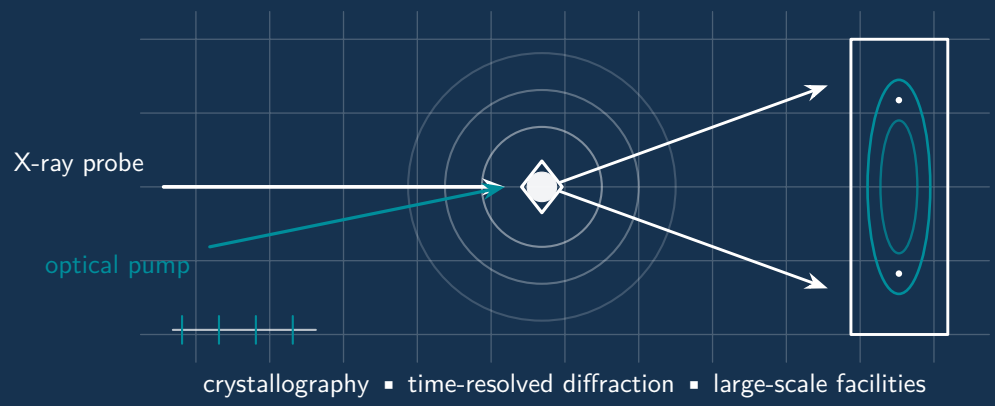
XRDpy

User Manual

Simulation and Analysis



Python toolkit for X-ray diffraction simulation and analysis

Julio Guzman-Brambila

Package version 3.3.0 ▪ Manual version 1.0

July 2026

About this edition

This manual documents the two major XRDpy branches in a single PDF: Simulation and Analysis. The branches share the same distribution and installation but remain computationally independent. Simulation output is not required as input to the Analysis workflows, and experimental Analysis workflows do not depend on running the Simulation branch first.

The text describes XRDpy version 3.3.0. The project is named XRDpy on GitHub and Zenodo, distributed on PyPI as `trxrdpy`, and imported in Python as `trxrdpy`.

Scientific conventions

Instrument-specific geometry, detector orientation, azimuthal convention, and reference choices should always be checked against the user's experiment. The examples in this manual show the package interface and the expected reproducibility record; they do not replace beamline- or sample-specific validation.

Project identifiers

Repository	https://github.com/julioguzmanb/XRDpy
Concept DOI	https://doi.org/10.5281/zenodo.18634909
PyPI distribution	`trxrdpy`
Python import	`import trxrdpy`
License	Creative Commons Attribution 4.0 International (CC BY 4.0)

Contributors

J. Guzman-Brambila	Univ Rennes, CNRS, Institut de Physique de Rennes (IPR) – UMR 6251, 35000 Rennes, France; Nantes Université, CNRS, Institut des Matériaux de Nantes Jean Rouxel (IMN), Nantes, France.
M. Lorenc	Univ Rennes, CNRS, Institut de Physique de Rennes (IPR) – UMR 6251, 35000 Rennes, France.
E. Janod	Nantes Université, CNRS, Institut des Matériaux de Nantes Jean Rouxel (IMN), Nantes, France.
C. Mariette	ESRF, The European Synchrotron, 71 Avenue des Martyrs, CS40220, 38043 Grenoble Cedex 9, France.

How to read this manual

Chapter 1 defines the package scope. Chapter 2 covers installation and a minimal verification. Part I is self-contained: it introduces the simulation model, explains its inputs and conventions, develops powder and single-crystal workflows, describes the GUI, and ends with reference and troubleshooting material.

Readers interested only in powder diffraction can proceed from Chapters 3–4 directly to Chapter 6. Readers interested in oriented crystals should also read Chapters 7 and 8. The API reference in Chapter 12 is selective: it documents the stable, user-facing surface rather than every internal helper.

Manual organization

The manual is organized into three parts.

Front matter and general introduction

- Title, package version, author, repository, DOI, citation, and license
- Purpose, intended readership, scope, and limitations
- Installation from PyPI or source and first verification
- Package naming, imports, optional dependencies, and GUI launch commands
- Complete table of contents

Part I: Simulation

Simulation model and conventions
: Beam, lattice, reflections, detector, units, coordinate systems, rotations, transforms, and coordinate-system diagnostics.

Crystallographic input and lattice construction
: Explicit cell parameters, CIF parsing, reciprocal lattice, allowed reflections, Bragg-condition testing, and sample orientation.

Detector definition and calibration
: Manual detectors, binning, pyFAI registry detectors, PONI files, orientation changes, laboratory grids, and validation checks.

Polycrystalline diffraction
: One-dimensional CIF-based patterns, two-dimensional detector rings, three-dimensional diffraction cones, PONI input, interpretation, and reproducible figure generation.

Diffractometers and motor-chain geometry
: Motors, local and laboratory frames, Euler and kappa geometries, custom instruments, serialization, and geometry inspection.

Single-crystal diffraction
: Legacy and geometry-aware 2D/3D simulation; explicit reflections; legacy, parameter, and named-motor sample scans; legacy and geometry-aware detector scans; fixed-energy inverse targeting; result objects; and generated figures.

Plotting, GUI, and examples
: Plot families, saving outputs, Simulation GUI operation, screenshots, complete programmatic examples, and reproducibility records.

Reference and validation
: Public Simulation API, return-object behavior, validation hierarchy, troubleshooting, performance, and reproducibility.

Part II: Analysis

Analysis branch overview and data organization
Independent Analysis scope, supported facilities, shared and facility-specific layers, path conventions, standardized outputs, and overall data flow.

Facility-specific preparation
ESRF ID09, Max IV FemtoMAX, and SPring-8 SACLA data structures; data reduction; homogenized images; ID09-specific processing; external software; legacy environments; and HPC submission.

Calibration, diagnostics, and azimuthal integration
Detector calibration, masks, polarization, detector/cake diagnostics, q-band azimuthal-distribution analysis, shared 2D integration, ID09 integration, and generation of standardized one-dimensional `xy` patterns.

Shared downstream analysis
Pattern visualization, differential analysis, peak fitting, export, uncertainty and quality checks, and reproducibility.

Analysis GUI and reference
Session preparation, facility selection, calibration, pattern creation, viewer, differential, fitting tabs, API reference, and troubleshooting.

Part III: Shared reference and appendices

- Project versioning, compatibility, contributions, issues, citation, license, and acknowledgments
- Glossary of diffraction and package terminology
- Simulation package/module map
- Figure manifest and revision record

Contents

About this edition i

How to read this manual ii

Manual organization iii

1 Introduction 1

1.1 What XRDpy is . . . 1

1.2 Purpose of the Simulation branch . . . 1

1.3 Intended readership . . . 1

1.4 Scope and limitations . . . 2

2 Installation and first verification 3

2.1 Requirements . . . 3

2.2 Install from PyPI . . . 3

2.3 Verify the installation . . . 3

2.4 Launch the Simulation GUI . . . 4

2.5 Imports used in this manual . . . 4

I Simulation 5

3 Simulation model and conventions 6

3.1 Conceptual model . . . 6

3.2 Quantities and units . . . 6

3.3 Laboratory and detector axes . . . 7

3.4 Rotations and transforms . . . 7

4 Crystallographic input and lattice construction 8

4.1 Explicit lattice parameters . . . 8

4.2 CIF input . . . 8

4.3 Generating reflections . . . 9

4.4 Testing the Bragg condition . . . 9
4.5 Orienting the crystal . . . 9

5 Detector definition and calibration 11
5.1 Manual detector . . . 11
5.2 Binning . . . 12
5.3 pyFAI detector registry . . . 12
5.4 PONI calibration files . . . 12
5.5 Changing orientation . . . 13
5.6 Detector validation checklist . . . 13

6 Polycrystalline diffraction 14
6.1 Available workflows . . . 14
6.2 One-dimensional pattern from CIF . . . 14
6.3 Three-dimensional powder cones . . . 15
6.4 Two-dimensional detector projection . . . 16
6.5 Manual detector from experimental geometry . . . 17
6.6 Using a PONI detector . . . 19
6.7 Interpreting powder output . . . 19

7 Diffractometers and motor-chain geometry 20
7.1 Why use motor chains . . . 20
7.2 Lab-fixed and local axes . . . 20
7.3 Predefined Euler geometry . . . 20
7.4 Predefined kappa geometry . . . 21
7.5 Custom geometry . . . 21
7.6 Inspecting and serializing geometry . . . 22

8 Single-crystal diffraction 23
8.1 High-level simulation . . . 23
8.2 Geometry-aware simulation . . . 24
8.3 Explicit additional reflections . . . 25
8.4 Searching sample orientations . . . 26
8.4.1 Legacy two-angle convenience workflow . . . 26
8.4.2 Named-motor two-dimensional scan . . . 27
8.5 Scanning two parameters . . . 28
8.6 Collecting Bragg reflections with detector rotations . . . 29
8.6.1 Legacy detector Euler scan . . . 29

8.6.2 Detector motor-chain and geometry scans . . . 30
8.7 Fixed-energy inverse targeting . . . 31

9 Plotting and result inspection 34
9.1 Plot families . . . 34
9.2 Separating calculation from presentation . . . 34
9.3 Plot interpretation . . . 34
9.4 Saving figures . . . 35

10 Simulation graphical interface 36
10.1 Purpose and architecture . . . 36
10.2 Launching the interface . . . 36
10.3 Polycrystalline tab . . . 36
10.4 Single-crystal tab . . . 38
10.5 Matrix-rotation helper . . . 40
10.6 Sessions and logs . . . 40
10.7 GUI operating sequence . . . 41

11 Complete simulation examples 42
11.1 Example data files . . . 42
11.2 Example A: lattice and detector inspection . . . 42
11.3 Example B: powder detector prediction . . . 43
11.4 Example C: manual detector from calibration values . . . 45
11.5 Example D: geometry-aware single crystal . . . 47
11.6 Example E: reproducible parameter record . . . 48

12 Simulation API reference 50
12.1 High-level modules . . . 50
12.2 Core classes . . . 50
12.3 Geometry factories . . . 51
12.4 Common utility functions . . . 51
12.5 Return objects and cached state . . . 52

13 Validation, reproducibility, and troubleshooting 53
13.1 Validation hierarchy . . . 53
13.2 Common problems . . . 53
13.2.1 Import fails . . . 53
13.2.2 CIF cannot be read . . . 53

13.2.3 No reflections satisfy Bragg condition . . . 53
13.2.4 No reflections appear on the detector . . . 53
13.2.5 Powder rings are displaced or mirrored . . . 54
13.2.6 Geometry scan is unexpectedly slow . . . 54
13.2.7 Changing angles has no effect . . . 54
13.2.8 Unexpected lattice state after repeated rotations . . . 54
13.3 Reproducibility record . . . 54
13.4 Known interface cautions . . . 54

II Analysis 55

14 Analysis branch overview 56
14.1 Scope of the Analysis branch . . . 56
14.2 Supported facilities . . . 56
14.3 Analysis data flow . . . 57
14.4 Programmatic and GUI routes . . . 58

15 Data organization and file conventions 59
15.1 Experiment root and analysis root . . . 59
15.2 Scan specifications and tags . . . 60
15.3 Standardized intermediate products . . . 60
15.4 Reproducibility records . . . 60

16 Facility-specific 2D image production 61
16.1 Max IV FemtoMAX . . . 61
16.1.1 FemtoMAX delay-scan 2D image production . . . 62
16.2 ESRF ID09 . . . 67
16.2.1 ID09 2D image production example . . . 67
16.3 SPring-8 / SACLA . . . 68
16.3.1 SACLA 2D image production example . . . 69
16.4 Facility comparison . . . 69

17 Calibration, masks, and detector diagnostics 71
17.1 Calibration inputs . . . 71
17.2 Detector and cake diagnostics . . . 71
17.3 Azimuthal windows . . . 72
17.4 Calibration peak fitting . . . 74
17.5 Cake azimuthal-distribution analysis . . . 75

18 Azimuthal integration and standardized 1D patterns **78**
18.1 Shared 2D integration workflow . . . 78
18.2 Delay-scan patterns . . . 79
18.3 Fluence-scan patterns . . . 80
18.4 Quality checks before fitting . . . 81

19 Differential analysis **82**
19.1 Peak and background specifications . . . 82
19.2 Delay differential traces . . . 82
19.3 Fluence differential traces . . . 85
19.4 Multi-experiment comparison . . . 86

20 Peak fitting **87**
20.1 Fitting philosophy . . . 87
20.2 Delay peak fitting . . . 87
20.3 Fluence peak fitting . . . 91
20.4 Fit outputs and diagnostics . . . 92

21 Facility-specific worked analysis examples **93**
21.1 Max IV FemtoMAX . . . 93
21.1.1 FemtoMAX single-delay-scan example . . . 93
21.2 SPring-8 / SACLA . . . 99
21.2.1 SACLA delay-scan examples from cached one-dimensional data . . . 99
21.2.2 SACLA fluence-scan examples from cached one-dimensional data . . . 106

22 Analysis GUI workflows **114**
22.1 GUI workflow map . . . 114
22.2 Session and facility setup . . . 114
22.3 Preparation tab . . . 115
22.4 Calibration and pattern tabs . . . 116
22.5 Differential and fitting tabs . . . 118

23 Analysis API reference and troubleshooting **120**
23.1 User-facing API map . . . 120
23.2 Common analysis failures . . . 120
23.3 Publication checklist . . . 121

III Shared reference and appendices **122**

24 Project and contribution information **123**

24.1 Versioning and compatibility 123

24.2 Reporting issues 123

24.3 Citation, license, and acknowledgments 123

Bibliography **124**

A Glossary **126**

B Package map **127**

C Citation and license **128**

Chapter 1

Introduction

1.1 What XRDpy is

XRDpy is a Python toolkit for X-ray diffraction (XRD) simulation and beamline-dependent analysis, with particular attention to time-resolved and pump–probe experiments. It combines two broad capabilities in one distribution:

- **Simulation**: crystal and detector geometry, powder rings and patterns, single-crystal reflections, orientation searches, diffractometer motor chains, and graphical workflows.
- **Analysis**: facility-specific data preparation, azimuthal integration, calibration, differential analysis, and fitting.

These branches share a project and installation but are computationally independent. A user may employ the Simulation branch without adopting the Analysis branch, or use the Analysis branch on experimental data without first running a simulation.

1.2 Purpose of the Simulation branch

The Simulation branch connects crystallographic descriptions, an incident X-ray beam, and detector geometry. Its main tasks are to:

- construct a lattice from explicit cell parameters or a CIF file;
- generate space-group-allowed reflections and reciprocal-space vectors;
- test reflections against a finite incident-energy bandwidth;
- represent manually specified, pyFAI-registered, or PONI-calibrated area detectors;
- visualize polycrystalline cones and detector rings;
- visualize single-crystal diffraction in three dimensions and on the detector plane;
- scan sample or detector rotations for Bragg conditions; and
- describe Euler, kappa, and custom motor-chain geometries.

