong pulse counts.
  - Overlaps: ensure each channel's `start + duration` does not overlap next pulse.
  - Dead-time: adjust your `readout_and_repol_gap` or the two sub-pulse spacing in the Experiments tab.
  - Sequencing repeats off-by-one: verify

    $$\texttt{sequencing\_repeats} = [1, \texttt{int}(n_{\text{ref}}/2)]$$

    matches your intended number of laser/read cycles.

- **GUI fails to launch / ImportError** *Symptoms:* `ModuleNotFoundError` for PyQt6, pyqtgraph, jinja2, scipy, etc.
  Fix:
  1. Activate your virtual environment (`.\venv\Scripts\activate`).
  2. Re-run `pip install -r requirements.txt` or manually install missing packages: `pip install PyQt6 pyqtgraph jinja2 scipy ....`
  3. Ensure `PYTHONPATH` includes your QuPyt root so that `odmr_gui.py` can import local modules.

- **"Start watcher" does nothing / No live output** *Symptoms:* Live tab remains red/idle, no console log appears.
  Fix:
  - Verify that `qupyt.main` is importable: in your venv, run `python -c "import qupyt.main"`.
  - Check that `~/.qupyt/waiting_room` exists and is writable.
  - Review permissions / antivirus settings blocking background processes.
  - Try launching from a terminal to view hidden errors.

- **“Start” button errors out before Live tab** *Symptoms:* MessageBox with “Deployment Error” or traceback in console.
  Fix:
  - Ensure your Desktop YAML path (`<Experiment>.yaml`) can be created—check write permissions.
  - If your experiment descriptor has invalid Jinja expressions, open it in the ExperimentEditor and verify fields.
  - Look for YAML syntax errors in existing `GUI/experiments/*.yaml`.

- **YAML parse / Jinja render failures in pulse preview** *Symptoms:* Pulse-diagram remains blank or errors popup when editing expressions.
  Fix:
  - Inspect your `start` and `duration` expressions for typos—ensure variable names are correct.
  - Wrap string constants in quotes if necessary.
  - Use only arithmetic expressions compatible with Jinja2.

- **Hardware Communication Errors**
  - WindFreak COM port failure
    Fix:
    * Ensure correct COM port (e.g., COM3) is set in the YAML.
    * Use Windows Device Manager or `ls /dev/tty*` (Linux/macOS) to identify ports.
  - DAQ device not found / hangs
    Fix:
    * Check USB connection.
    * Restart the DAQ software or driver.
  - PulseBlaster not initializing
    Fix:
    * Confirm drivers are installed.
    * Check permissions (e.g., `sudo chmod a+rw /dev/pulseblasterX` on Linux).

- **Quick Debug Checklist**
  1. Inspect `~/.qupyt/.../sequence.yaml` — verify pulse entries per block.
  2. Probe each TTL line (START, LASER, READ, SAMPLE_CLK) on the scope.
  3. Check DAQ task triggers and clock sources in code vs. hardware wiring.
  4. Run `python -m qupyt.main` in a terminal to catch hidden errors.
  5. Enable DEBUG logging: `logging.basicConfig(level=logging.DEBUG)` and review console output.
  6. Ensure the modified QuPyt files (`SequenceDesigner.py`, `sensors.py`, and `yaml_sequence.py`) have replaced the originals in your working directory.

11 References

- QuPyT GitHub Repository
- GUI Repository
- PyQt6 Documentation