The package offers both high-level functions for common tasks and lower-level objects for workflows that require explicit control.

1.3 Intended readership

This manual assumes familiarity with Python and the basic language of diffraction: Miller indices, reciprocal space, Bragg's law, detector distance, and beam center. It does not assume familiarity with the internal structure of XRDpy. The geometry chapters are also intended for beamline scientists or advanced users who need to encode real motor axes.

1.4 Scope and limitations

The Simulation branch is a geometrical and crystallographic toolkit. Results depend on the supplied lattice, detector calibration, coordinate conventions, energy, bandwidth, and orientation. A simulated detector hit is not by itself a prediction of experimentally observable intensity. Sample texture, absorption, extinction, instrument resolution, background, polarization, and other effects require explicit treatment where relevant.

The one-dimensional powder workflow can calculate crystallographic intensities from a CIF and supports selected corrections and peak convolution. The two- and three-dimensional powder workflows primarily position labeled reflections geometrically. Single-crystal routines identify reciprocal-lattice directions compatible with the specified beam and detector geometry.

Chapter 2

Installation and first verification

2.1 Requirements

XRDpy 3.3.0 requires Python 3.10 or newer. Its core dependencies include NumPy, SciPy, Matplotlib, pandas, h5py, lmfit [16], pyFAI [9, 10], fabio [11, 12], PyCifRW [13], and xrayutilities [14, 15]. The graphical interfaces additionally require PyQt5 and mplcursors.

2.2 Install from PyPI

Install the base distribution with:

```
python -m pip install trxrdpy
```

Install the GUI dependencies when graphical use is required:

```
python -m pip install "trxrdpy[gui]"
```

The optional `analysis` extra is not required for the Simulation branch. A complete development installation from a source checkout is:

```
git clone https://github.com/julioguzmanb/XRDpy.git
cd XRDpy
python -m pip install -e ".[analysis,gui]"
```

2.3 Verify the installation

Run a small import and unit-conversion check:

```
import trxrdpy
from trxrdpy.simulation import utils

energy_ev = 10000.0
wavelength_m = utils.energy_to_wavelength(energy_ev)

print(trxrdpy.__name__)
print(f"{wavelength_m * 1e10:.4f} angstrom")
```

A 10 keV photon has a wavelength of approximately 1.2398 Å. The exact printed value follows the physical constants used by the installed package.

2.4 Launch the Simulation GUI

The installed console entry point is:

```
trxrdpy-sim-gui
```

The equivalent module invocation is:

```
python -m trxrdpy.simulation.gui.main_window
```

On a remote system, a graphical display or an appropriate forwarding mechanism is required.

2.5 Imports used in this manual

```
from trxrdpy.simulation import (
    cif,
    detector,
    diffractometers,
    geometry,
    polycrystalline,
    sample,
    single_crystal,
    utils,
)
```

Importing individual modules makes examples explicit and avoids relying on internal names that are not part of the public interface.

Part I

Simulation

Chapter 3

Simulation model and conventions

3.1 Conceptual model

A simulation combines four descriptions:

1. a beam, defined by photon energy and a relative energy bandwidth;
2. a sample lattice, defined by cell parameters, symmetry, and orientation;
3. reciprocal-lattice reflections, derived from the lattice or provided explicitly; and
4. a detector plane, defined by pixel geometry, position, and orientation.

The high-level functions assemble these descriptions into an `Experiment`. Lower-level use constructs a `LatticeStructure` and `Detector` directly, then supplies them to `Experiment` or calls their methods.

3.2 Quantities and units

Quantity	Unit	Notes
Photon energy	eV	High-level defaults are typically `10e3`.
Wavelength	m	Utility and geometry functions use SI meters. Convert to Å for display if desired.
Cell lengths	Å	Values `a`, `b`, and `c`.
Reciprocal magnitude q	$Å^{-1}$	Reflection range and powder input.
d **spacing**	Å	Related by $q = 2\pi/d$.
Angles	degrees	Public rotation arguments and motor-angle maps use degrees.
PONI rotations	radians in file	Converted to degrees by the PONI reader when building a detector.
Detector distance	m	Sample-to-detector distance.
Pixel size	m	Unbinned size on construction; effective size is stored after binning.
PONI coordinates	m	Axis 1 is slow/vertical; axis 2 is fast/horizontal.
Energy bandwidth	percent	Full relative bandwidth $\Delta E/E$, not a fractional value.

For elastic scattering,

$$q = \frac{4\pi \sin\theta}{\lambda}, \qquad d = \frac{2\pi}{q}, \qquad 2\theta = 2\arcsin\left(\frac{q\lambda}{4\pi}\right). \tag{3.1}$$

The user-facing utilities `energy_to_wavelength` and `wavelength_to_energy` convert the beam quantity. The functions `q_to_two_theta` and `d_to_two_theta` convert diffraction coordinates.

3.3 Laboratory and detector axes

The unrotated detector plane is placed at positive laboratory x, with the incident beam directed along the detector-distance axis. Within the detector representation:

- PONI axis 1 is the slow image dimension and vertical direction;
- PONI axis 2 is the fast image dimension and horizontal direction;
- `pxsize_v` and `num_pixels_v` refer to axis 1;
- `pxsize_h` and `num_pixels_h` refer to axis 2; and
- binning is ordered `(horizontal, vertical)`.

In the detector grid, the horizontal coordinate includes a sign reversal relative to increasing axis-2 pixel index. This is a plotting and laboratory-frame convention, not a statement about every detector file format.

> **Coordinate-system check**
>
> Before comparing a simulated detector image with experimental data, confirm the beam direction, image origin, axis order, handedness, and any display flips. A correct PONI calibration cannot compensate for a mismatched image convention.

3.4 Rotations and transforms

Legacy interfaces describe sample and detector orientation with three Euler angles and a rotation-order string. The accepted order is a permutation of `xyz`; detector defaults commonly use `zyx`, while sample defaults commonly use `xyz`.

Modern interfaces also accept:

- a 3×3 rotation matrix;
- a 4×4 homogeneous transform;
- an `AxisRotation` or `RotationChain`;
- a named `MotorChain`; or
- a `DiffractometerGeometry` with independent sample and detector chains.

A custom detector transform takes precedence over legacy Euler values until `clear_transform()` is called. For a sample, `mode="absolute"` applies the requested orientation to the initial lattice, while `mode="relative"` starts from the current orientation.

Chapter 4

Crystallographic input and lattice construction

4.1 Explicit lattice parameters

`LatticeStructure` infers symmetry-constrained parameters from the International Tables space-group number. For example, a cubic cell needs only `a`; a hexagonal cell needs `a` and `c`; a triclinic cell needs all three lengths and all three angles.

```
from trxrdpy.simulation.sample import LatticeStructure

silicon = LatticeStructure(space_group=227, a=5.431)
print(silicon.phase)              # cubic
print(silicon.crystal_orientation)
```

The initial orientation matrix contains the direct-lattice vectors as rows. It is derived from the supplied cell unless `initial_crystal_orientation` is supplied explicitly as a 3×3 array.

4.2 CIF input

A CIF file can supply the unit cell, International Tables space-group number, and fractional atom positions:

```
from trxrdpy.simulation.cif import Cif
from trxrdpy.simulation.sample import LatticeStructure

cif_data = Cif("sample.cif")
print(cif_data.get_lattice_parameters())
print(cif_data.get_space_group())
print(cif_data.get_atom_positions())

lattice = LatticeStructure(cif_file_path="sample.cif")
```

The reader uses PyCifRW for CIF parsing [13] and selects the first CIF data block containing six present, non-zero cell parameters. Parenthesized uncertainty suffixes such as `5.431(2)` are removed during numeric parsing. Fractional coordinates must lie in the inclusive interval $[0, 1]$.

CIF requirements

The current reader expects `_space_group_IT_number` and the atom-site label and fractional-coordinate columns. A syntactically valid CIF that omits these fields may still be unsuitable for this workflow. Inspect imported values before simulation.

4.3 Generating reflections

Reflection generation is deliberately explicit in the lower-level API:

```
lattice = LatticeStructure(space_group=227, a=5.431)
lattice.calculate_reciprocal_lattice()
lattice.create_possible_reflections(qmax=6.0)
lattice.calculate_q_hkls()
lattice.calculate_d_hkls()

print(lattice.allowed_hkls[:5])
print(lattice.q_hkls[:5])
print(lattice.d_hkls[:5])
```

`create_possible_reflections(qmax)` uses the xrayutilities space-group lattice [14, 15] to retain allowed Miller indices up to a positive reciprocal-space limit in Å^{-1}. The derived arrays are cached on the object. Recalculate them after changing any lattice property outside the provided rotation methods.

4.4 Testing the Bragg condition

```
from trxrdpy.simulation.utils import energy_to_wavelength

wavelength = energy_to_wavelength(15000.0)
lattice.check_Bragg_condition(
    wavelength=wavelength,
    e_bandwidth=1.5,
)

print(lattice.hkls_in_Bragg_condition)
```

The bandwidth is the full relative energy spread in percent. A larger value broadens the accepted Ewald-shell region and can admit more reflections. It should represent the intended source or effective simulation tolerance rather than being increased merely to force a desired reflection.

4.5 Orienting the crystal

Legacy Euler orientation is concise:

```
lattice.apply_rotation(
    rotx=5.0,
    roty=-2.0,
    rotz=12.0,
    mode="absolute",
)
```

For a direct rotation matrix or homogeneous transform:

```
import numpy as np
from trxrdpy.simulation import utils

rotation = utils.axis_angle_to_matrix(
    axis=[0, 0, 1],
    angle=30.0,
```

```
)
lattice.apply_transform(rotation, mode="absolute")
```

The rotation helpers validate matrix shape and, where applicable, rotation-matrix properties. Prefer a named motor chain when angles correspond to real instrument motors; this preserves their meaning and application order.

Chapter 5

Detector definition and calibration

5.1 Manual detector

The manual route requires pixel sizes, pixel counts, distance, and PONI coordinates:

```
from trxrdpy.simulation.detector import Detector

det = Detector(
    detector_type="manual",
    pxsize_h=75e-6,
    pxsize_v=75e-6,
    num_pixels_h=1024,
    num_pixels_v=1024,
    dist=0.30,
    poni1=0.0384,
    poni2=0.0384,
    rotx=0.0,
    roty=0.0,
    rotz=0.0,
    rotation_order="zyx",
    binning=(1, 1),
)
det.calculate_lab_grid()
print(det.lab_grid.shape)
```

The grid shape is `(num_pixels_h, num_pixels_v, 3)`. Large detectors therefore allocate substantial memory; use realistic binning or smaller dimensions while developing a workflow.

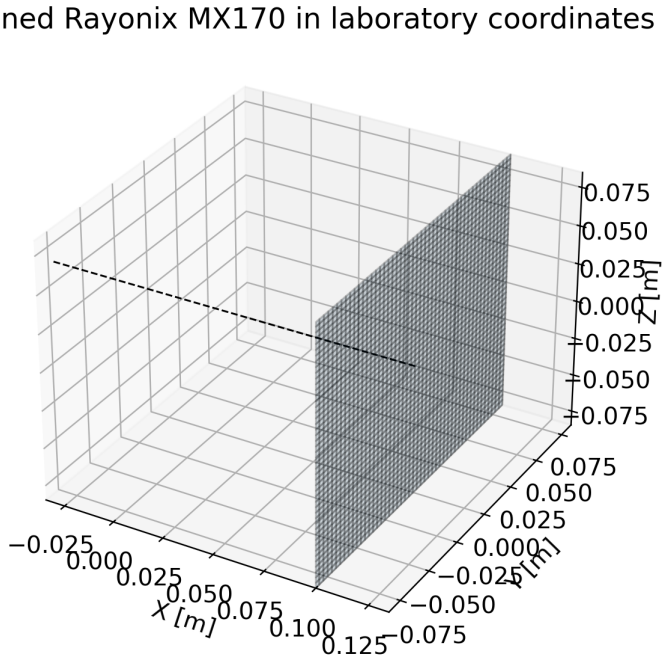


Figure 5.1: Three-dimensional laboratory-coordinate grid for a 16 × 16 binned Rayonix MX170 detector at 0.10 m sample distance.

5.2 Binning

Constructor pixel sizes are unbinned. With `binning=(H,V)`, stored horizontal and vertical pixel sizes are multiplied by their respective factors, while stored pixel counts are divided by them. The overall detector extent is preserved subject to integer truncation.

5.3 pyFAI detector registry

A detector name accepted by `pyFAI.detectors.detector_factory` can replace manual pixel geometry. XRDpy uses pyFAI for detector models, PONI geometry, masking, and azimuthal regrouping in the simulation and analysis workflows [9, 10]:

```
det = Detector(
    detector_type="rayonixmx170",
    dist=0.10,
    poni1=0.085,
    poni2=0.085,
    binning=(16, 16),
)
```

The registry supplies pixel sizes and maximum shape. Distance, PONI coordinates, rotations, and binning remain experiment-specific. The Rayonix MX170 example is binned for a fast diagnostic plot while preserving the physical detector size.

5.4 PONI calibration files

The PONI route loads detector geometry from a pyFAI calibration file:

```
det = Detector(
    detector_type="poni",
    poni_file="calibration.poni",
)
det.calculate_lab_grid()
```

The reader extracts distance, PONI coordinates, rotations, wavelength metadata, detector information, pixel sizes, and detector shape when available. Both JSON and Python-literal forms of `Detector_config` are accepted. Values supplied by the PONI file override explicit detector geometry arguments.

PONI axis mapping is:

PONI field	XRDpy name	Meaning
`PixelSize1` / `pixel1`	`pxsize_v`	slow, vertical
`PixelSize2` / `pixel2`	`pxsize_h`	fast, horizontal
shape index 0	`num_pixels_v`	vertical pixels
shape index 1	`num_pixels_h`	horizontal pixels
`Poni1`	`poni1`	vertical coordinate
`Poni2`	`poni2`	horizontal coordinate

5.5 Changing orientation

Euler angles can be updated after construction:

```
det.set_rotation_angles(roty=3.0, rotation_order="zyx")
det.calculate_lab_grid()
```

A custom transform can instead be attached:

```
rotation = utils.axis_angle_to_matrix([0, 1, 0], 3.0)
det.set_transform(rotation)
det.calculate_lab_grid()

# Restore the legacy Euler path later.
det.clear_transform()
```

Changing an angle or transform does not automatically recalculate the detector grid. Call `calculate_lab_grid()` before plotting or using the updated geometry.

5.6 Detector validation checklist

Before interpreting simulated hits, verify:

- pixel sizes and counts correspond to the unbinned detector input;
- binning order is horizontal then vertical;
- distance and PONI values are in meters;
- PONI axes match the image storage order;
- rotations use the intended order and sign convention;
- the calibration belongs to the relevant detector configuration; and
- the resulting lab grid and beam-center position are physically plausible.

Chapter 6

Polycrystalline diffraction

6.1 Available workflows

The high-level powder interface exposes three complementary views:

`simulate_1d` Build a powder pattern from a CIF, calculate discrete peaks, and optionally convolve them into a continuous profile.
`simulate_2d` Project reflection cones onto an area detector and plot labeled rings.
`simulate_3d` Plot the experiment and diffraction cones in laboratory coordinates.

The 1D workflow derives reflections and intensities from crystallographic information. The 2D and 3D workflows accept explicit q or d positions with corresponding Miller-index labels.

6.2 One-dimensional pattern from CIF

The distributable manual example uses the room-temperature corundum phase of V_2O_3 supplied in `cif_files/V2O3_R-3c_room_T.cif`. Keeping crystallographic inputs beside the manual makes examples executable and records the exact structure used to generate each result. For publication or redistribution, retain the CIF citation metadata and confirm that the source permits redistribution.

```
from trxrdpy.simulation import polycrystalline

result = polycrystalline.simulate_1d(
    cif_file_path="cif_files/V2O3_R-3c_room_T.cif",
    qmax=6.0,
    energy=15000.0,
    x_axis="q",
    include_lorentz_polarization=True,
    include_multiplicity=False,
    atom_positions=True,
    step=0.002,
    fwhm=0.04,
    convolve=True,
    normalize=True,
    plot_result=True,
)

q = result["x"]
intensity = result["I"]
peaks = result["peaks"]
```

Set `x_axis="two_theta"` for an angular axis. The continuous profile uses a Gaussian broadening width controlled by `fwhm`; its unit follows the selected horizontal axis. The `step` parameter sets

sampling on the same axis. If `convolve=False`, consult the returned discrete peak table rather than assuming the continuous arrays are populated in the same way.

`atom_positions=True` requests construction of the structure-factor crystal from parsed atomic positions. This requires suitable CIF atom labels and coordinates. Always inspect several peak positions and relative intensities against a trusted reference before using the simulated profile quantitatively.

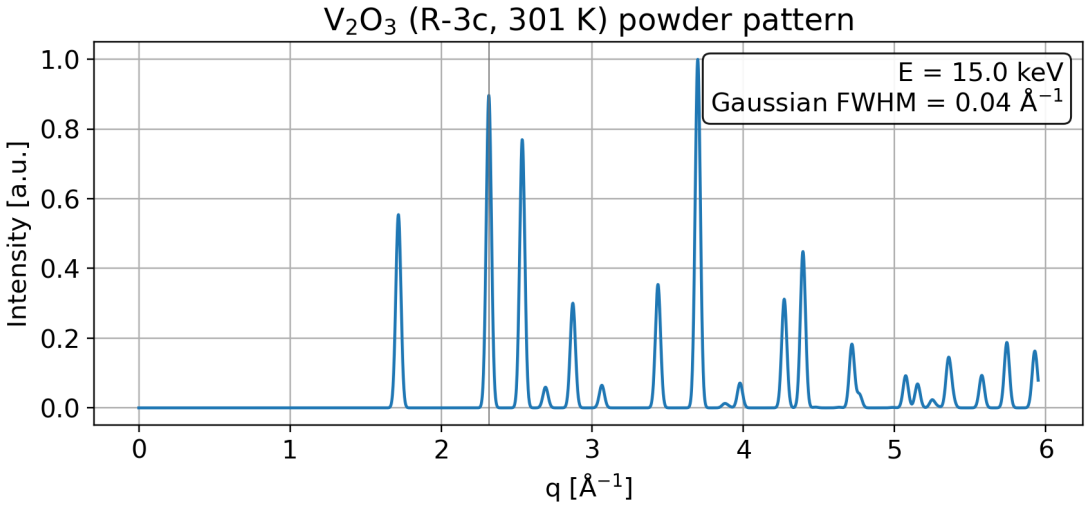


Figure 6.1: Simulated room-temperature R-3c V_2O_3 powder pattern at 15 keV, shown against q with Gaussian FWHM 0.04 $Å^{-1}$.

6.3 Three-dimensional powder cones

```
import numpy as np
from trxrdpy.simulation import polycrystalline

hkls = np.array([
    [0, 1, 2],
    [1, 0, 4],
    [1, 1, 0],
    [1, 1, 6],
    [2, 1, 4],
    [3, 0, 0],
])
q_values = np.array([
    1.7174,
    2.3157,
    2.5368,
    3.6979,
    4.2700,
    4.3938,
])

experiment = polycrystalline.simulate_3d(
    det_type="rayonixmx170",
    det_binning=(4, 4),
    det_dist=0.10,
    det_poni1=0.085,
    det_poni2=0.085,
```

```
    energy=15000.0,
    e_bandwidth=1.0,
    q_hkls=q_values,
    hkls_names=hkls,
    cones_num_of_points=90,
)
```

Exactly one reflection-position representation is conceptually required. If both `q_hkls` and `d_hkls` are supplied, the current implementation gives `d_hkls` precedence and converts it using $q = 2\pi/d$. Labels must contain one Miller triplet per position.

The Rayonix MX170 geometry is supplied by the pyFAI detector registry; the example uses 4×4 binning to keep the calculation responsive while preserving the physical detector size. The PONI values place the beam center near the detector center so the selected rings are visible. The selected V2O3 reflections are (012), (104), (110), (116), (214), and (300), spanning approximately $q = 1.72$ to 4.39 Å^{-1}.

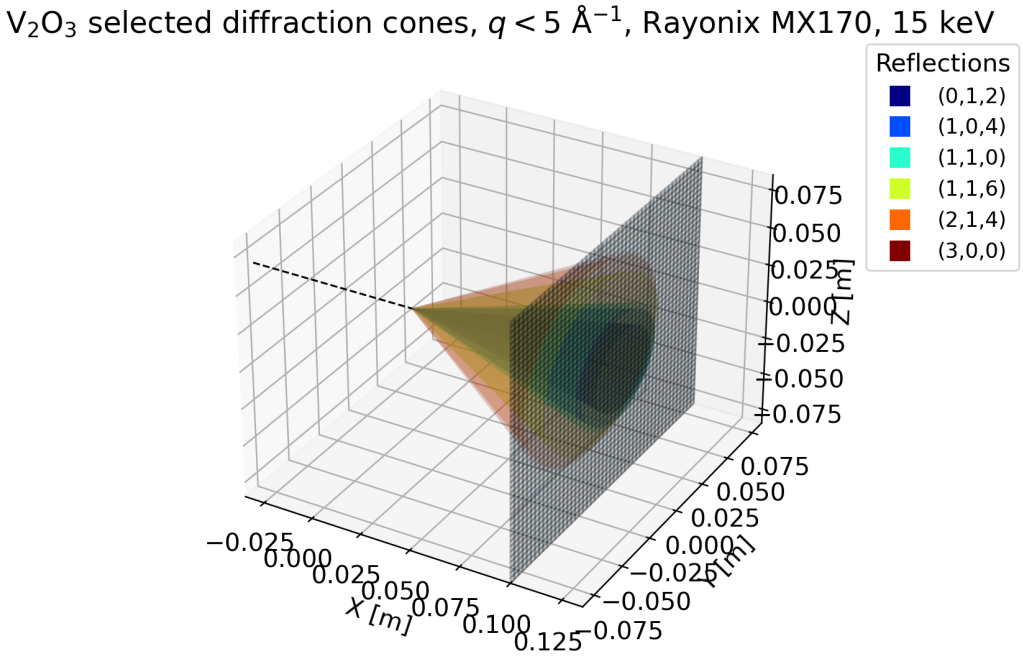


Figure 6.2: Three-dimensional view of the selected R-3c V_2O_3 diffraction cones, all within $q < 5$ Å^{-1}, the incident beam, sample position, and Rayonix MX170 detector plane at 15 keV.

6.4 Two-dimensional detector projection

After the cone geometry has been checked, the same detector and reflection settings can be projected onto the detector plane:

```
experiment = polycrystalline.simulate_2d(
    det_type="rayonixmx170",
    det_binning=(4, 4),
    det_dist=0.10,
    det_poni1=0.085,
    det_poni2=0.085,
    energy=15000.0,
    e_bandwidth=1.0,
```

```
    q_hkls=q_values,
    hkls_names=hkls,
    cones_num_of_points=720,
)
```

Use fewer cone points for a responsive 3D preview and more points for a smooth 2D ring. The returned `Experiment` retains the detector and calculated cone information for further inspection.

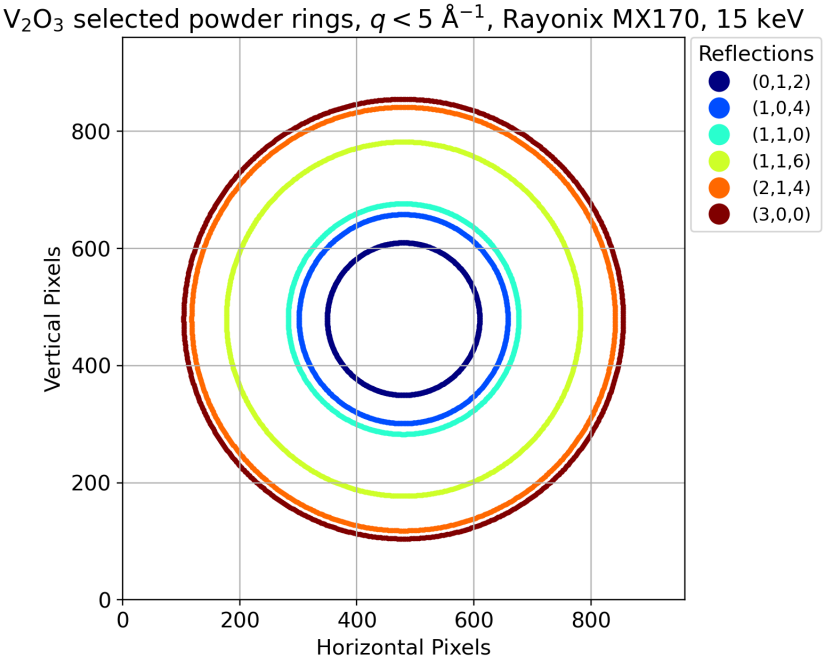


Figure 6.3: Selected R-3c V_2O_3 powder reflections (012), (104), (110), (116), (214), and (300), all within $q < 5$ Å^{-1}, intersecting a 4×4 binned Rayonix MX170 detector at 15 keV and 0.10 m sample distance.

6.5 Manual detector from experimental geometry

The same reflections can be projected onto a detector defined entirely by the experimental geometry. In the following example, `max_shape=[831,1475]` maps to `num_pixels_v=831` and `num_pixels_h=1475`. The rotation values are written as PONI-style `Rot1`, `Rot2`, and `Rot3` values in radians, then converted to degrees because the XRDpy detector API uses degree-valued Euler angles.

```
wavelength_m = 1.3829804621662048e-10
energy_eV = 12398.419843320026 / (wavelength_m * 1e10)

manual_detector_kwargs = dict(
    det_type="manual",
    det_pxsize_h=0.000172,
    det_pxsize_v=0.000172,
    det_ntum_pixels_h=1475,
    det_num_pixels_v=831,
    det_dist=0.12,
    det_poni1=0.050,
    det_poni2=0.123,
    det_rotx=np.degrees(-0.023),
```

```
    det_roty=np.degrees(-0.5238161921742152),
    det_rotz=np.degrees(0.0),
    energy=energy_eV,
    e_bandwidth=1.0,
    q_hkls=q_values,
    hkls_names=hkls,
)

experiment_3d = polycrystalline.simulate_3d(
    **manual_detector_kwargs,
    cones_num_of_points=90,
)

experiment_2d = polycrystalline.simulate_2d(
    **manual_detector_kwargs,
    cones_num_of_points=720,
)
```

This wavelength corresponds to approximately 8.965 keV. The off-center PONI and nonzero detector rotations intentionally produce an asymmetric detector cut through the diffraction cones.

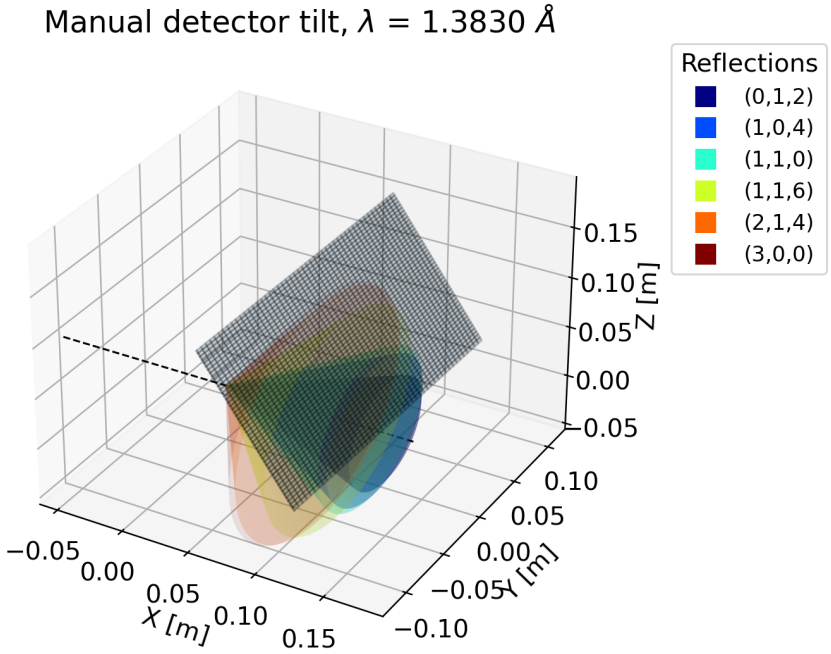


Figure 6.4: Three-dimensional view of the manually defined detector using Rot1 = −0.023, Rot2 = −0.5238161921742152, Rot3 = 0 radians, showing the tilted detector plane relative to the diffraction cones.

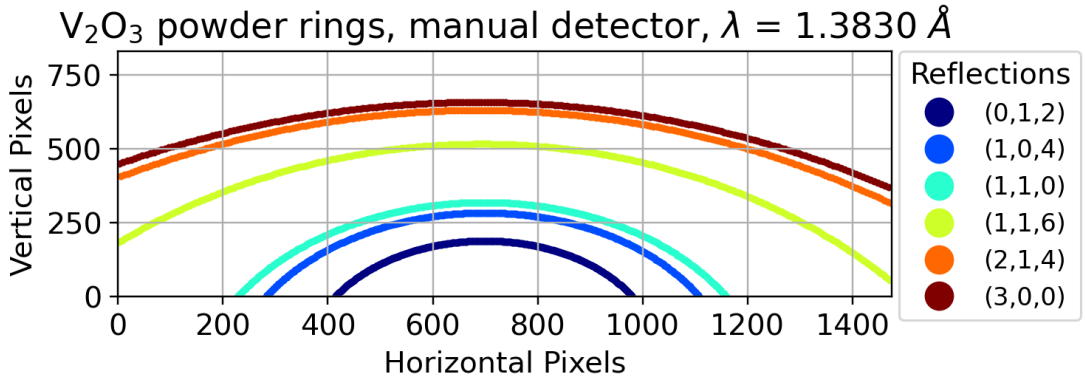


Figure 6.5: Selected R-3c V_2O_3 powder reflections on the manually defined 831×1475 detector with 172 μm pixels, 0.12 m sample distance, off-center PONI, and wavelength 1.38298 Å.

6.6 Using a PONI detector

Replace the manual geometry arguments with:

```
experiment = polycrystalline.simulate_2d(
    det_type="poni",
    det_poni_file="calibration.poni",
    energy=15000.0,
    d_hkls=d_angstrom,
    hkls_names=hkls,
)
```

The incident energy remains an explicit simulation input. A wavelength stored in the PONI file is parsed as metadata but does not silently replace the high-level `energy` argument.

6.7 Interpreting powder output

A ring outside the detector is not an error: it may follow from energy, detector distance, beam-center position, or the selected q range. Partial rings arise naturally when the detector intercepts only part of a cone or when it is tilted. If all rings are unexpectedly absent, check units first, then detector orientation and reflection values.

The 2D and 3D routines label geometry; they do not automatically model experimental backgrounds, detector response, texture, finite crystallite size, or all intensity corrections. Their primary purpose is to answer where reflections should intersect the specified detector.

Chapter 7

Diffractometers and motor-chain geometry

7.1 Why use motor chains

Three Euler angles are sufficient for a mathematical orientation but do not necessarily describe a physical instrument. A motor chain names each axis, fixes its origin and reference frame, defines its application order, and separates sample-stage motion from detector-arm motion.

The geometry model contains:

- `Motor`: a named axis, origin, frame, default angle, and metadata;
- `MotorChain`: an ordered list of motors; and
- `DiffractometerGeometry`: independent sample and detector chains.

7.2 Lab-fixed and local axes

A motor with `frame="lab"` keeps its axis fixed in laboratory coordinates. A motor with `frame="local"` inherits the transformations applied by earlier motors in the chain. This distinction is essential for nested cradles such as a kappa stage.

Motor order is list order. Reordering two non-commuting rotations generally changes the result.

7.3 Predefined Euler geometry

```
from trxrdpy.simulation.diffractometers import make_euler_geometry

euler = make_euler_geometry(
    sample_rotation_order="xyz",
    detector_rotation_order="zyx",
    sample_frame="lab",
    detector_frame="lab",
)

sample_angles = {
    "sam_rotx": 2.0,
    "sam_roty": 0.0,
    "sam_rotz": 15.0,
}
detector_angles = {
    "det_rotx": 0.0,
    "det_roty": 3.0,
    "det_rotz": 0.0,
```

```
}
```

With lab-fixed axes, this reproduces the package's Euler-style rotation approach while exposing explicit motor names.

7.4 Predefined kappa geometry

```
from trxrdpy.simulation.diffractometers import make_kappa_geometry

kappa = make_kappa_geometry(
    kappa_tilt_deg=50.0,
    omega_axis="z",
    phi_axis="z",
    two_theta_axis="y",
)

sample_angles = {"omega": 10.0, "kappa": 20.0, "phi": -5.0}
detector_angles = {"tth": 24.0}
```

The default sample chain is omega about a lab-fixed axis, followed by local kappa and phi axes. The detector chain contains one lab-fixed two-theta motor. This is a generic model, not a guarantee that the axes match a particular facility instrument.

7.5 Custom geometry

```
from trxrdpy.simulation.geometry import DiffractometerGeometry

custom = DiffractometerGeometry.from_dict({
    "name": "my_instrument",
    "sample": [
        {
            "name": "omega",
            "axis": "z",
            "origin": [0, 0, 0],
            "frame": "lab",
            "default_angle": 0,
        },
        {
            "name": "chi",
            "axis": [1, 0, 0],
            "origin": [0, 0, 0],
            "frame": "local",
            "default_angle": 0,
        },
    ],
    "detector": [
        {
            "name": "delta",
            "axis": "y",
            "origin": [0, 0, 0],
            "frame": "lab",
            "default_angle": 0,
        },
    ],
```

```
})
```

Axes may use `x`, `-x`, `y`, `-y`, `z`, `-z`, or an explicit three-vector. Origins are three-vectors in meters. Unknown motor names in an angle map raise `KeyError`, which helps catch typographical errors.

7.6 Inspecting and serializing geometry

```
from trxrdpy.simulation.diffractometers import (
    available_diffractometers,
    diffractometer_to_dict,
)

print(available_diffractometers())
print(custom.sample.motor_names)
print(custom.sample.resolved_motors({"omega": 5, "chi": 10}))

geometry_dict = diffractometer_to_dict(custom)
```

The serializer returns JSON-compatible dictionaries. Persist the resulting dictionary with the application's preferred JSON or configuration mechanism. When debugging geometry, `resolved_motors()` exposes each active axis and origin in laboratory coordinates.

Chapter 8

Single-crystal diffraction

8.1 High-level simulation

The main single-crystal entry points create a lattice, generate reflections to `qmax`, apply an orientation, test the Bragg condition, construct the detector, and plot the result. Both 2D and 3D forms are available.

```
from trxrdpy.simulation import single_crystal

simulation_kwargs = dict(
    det_type="manual",
    det_pxsize_h=75e-6,
    det_pxsize_v=75e-6,
    det_ntum_pixels_h=1024,
    det_num_pixels_v=1024,
    det_dist=0.08,  # compact geometry keeps the hit on detector
    det_poni1=0.0384,
    det_poni2=0.0384,
    energy=15000.0,
    e_bandwidth=1.0,
    sam_space_group=227,
    sam_a=5.431,
    sam_rotx=-30.0,
    sam_roty=-20.0,
    sam_rotz=20.0,
    sam_rotation_order="xyz",
    qmax=6.0,
)

experiment_3d = single_crystal.simulate_3d(**simulation_kwargs)
experiment_2d = single_crystal.simulate_2d(**simulation_kwargs)
```

As in the powder interface, the horizontal pixel-count keyword retains the historical spelling `det_ntum_pixels_h`. The sample arguments begin with `sam_`; detector arguments begin with `det_`.

The 3D call visualizes the experiment in laboratory coordinates before the detector projection is inspected. Three-dimensional views are useful for diagnosing orientation and sign conventions before concentrating on detector pixels.

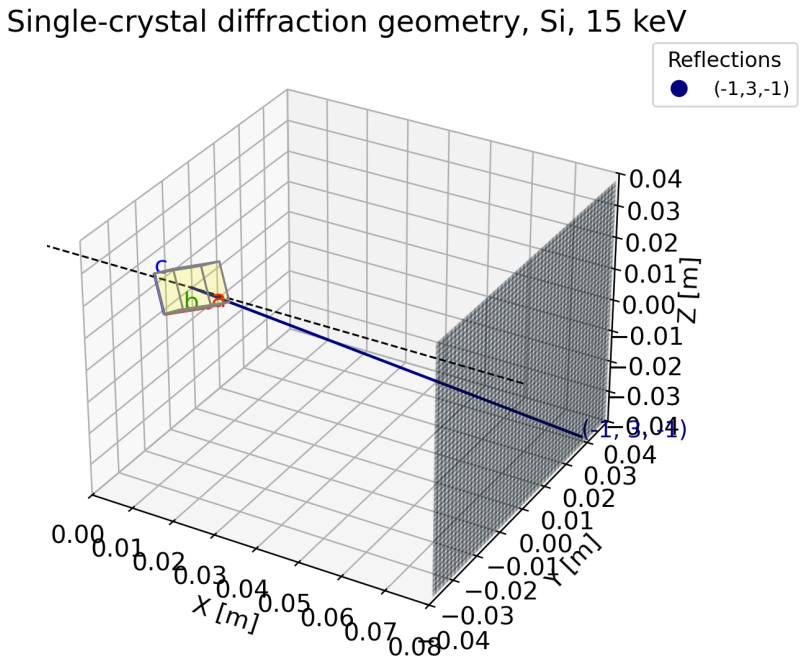


Figure 8.1: Three-dimensional single-crystal silicon experiment showing the incident beam, oriented reciprocal-lattice solution, diffracted ray, and detector plane at 15 keV.

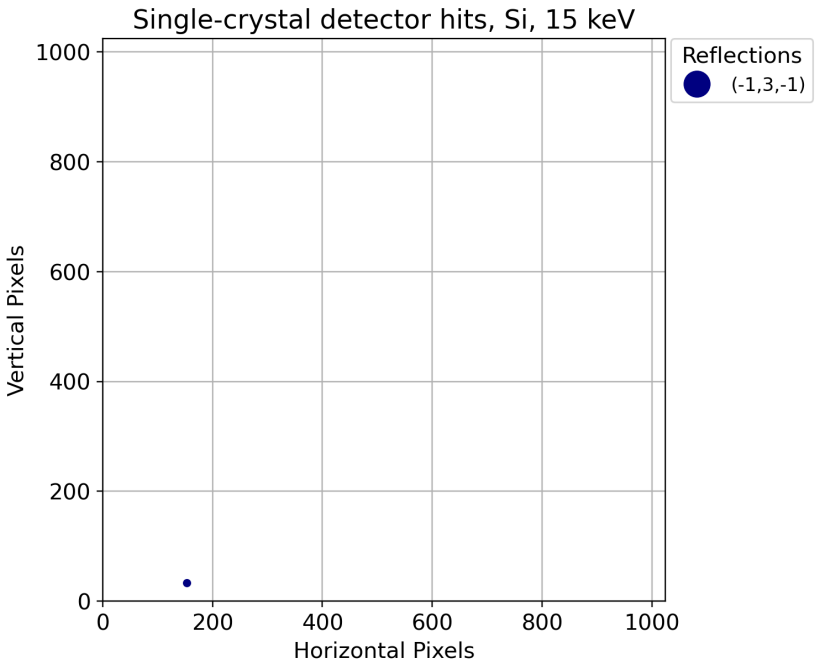


Figure 8.2: Single-crystal silicon reflection projected onto the manual area detector at 15 keV, using the high-level 2D workflow.

8.2 Geometry-aware simulation

```
from trxrdpy.simulation.diffractometers import make_kappa_geometry

instrument = make_kappa_geometry(kappa_tilt_deg=50.0)
```

```
experiment = single_crystal.simulate_2d_with_geometry(
    det_type="manual",
    det_pxsize_h=75e-6,
    det_pxsize_v=75e-6,
    det_ntum_pixels_h=1024,
    det_num_pixels_v=1024,
    det_dist=0.08,
    det_poni1=0.0384,
    det_poni2=0.0384,
    energy=15000.0,
    e_bandwidth=1.0,
    sam_space_group=227,
    sam_a=5.431,
    qmax=6.0,
    geometry=instrument,
    sample_angles={"omega": -30.0, "kappa": 40.0, "phi": 30.0},
    detector_angles={"tth": 0.0},
)
```

Passing `geometry` supplies it to both sample and detector unless `sample_geometry` or `detector_geometry` is set separately. More advanced calls can pass independent motor chains or direct transforms. Avoid combining several geometry routes in one call; the high-level builder applies a defined precedence, but a single explicit route is easier to audit.

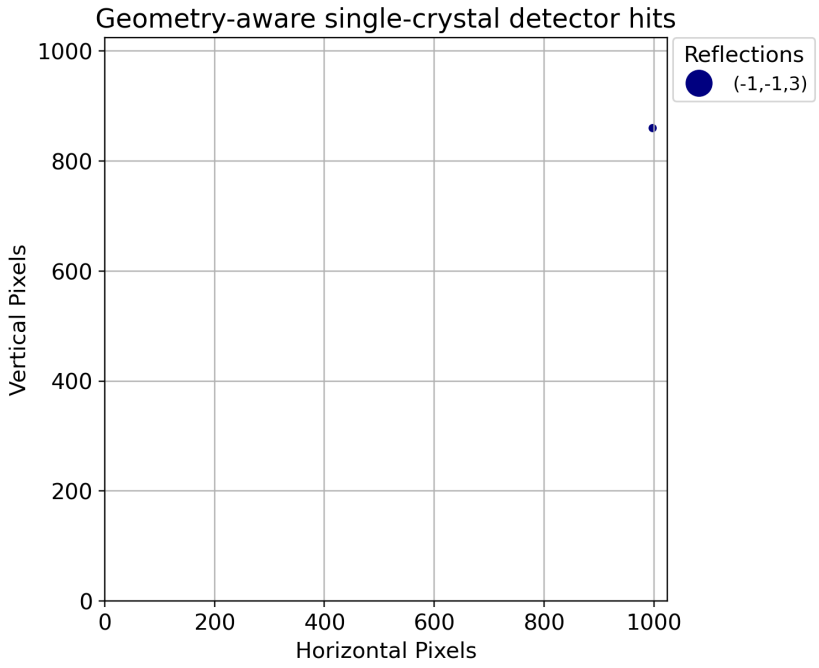


Figure 8.3: Geometry-aware single-crystal detector simulation using named sample and detector motors.

8.3 Explicit additional reflections

The `extra_hkls` argument adds requested Miller indices to the space-group-generated list while preserving first occurrence and removing exact duplicates:

```
experiment = single_crystal.simulate_3d(
```

```
    sam_space_group=227,
    sam_a=5.431,
    qmax=6.0,
    energy=15000.0,
    e_bandwidth=1.0,
    sam_rotx=-30.0,
    sam_roty=-20.0,
    sam_rotz=20.0,
    extra_hkls=[[4, 0, 0], [5, 1, 1]],
)
```

Adding an index does not guarantee that it satisfies symmetry, energy, or detector-intersection requirements. Treat this option as an explicit request for evaluation, not as proof of physical observability.

8.4 Searching sample orientations

There are three related routes for sample-orientation searches. The lattice method stores results on an existing object; `sample_rotations_for_Bragg_condition` is a complete legacy convenience workflow; and `scan_two_motors_for_Bragg_condition` scans named motors from a physical chain.

The tracking examples below use the same room-temperature R-3c V_2O_3 reflection set used in the powder chapter:

```
import numpy as np

v2o3_sample = {
    "sam_space_group": 167,
    "sam_a": 4.9537,
    "sam_c": 14.0111,
}

tracked_hkls = np.array([
    [0, 1, 2],  # (012)
    [1, 0, 4],  # (104)
    [1, 1, 0],  # (110)
    [1, 1, 6],  # (116)
    [2, 1, 4],  # (214)
    [3, 0, 0],  # (300)
])
```

8.4.1 Legacy two-angle convenience workflow

The legacy high-level function scans sample `roty` and `rotz` over the same inclusive range, opens a rotation map, and returns a `LatticeStructure` whose `rotations_for_Bragg_condition` attribute contains the accepted orientations:

```
legacy_scan = single_crystal.sample_rotations_for_Bragg_condition(
    **v2o3_sample,
    sam_rotx=-30.0,
    angle_range=(-180, 180, 0.5),
    energy=15000.0,
    e_bandwidth=2.0,
    hkls_names=tracked_hkls,
)
```

```
accepted = legacy_scan.rotations_for_Bragg_condition
```

At lattice level, the equivalent operation is `lattice.find_Bragg_orientations(...)`. That method updates the lattice in place rather than returning the accepted mapping directly.

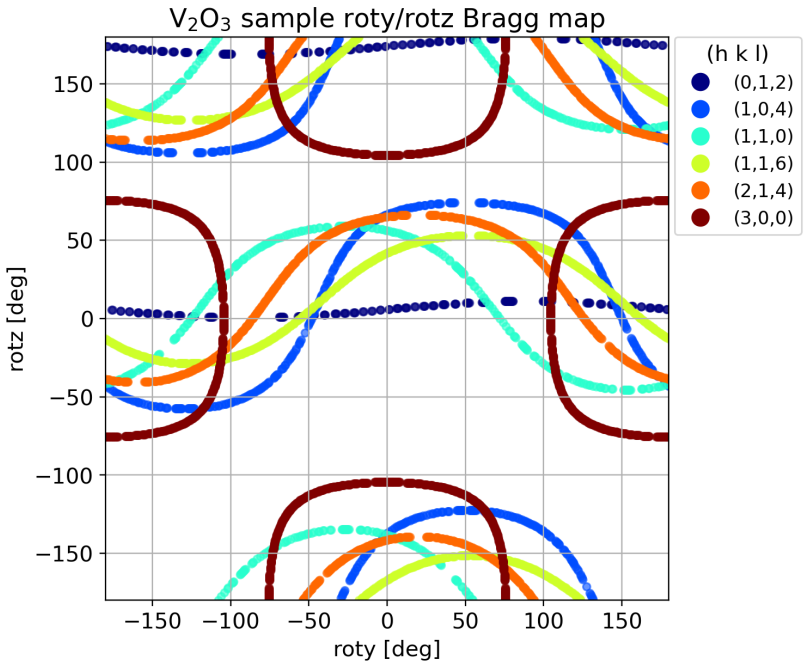


Figure 8.4: Legacy sample `roty`/`rotz` combinations satisfying the Bragg condition for the requested V_2O_3 reflections.

8.4.2 Named-motor two-dimensional scan

Use `scan_two_motors_for_Bragg_condition` when the two variables are real sample motors. The function accepts either a `MotorChain` or a `DiffractometerGeometry`; a supplied geometry takes precedence and its sample chain is scanned.

```
from trxrdpy.simulation.single_crystal import (
    scan_two_motors_for_Bragg_condition,
)

motor_scan = scan_two_motors_for_Bragg_condition(
    motor1_name="omega",
    motor2_name="phi",
    motor1_range=(-180, 180, 0.5),
    motor2_range=(-180, 180, 0.5),
    **v2o3_sample,
    energy=15000.0,
    e_bandwidth=2.0,
    hkls_names=tracked_hkls,
    hkl_equivalent=False,
    geometry=instrument,
    fixed_angles={"kappa": 40.0},
)
```

The returned dictionary maps a stringified Miller index to accepted `(motor1, motor2)` pairs. Set `hkl_equivalent=True` to expand every requested reflection through lattice symmetry before

scanning. Search cost grows as the product of both grid sizes, reflection count, and any equivalence expansion.

```
from trxrdpy.simulation.plot import plot_parameter_mapping

plot_parameter_mapping(motor_scan, "omega", "phi")
```

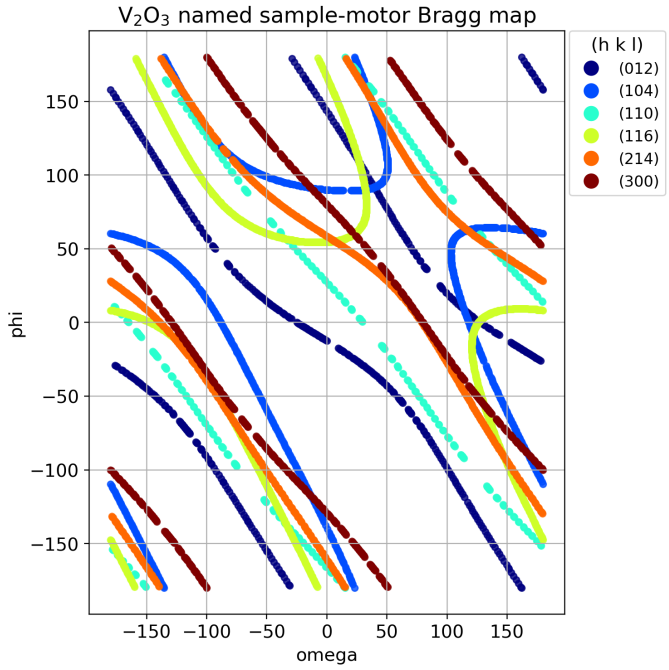


Figure 8.5: Named sample-motor combinations satisfying the Bragg condition for the selected V_2O_3 reflections.

8.5 Scanning two parameters

`scan_two_parameters_for_Bragg_condition` scans any two distinct choices from `rotx`, `roty`, `rotz`, and `energy`. It is therefore useful for both angle–angle maps and energy–angle maps. The sample rotations supplied through `sam_rotx`, `sam_roty`, and `sam_rotz` form the baseline; scanned rotations are added to it.

```
energy_angle_scan = (
    single_crystal.scan_two_parameters_for_Bragg_condition(
        param1_name="roty",
        param2_name="energy",
        param1_range=(-180, 180, 0.5),    # degree increment
        param2_range=(13000, 17000, 100),  # eV
        **v2o3_sample,
        sam_rotx=-30.0,
        sam_roty=-20.0,
        sam_rotz=20.0,
        e_bandwidth=2.0,
        hkls_names=tracked_hkls,
        hkl_equivalent=False,
    )
)

plot_parameter_mapping(energy_angle_scan, "roty", "energy")
```

The output has the same general mapping form as the named-motor scan. Rotation ranges are in degrees; an energy range is in eV. Use physically meaningful bounds and an initial coarse grid, then refine locally.

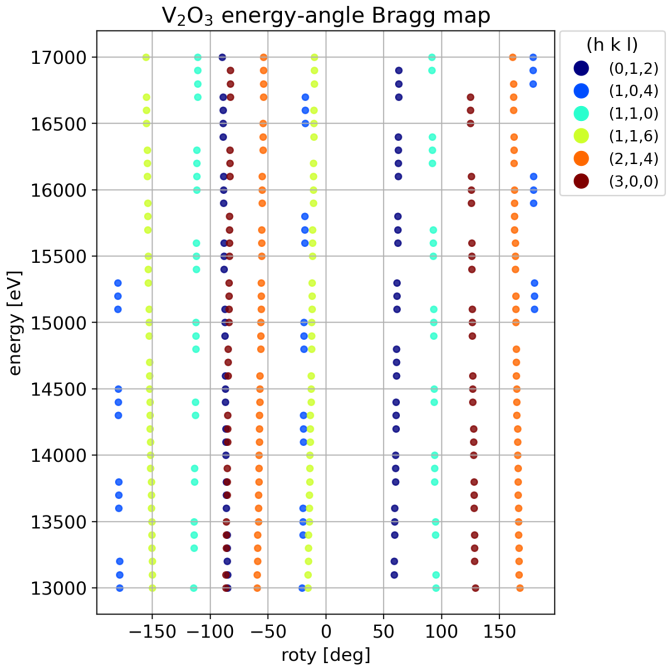


Figure 8.6: Example energy-versus-sample-angle map for the selected V_2O_3 reflections without symmetry-equivalent expansion.

8.6 Collecting Bragg reflections with detector rotations

Detector-rotation search functions answer a different question from sample-orientation searches: given a crystal state and selected reflections, which detector-arm settings intercept the diffracted rays? The package provides legacy Euler, motor-chain, and geometry-aware variants.

8.6.1 Legacy detector Euler scan

`detector_rotations_collecting_Braggs` scans detector `roty` and `rotz` over one common inclusive range. It returns an `Experiment`, populates successful detector rotations, and opens the legacy rotation map.

```
legacy_detector_scan = (
    single_crystal.detector_rotations_collecting_Braggs(
        det_type="manual",
        det_pxsize_h=75e-6,
        det_pxsize_v=75e-6,
        det_ntum_pixels_h=1024,
        det_num_pixels_v=1024,
        det_dist=0.08,
        det_poni1=0.0384,
        det_poni2=0.0384,
        angle_range=(-45, 45, 5),
        energy=15000.0,
        e_bandwidth=2.0,
```

```
        **v2o3_sample,
        sam_rotx=-30.0,
        sam_roty=83.0,
        sam_rotz=50.0,
        hkls=tracked_hkls,
    )
)
```

8.6.2 Detector motor-chain and geometry scans

`detector_rotations_collecting_Braggs_with_chain` accepts arbitrary detector motor names and independent scan ranges. `detector_rotations_collecting_Braggs_with_geometry` is the corresponding convenience wrapper for the detector arm of a `DiffractometerGeometry`.

```
geometry_detector_scan = (
    single_crystal.detector_rotations_collecting_Braggs_with_geometry(
        det_type="manual",
        det_pxsize_h=75e-6,
        det_pxsize_v=75e-6,
        det_ntum_pixels_h=1024,
        det_num_pixels_v=1024,
        det_dist=0.08,
        det_poni1=0.0384,
        det_poni2=0.0384,
        energy=15000.0,
        e_bandwidth=2.0,
        **v2o3_sample,
        sam_rotx=-30.0,
        sam_roty=83.0,
        sam_rotz=50.0,
        hkls=tracked_hkls,
        geometry=instrument,
        scan_ranges={"tth": (-45, 45, 1)},
    )
)

successful = geometry_detector_scan.success_detector_rotations
```

For the chain form, replace `geometry=instrument` with `detector_motor_chain=...`; use `fixed_detector_angles` for unscanned detector motors. In both modern forms, the returned experiment stores successful configurations by Miller index. Keep the sample orientation fixed and document the detector rotation order or motor chain used.

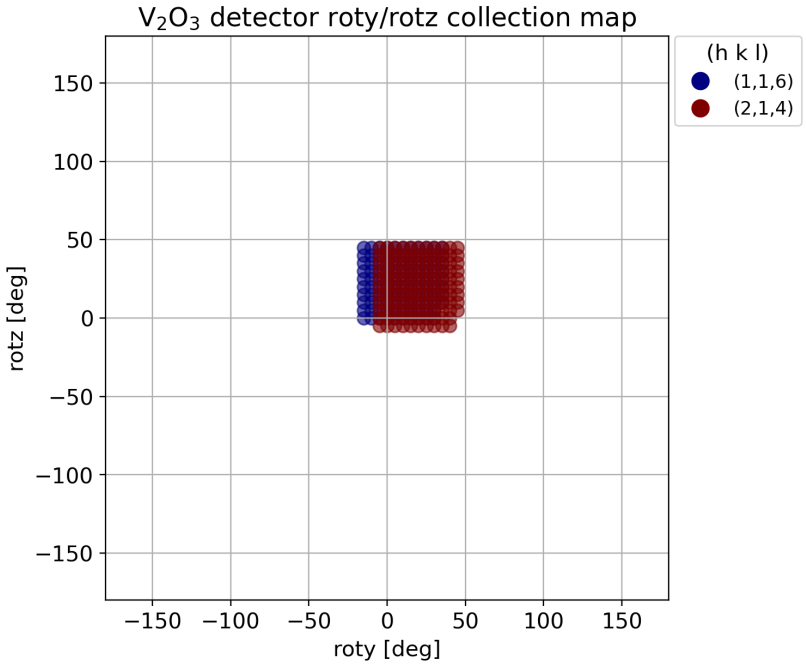


Figure 8.7: Detector-arm settings that intercept the V_2O_3 (116) and (214) Bragg reflections for the fixed sample orientation selected from the requested tracking set.

8.7 Fixed-energy inverse targeting

`target_hkl_near_pixel_fixed_energy` searches for orientations that place one selected reflection near a target detector pixel at fixed energy. It samples the elastic q cone, intersects rays with the detector, retains hits inside a pixel tolerance, and expands the free rotation around each accepted q target.

```
targeting = single_crystal.target_hkl_near_pixel_fixed_energy(
    det_type="manual",
    det_pxsize_h=75e-6,
    det_pxsize_v=75e-6,
    det_ntum_pixels_h=1024,
    det_num_pixels_v=1024,
    det_dist=0.08,
    det_poni1=0.0384,
    det_poni2=0.0384,
    energy=15000.0,
    **v2o3_sample,
    target_hkl=[1, 0, 4],
    target_pixel=(737, 737),
    pixel_tolerance_px=10.0,
    eta_samples=721,
    phi_samples=181,
    do_detector_plot=True,
    do_2d_plot=True,
    do_3d_plot=True,
)
```

The tested geometry above produces accepted Euler solutions. With an active sample motor chain, the solution object additionally carries ordered `motor_names` and an `(N,M)` array in `motor_angles_deg`.

```
print(targeting.all_pixels[targeting.best_idx])
print(targeting.pixel_error_px[targeting.best_idx])
print(targeting.accepted_mask.sum())

solutions = targeting.solutions
if solutions is not None:
    print(solutions.solution_mode)
    print(solutions.predicted_pixels)
    print(solutions.pixel_error_px)
```

The returned `FixedEnergyTargetingResult` also retains the complete sampled cone, all detector intersections, on-detector and accepted masks, inferred beam center, detector object, and best-hit index. Its `solutions` field is either `None` or a `FixedEnergyAcceptedSolutions` object. Set plotting flags to false when only numerical post-processing is needed.

This is an inverse problem. Multiple orientations can satisfy closely related constraints, and limits or seed choices may affect which families are found. Validate each accepted solution by forward simulation and examine motor feasibility before using it experimentally.

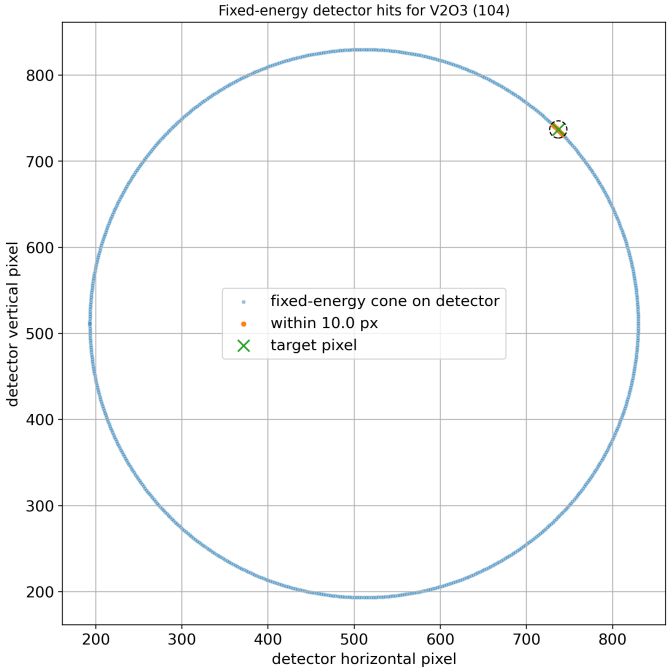


Figure 8.8: Reachable fixed-energy detector hits for V_2O_3 (104), requested target pixel, acceptance radius, and best sampled hit.

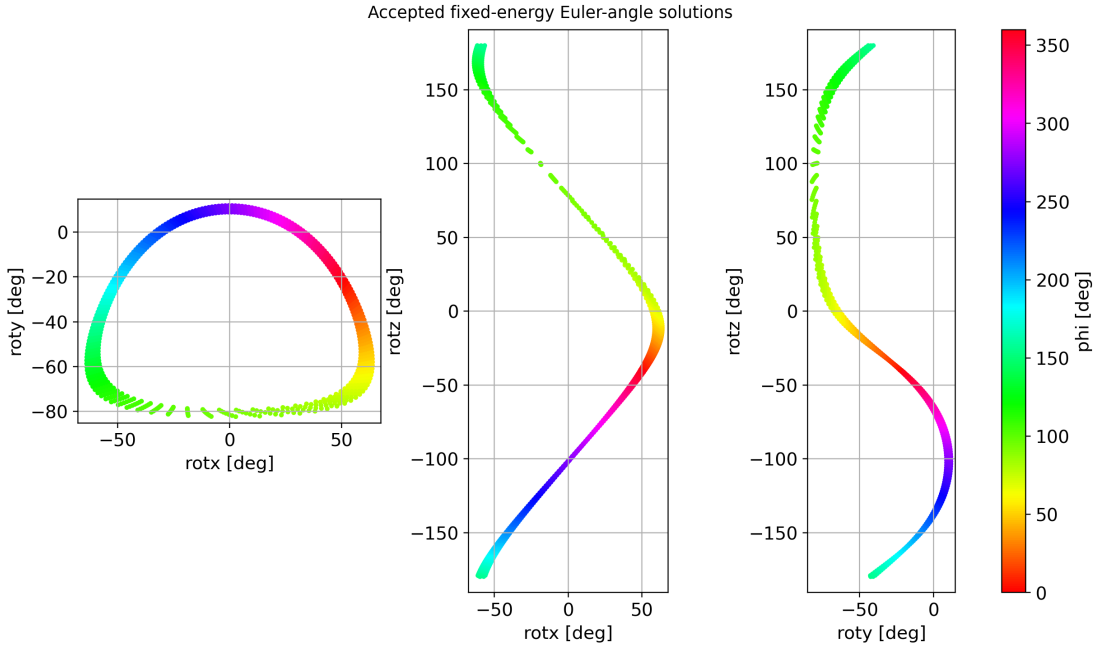


Figure 8.9: Pairwise projections of the accepted Euler-angle or named-motor solution family.

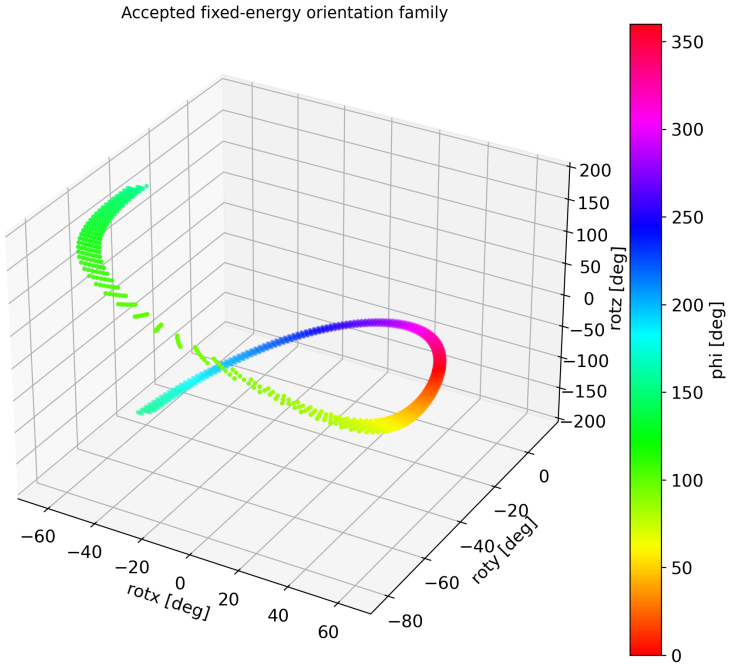


Figure 8.10: Optional three-dimensional view of the accepted orientation family, colored by the free rotation coordinate.

Instrument safety

Geometry solutions are computational results, not collision-checked motion commands. Never transfer angles directly to hardware without instrument limits, collision envelopes, sign conventions, and beamline procedures being independently verified.

Chapter 9

Plotting and result inspection

9.1 Plot families

The Simulation plotting module provides:

- three-dimensional detector and crystal views;
- two-dimensional powder and single-crystal detector plots;
- diffraction-cone plots;
- one-dimensional powder patterns;
- reciprocal-space views;
- rotation and parameter mappings; and
- fixed-energy detector-hit and motor-family visualizations.

High-level simulation calls normally create the relevant plots automatically. Lower-level functions accept existing Matplotlib axes in several cases, which makes it possible to compose publication or diagnostic figures.

9.2 Separating calculation from presentation

For reusable workflows, retain the returned `Experiment`, lattice, detector, or result object and treat plotting as a separate step. This makes it easier to:

- inspect numerical arrays before display;
- compare several configurations on consistent axes;
- save derived values alongside figures; and
- rerender plots without repeating expensive scans.

9.3 Plot interpretation

Detector plots should be read together with their coordinate labels, image origin, and beam-center convention. Reciprocal-space plots help diagnose whether a missing detector reflection fails the Bragg condition or merely misses the active detector area. Rotation maps reveal families of candidate orientations but do not replace forward validation.

For dense reflection sets, labels and hover annotations can become crowded. Reduce `qmax`, select explicit reflections, or use a coarse view first. The optional `mplcursors` dependency supports interactive hover information in applicable GUI plots.

9.4 Saving figures

Matplotlib figures can be saved using the active figure object:

```
import matplotlib.pyplot as plt

experiment = polycrystalline.simulate_2d(...)
plt.gcf().savefig(
    "figures/powder_2d_rings.png",
    dpi=300,
    bbox_inches="tight",
)
```

The filename matches the generated figure reference in Chapter 6; recompilation inserts it automatically. For workflows that open several figures, inspect `plt.get_fignums()` and save each figure with the filename used by its `\manualfigure` call. For reproducibility, save the input parameters next to the figure. A plot without energy, lattice, detector calibration, orientation, and package version is difficult to reconstruct.

Chapter 10

Simulation graphical interface

10.1 Purpose and architecture

The Simulation GUI exposes the backend workflows without replacing them. It contains polycrystalline and single-crystal tabs, a matrix-rotation helper, state persistence, summary and log areas, and plot-window management. The computational work remains in the Simulation modules, so a GUI configuration can be translated into a Python workflow when automation is needed.

10.2 Launching the interface

The GUI is launched with the console entry point:

```
trxrdpy-sim-gui
```

The equivalent module invocation is:

```
python -m trxrdpy.simulation.gui.main_window
```

The screenshots in this chapter show the main GUI states and the figures that the same parameter choices produce. The code examples show the corresponding backend calls for the selected tab; use them as the reproducible form of a GUI configuration.

The GUI is most useful when it is treated as a guided parameter editor. A typical session is to select the calculation family, complete only the active controls, run a three-dimensional view to confirm the geometry, and then inspect the detector image or one-dimensional pattern. This order is deliberate: the 3D view shows whether the beam, sample, detector, and reciprocal-space construction are physically reasonable before the detector projection is interpreted as a final result.

10.3 Polycrystalline tab

The powder tab selects `simulate_1d`, `simulate_2d`, or `simulate_3d`. Its controls cover:

- CIF path, space group, cell parameters, and `qmax`;
- beam energy and relative bandwidth;
- manual, PONI, or pyFAI-registered detector input;
- binning, distance, PONI coordinates, Euler angles, and rotation order;
- explicit q values, d values, and Miller-index labels for 2D/3D use; and
- 1D profile options such as axis, sampling, width, corrections, convolution, and normalization.

For `simulate_1d`, a CIF is required. For 2D and 3D, reflections can be supplied manually or derived through the available crystallographic controls. The interface hides or disables controls that do not apply to the selected function.

```
import numpy as np
from trxrdpy.simulation import polycrystalline

q_hkls = np.array([1.7174, 2.3157, 2.5368])
hkls_names = np.array([[0, 1, 2], [1, 0, 4], [1, 1, 0]])

experiment_3d = polycrystalline.simulate_3d(
    det_type="rayonixmx170",
    det_binning=(4, 4),
    det_dist=0.10,
    det_poni1=0.085,
    det_poni2=0.085,
    energy=15000.0,
    e_bandwidth=1.0,
    q_hkls=q_hkls,
    hkls_names=hkls_names,
    cones_num_of_points=120,
)

experiment_2d = polycrystalline.simulate_2d(
    det_type="rayonixmx170",
    det_binning=(4, 4),
    det_dist=0.10,
    det_poni1=0.085,
    det_poni2=0.085,
    energy=15000.0,
    e_bandwidth=1.0,
    q_hkls=q_hkls,
    hkls_names=hkls_names,
    cones_num_of_points=720,
)
```

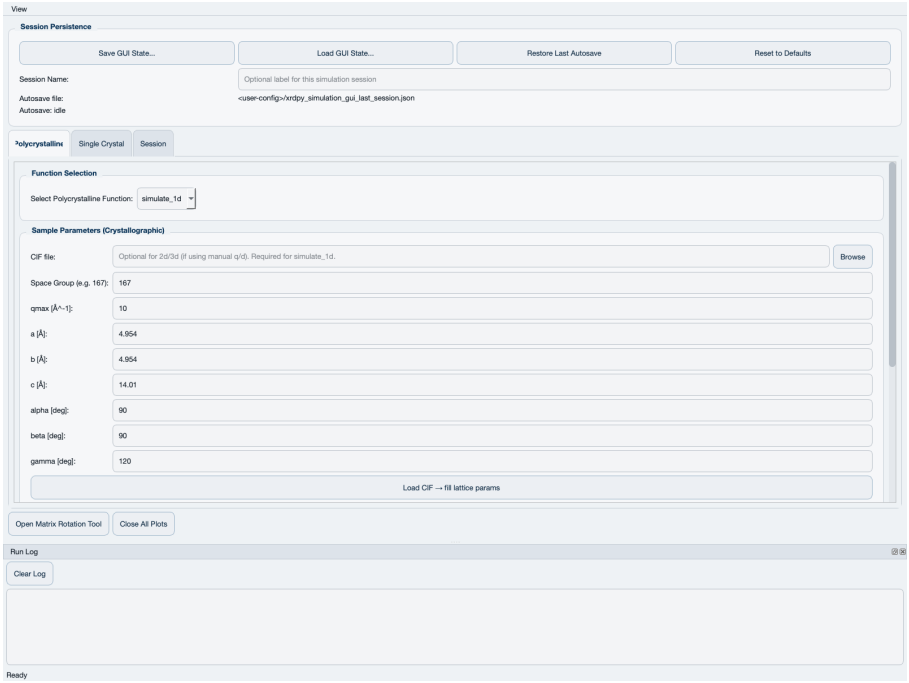


Figure 10.1: Polycrystalline Simulation GUI with the function, sample, beam, detector, and reflection controls visible.

The corresponding simulation outputs are shown with complete executable examples in Chapter 11. Keeping this chapter focused on GUI state avoids repeating the same detector, 3D, and 1D result figures in two places.

10.4 Single-crystal tab

The single-crystal tab exposes:

- 2D and 3D simulation;
- fixed-energy reflection targeting near a detector pixel;
- detector-rotation searches for Bragg reflections;
- two-parameter Bragg-condition scans;
- detector, beam, lattice, orientation, and reflection controls;
- legacy Euler, predefined, and custom geometry modes; and
- generated sample and detector motor-angle controls.

The geometry panel distinguishes legacy Euler mode from geometry-aware mode. In geometry-aware mode, motor names, meanings, axes, frames, origins, defaults, and editable angles are presented explicitly.

```
from trxrdpy.simulation import single_crystal
from trxrdpy.simulation.diffractometers import make_kappa_geometry

geometry = make_kappa_geometry(kappa_tilt_deg=50.0)

experiment_3d = single_crystal.simulate_3d_with_geometry(
    det_type="manual",
    det_pxsize_h=75e-6,
    det_pxsize_v=75e-6,
    det_ntum_pixels_h=1024,
    det_num_pixels_v=1024,
```

```
    det_dist=0.08,
    det_poni1=0.0384,
    det_poni2=0.0384,
    energy=15000.0,
    e_bandwidth=1.0,
    sam_space_group=227,
    sam_a=5.431,
    qmax=6.0,
    geometry=geometry,
    sample_angles={"omega": -30.0, "kappa": 40.0, "phi": 30.0},
    detector_angles={"tth": 0.0},
)

experiment_2d = single_crystal.simulate_2d_with_geometry(
    det_type="manual",
    det_pxsize_h=75e-6,
    det_pxsize_v=75e-6,
    det_ntum_pixels_h=1024,
    det_num_pixels_v=1024,
    det_dist=0.08,
    det_poni1=0.0384,
    det_poni2=0.0384,
    energy=15000.0,
    e_bandwidth=1.0,
    sam_space_group=227,
    sam_a=5.431,
    qmax=6.0,
    geometry=geometry,
    sample_angles={"omega": -30.0, "kappa": 40.0, "phi": 30.0},
    detector_angles={"tth": 0.0},
)
```

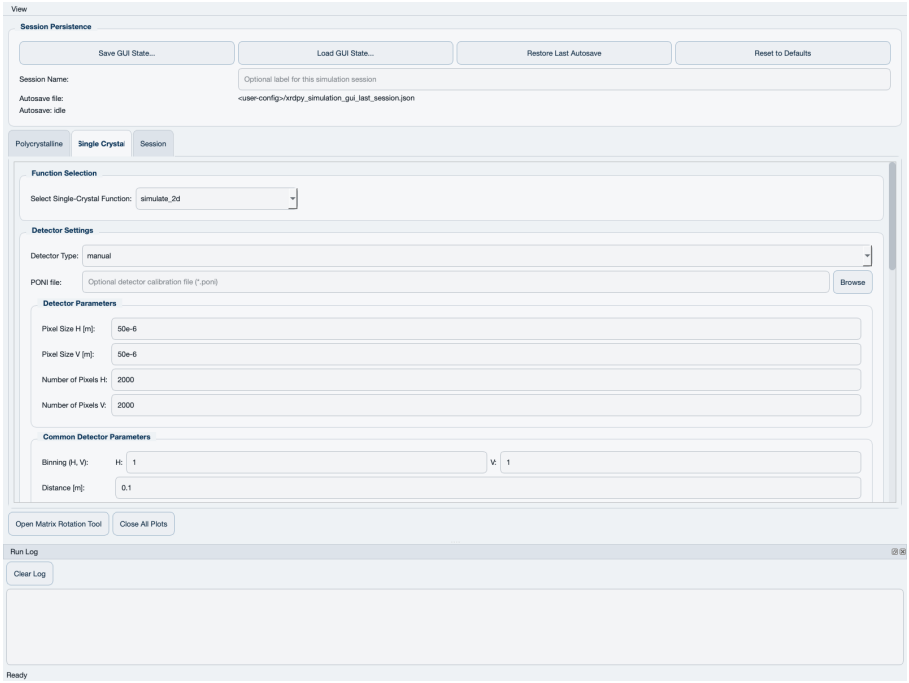


Figure 10.2: Single-crystal Simulation GUI showing geometry selection and generated sample/detector motor controls.

The complete single-crystal examples in Chapter 11 show the generated 3D and detector-projection

figures. In GUI use, the same pedagogical order applies: inspect the 3D geometry first, then inspect detector pixels.

10.5 Matrix-rotation helper

The matrix-rotation window assists with constructing or transforming orientation matrices. When transferring a matrix to the single-crystal workflow, confirm whether it represents direct-lattice row vectors, a pure rotation, or a complete orientation. Matrix shape alone does not establish its physical meaning.

```
import numpy as np
from trxrdpy.simulation import utils

orientation = np.eye(3)
rotation = utils.euler_to_matrix(10.0, 20.0, 0.0, rotation_order="xyz")
rotated_orientation = utils.apply_rotation_matrix(orientation, rotation)
```

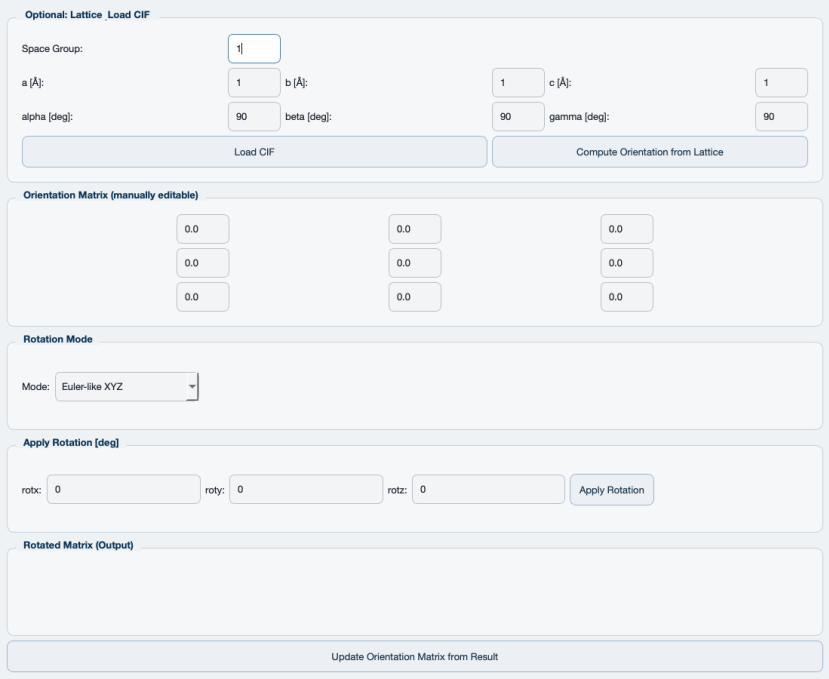


Figure 10.3: Matrix-rotation helper used to construct or transform an orientation matrix before importing it into a single-crystal configuration.

10.6 Sessions and logs

The GUI maintains session state and supports autosave and restore. Logs and summaries should be retained with significant simulations because they provide a human-readable record of selected functions and parameters. Session recovery is a convenience, not a replacement for a deliberate, version-controlled simulation record.

10.7 GUI operating sequence

1. Select the powder or single-crystal tab and the desired function.
2. Load a CIF or enter the required lattice/reflection values.
3. Enter beam energy and bandwidth.
4. Choose and validate detector input.
5. Choose legacy or geometry-aware rotations.
6. Enter scan or targeting parameters when applicable.
7. For 2D/3D families, run or inspect the 3D view first so the beam, sample, reciprocal-space object, and detector plane can be checked together.
8. Inspect the detector projection or one-dimensional pattern.
9. Check the summary and log area for the exact function family and parameter values.
10. Save the result figures and the parameter record before changing the configuration.

If a GUI result appears inconsistent, reproduce a minimal version through the Python API. Explicit keyword arguments make unit and convention errors easier to identify.

Chapter 11

Complete simulation examples

11.1 Example data files

Files needed to run the examples are kept with the manual rather than referred to by generic local paths. The current data set is:

`cif_files/V2O3_R-3c_room_T.cif` Corundum-type V_2O_3 at 301 K, space group R-3c (No. 167), with lattice parameters $a = b = 4.9537$ Å, $c = 14.0111$ Å, and $\alpha = \beta = 90°$, $\gamma = 120°$.

The package selects the first CIF data block containing a complete non-zero unit cell. Single-block example files are nevertheless preferable because the intended phase remains explicit to both readers and software.

11.2 Example A: lattice and detector inspection

This example constructs the two central objects without running a high-level simulation:

```
from trxrdpy.simulation.sample import LatticeStructure
from trxrdpy.simulation.detector import Detector
from trxrdpy.simulation.utils import energy_to_wavelength

lattice = LatticeStructure(space_group=227, a=5.431)
lattice.create_possible_reflections(qmax=5.0)
lattice.calculate_q_hkls()
lattice.calculate_d_hkls()
lattice.apply_rotation(rotx=-30, roty=-20, rotz=20)
lattice.check_Bragg_condition(
    wavelength=energy_to_wavelength(15000.0),
    e_bandwidth=1.0,
)

det = Detector(
    detector_type="manual",
    pxsize_h=75e-6,
    pxsize_v=75e-6,
    num_pixels_h=512,
    num_pixels_v=512,
    dist=0.30,
    poni1=0.0192,
    poni2=0.0192,
)
det.calculate_lab_grid()
```

```
print("Allowed reflections:", len(lattice.allowed_hkls))
print("Bragg reflections:", lattice.hkls_in_Bragg_condition)
print("Detector grid:", det.lab_grid.shape)
```

This is a useful first diagnostic because it separates crystallographic reflection generation from detector intersection.

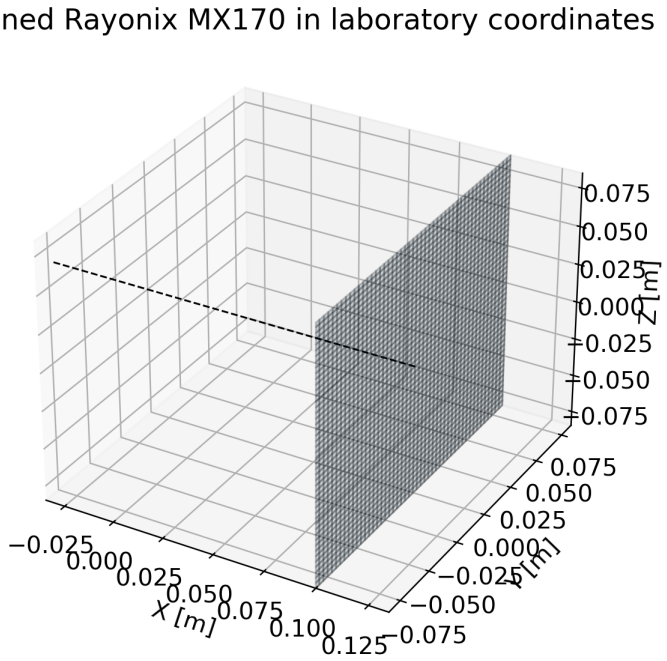


Figure 11.1: Detector laboratory-coordinate grid used as a first geometry diagnostic before running full diffraction simulations.

11.3 Example B: powder detector prediction

```
import numpy as np
from trxrdpy.simulation.polycrystalline import simulate_2d, simulate_3d

hkls = np.array([
    [0, 1, 2],
    [1, 0, 4],
    [1, 1, 0],
    [1, 1, 6],
    [2, 1, 4],
    [3, 0, 0],
])
q = np.array([1.7174, 2.3157, 2.5368,
              3.6979, 4.2700, 4.3938])

exp_3d = simulate_3d(
    det_type="rayonixmx170",
    det_binning=(4, 4),
    det_dist=0.10,
    det_poni1=0.085,
    det_poni2=0.085,
    energy=15000.0,
    q_hkls=q,
```

```
    hkls_names=hkls,
    cones_num_of_points=120,
)

exp_2d = simulate_2d(
    det_type="rayonixmx170",
    det_binning=(4, 4),
    det_dist=0.10,
    det_poni1=0.085,
    det_poni2=0.085,
    energy=15000.0,
    q_hkls=q,
    hkls_names=hkls,
    cones_num_of_points=720,
)

print(exp_2d.detector.num_pixels_h, exp_2d.detector.num_pixels_v)
```

The pyFAI registry supplies the Rayonix MX170 pixel size and native shape. The binning preserves the physical detector extent while reducing the calculation to 960×960 effective pixels.

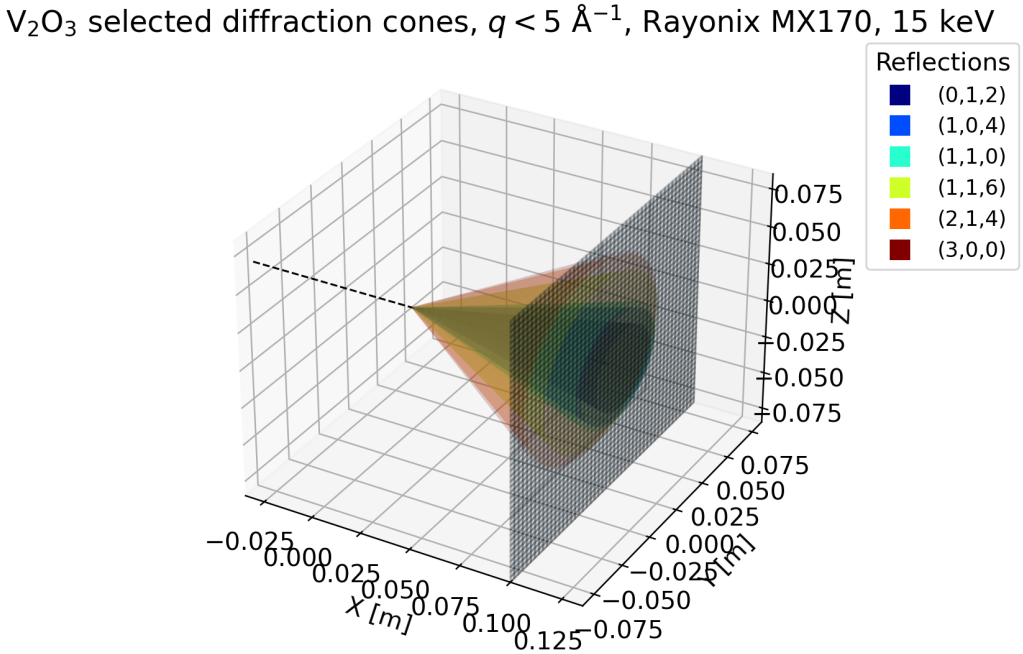


Figure 11.2: Complete simulation example B: three-dimensional powder-cone geometry for the selected R-3c V_2O_3 reflections.

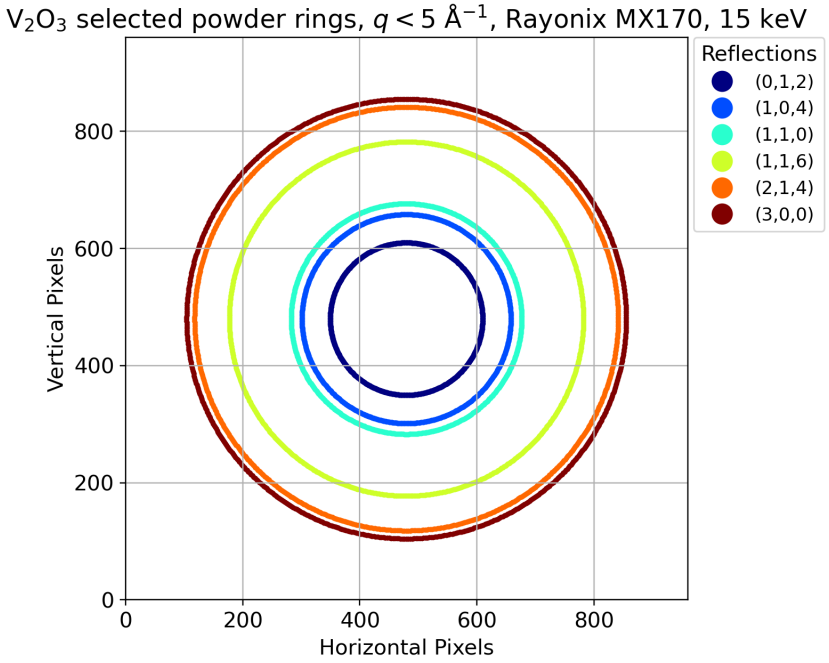


Figure 11.3: Complete simulation example B: detector projection generated after the 3D powder geometry has been checked.

11.4 Example C: manual detector from calibration values

```
import numpy as np
from trxrdpy.simulation.polycrystalline import simulate_2d, simulate_3d

hkls = np.array([
    [0, 1, 2],
    [1, 0, 4],
    [1, 1, 0],
    [1, 1, 6],
    [2, 1, 4],
    [3, 0, 0],
])
q = np.array([1.7174, 2.3157, 2.5368,
              3.6979, 4.2700, 4.3938])

wavelength_m = 1.3829804621662048e-10
energy_eV = 12398.419843320026 / (wavelength_m * 1e10)

exp_3d = simulate_3d(
    det_type="manual",
    det_pxsize_h=0.000172,
    det_pxsize_v=0.000172,
    det_ntum_pixels_h=1475,
    det_num_pixels_v=831,
    det_dist=0.12,
    det_poni1=0.050,
    det_poni2=0.123,
    det_rotx=np.degrees(-0.023),
    det_roty=np.degrees(-0.5238161921742152),
    det_rotz=np.degrees(0.0),
    energy=energy_eV,
```

```
    q_hkls=q,
    hkls_names=hkls,
    cones_num_of_points=120,
)

exp_2d = simulate_2d(
    det_type="manual",
    det_pxsize_h=0.000172,
    det_pxsize_v=0.000172,
    det_ntum_pixels_h=1475,
    det_num_pixels_v=831,
    det_dist=0.12,
    det_poni1=0.050,
    det_poni2=0.123,
    det_rotx=np.degrees(-0.023),
    det_roty=np.degrees(-0.5238161921742152),
    det_rotz=np.degrees(0.0),
    energy=energy_eV,
    q_hkls=q,
    hkls_names=hkls,
    cones_num_of_points=720,
)

print(f"Energy: {energy_eV:.3f} eV")
print(exp_2d.detector.num_pixels_h, exp_2d.detector.num_pixels_v)
```

This example keeps the detector definition close to the calibration fields: distance, PONI coordinates, rotations, wavelength, pixel size, and maximum shape. Convert PONI-file rotations from radians before passing them to the current high-level simulation API.

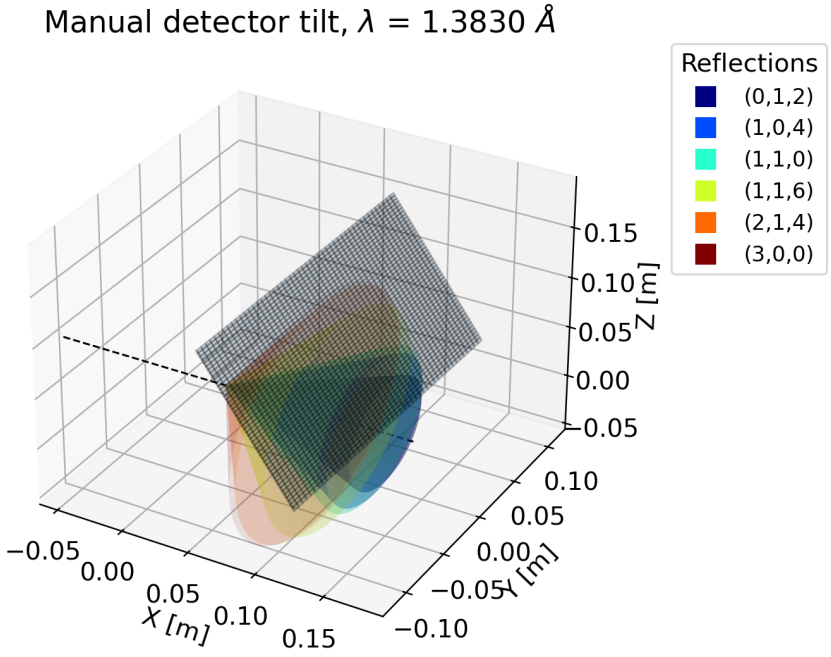


Figure 11.4: Complete simulation example C: three-dimensional powder-cone geometry with the manually defined tilted detector.

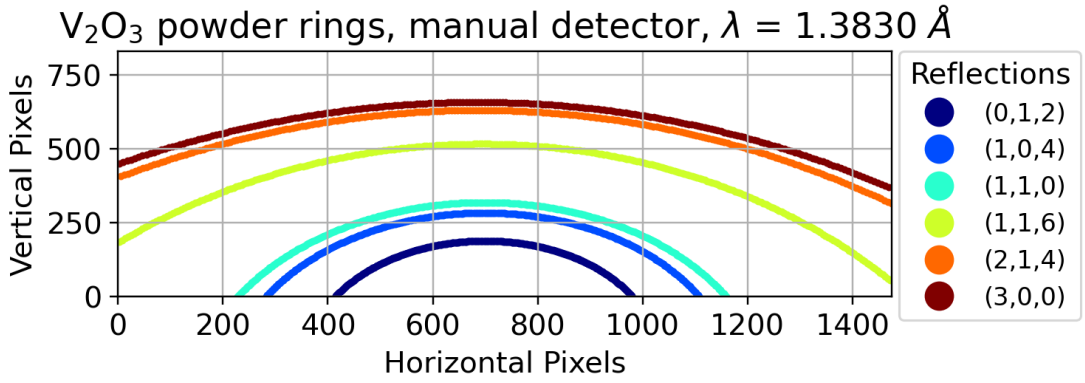


Figure 11.5: Complete simulation example C: detector projection from the same manually defined geometry.

11.5 Example D: geometry-aware single crystal

```
from trxrdpy.simulation.diffractometers import make_kappa_geometry
from trxrdpy.simulation.single_crystal import simulate_3d_with_geometry

geom = make_kappa_geometry(
    kappa_tilt_deg=50.0,
    sample_defaults={"omega": 0, "kappa": 0, "phi": 0},
    detector_defaults={"tth": 0},
)

exp = simulate_3d_with_geometry(
    det_type="manual",
    det_pxsize_h=75e-6,
    det_pxsize_v=75e-6,
    det_ntum_pixels_h=512,
    det_num_pixels_v=512,
    det_dist=0.30,
    det_poni1=0.0192,
    det_poni2=0.0192,
    energy=15000.0,
    e_bandwidth=1.0,
    sam_space_group=227,
    sam_a=5.431,
    qmax=5.0,
    geometry=geom,
    sample_angles={"omega": -30, "kappa": 40, "phi": 30},
    detector_angles={"tth": 25},
)
```

The example encodes the instrument motions by name. Before applying it to a real diffractometer, replace the generic axes and origins with the instrument definition.

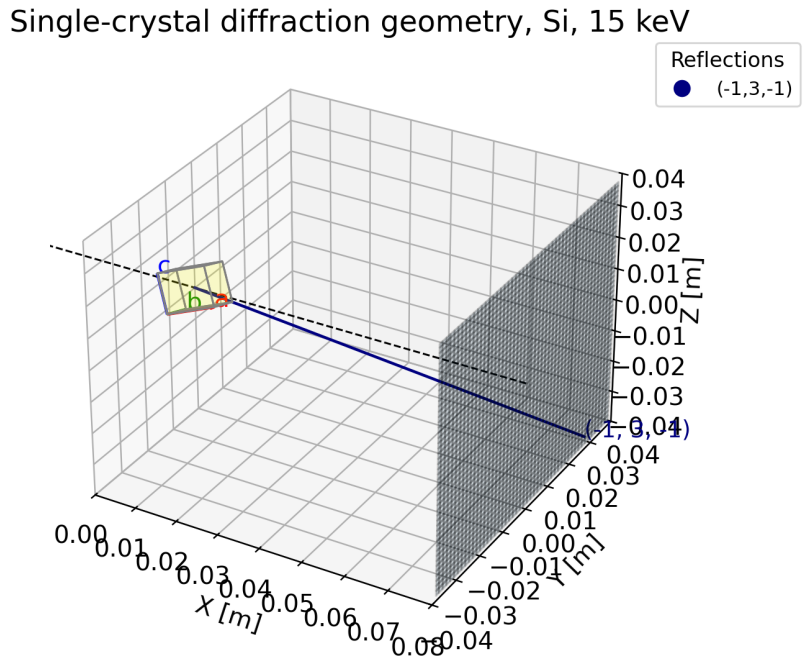


Figure 11.6: Complete simulation example D: three-dimensional single-crystal geometry used to check the beam, oriented reciprocal-lattice solution, and detector plane.

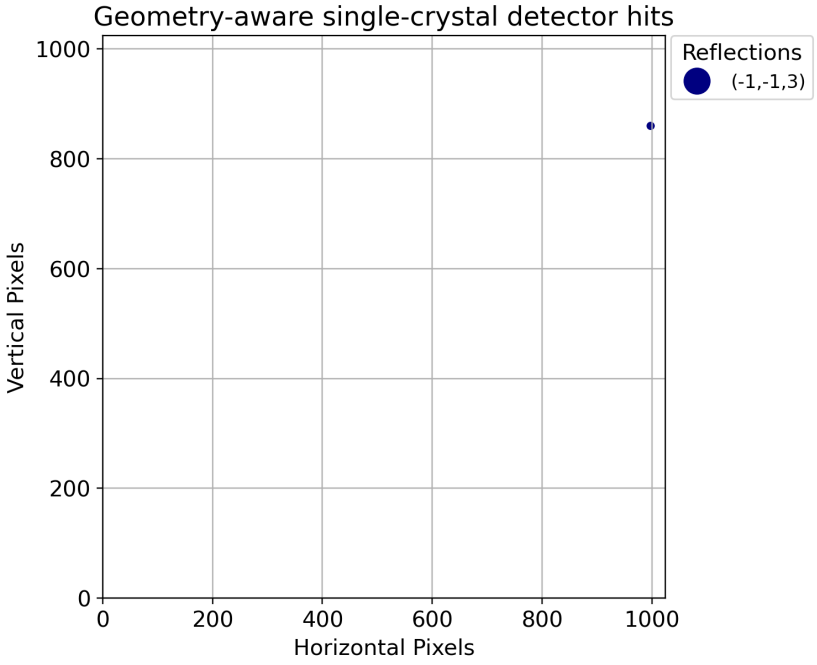


Figure 11.7: Complete simulation example D: geometry-aware detector projection generated from named sample and detector motors.

11.6 Example E: reproducible parameter record

Save a plain dictionary beside numerical or graphical results:

```
import json
```

```
import trxrdpy

record = {
    "package": "trxrdpy",
    "model": "polycrystalline.simulate_2d",
    "energy_eV": 15000.0,
    "bandwidth_percent": 1.0,
    "detector": {
        "type": "rayonixmx170",
        "distance_m": 0.10,
        "poni1_m": 0.085,
        "poni2_m": 0.085,
        "binning_hv": [4, 4],
    },
    "reflection_units": "q in inverse angstrom",
    "q_hkls": q.tolist(),
    "hkls": hkls.tolist(),
}

with open("simulation_parameters.json", "w", encoding="utf-8") as handle:
    json.dump(record, handle, indent=2)
```

If an installed version attribute is unavailable, record the distribution version using `importlib.metadata.version("trxrdpy")`. Include the CIF or its persistent identifier, PONI file or checksum, geometry definition, rotation conventions, and script revision for serious work.

Chapter 12

Simulation API reference

12.1 High-level modules

Callable	Purpose
`polycrystalline.simulate_1d`	Powder pattern from CIF; discrete peaks and optional convolved profile.
`polycrystalline.simulate_2d`	Project explicit q/d reflections as powder rings on a detector.
`polycrystalline.simulate_3d`	Display powder diffraction cones and detector geometry in 3D.
`single_crystal.simulate_2d`	Legacy-compatible single-crystal detector simulation.
`single_crystal.simulate_3d`	Legacy-compatible single-crystal 3D simulation.
`single_crystal.simulate_2d_with_geometry`	Single-crystal detector simulation with motor chains or transforms.
`single_crystal.simulate_3d_with_geometry`	Single-crystal 3D simulation with motor chains or transforms.
`single_crystal.sample_rotations_for_Bragg_condition`	Legacy sample `roty`/`rotz` orientation scan and rotation map.
`single_crystal.scan_two_motors_for_Bragg_condition`	Geometry-aware two-motor scan.
`single_crystal.scan_two_parameters_for_Bragg_condition`	Legacy/general two-parameter Bragg map.
`single_crystal.detector_rotations_collecting_Braggs`	Legacy detector `roty`/`rotz` collection scan.
`single_crystal.detector_rotations_collecting_Braggs_with_chain`	Arbitrary detector motor-chain collection scan.
`single_crystal.detector_rotations_collecting_Braggs_with_geometry`	Diffractometer detector-arm collection scan.
`single_crystal.target_hkl_near_pixel_fixed_energy`	Inverse orientation targeting for a reflection and detector pixel.

12.2 Core classes

Class	Role
`cif.Cif`	Validated extraction of cell, space group, and fractional coordinates.

`sample.LatticeStructure`	Direct/reciprocal lattice, reflections, orientation, Bragg tests, and powder data.
`detector.Detector`	Detector pixels, calibration, rotations, transforms, and lab grid.
`experiment.Experiment`	Combined beam, lattice, detector, diffraction directions, pixel positions, and plots.
`geometry.Motor`	One named axis rotation with origin and frame.
`geometry.MotorChain`	Ordered instrument rotations evaluated by motor-name angle maps.
`geometry.DiffractometerGeometry`	Paired sample-stage and detector-arm chains.
`poni.PoniGeometry`	Normalized immutable values parsed from a PONI file.
`utils.AxisRotation`	General axis-angle rotation object.
`utils.RotationChain`	Ordered general rotation composition.
`single_crystal.` `FixedEnergyTargetingResult`	Complete fixed-energy cone, detector-intersection, mask, best-hit, and solution result.
`single_crystal.` `FixedEnergyAcceptedSolutions`	Accepted Euler or named-motor orientation family and predicted detector pixels.

12.3 Geometry factories

`make_euler_geometry` Independent three-axis sample and detector chains using selectable Cartesian order and frame.

`make_legacy_euler_geometry` Alias of `make_euler_geometry`.

`make_kappa_geometry` Generic omega–kappa–phi sample stage and single-circle detector arm.

`make_diffractometer` Registry factory for predefined geometries.

12.4 Common utility functions

Function family	Purpose
`energy_to_wavelength`, `wavelength_to_energy`	Beam conversion between eV and meters.
`q_to_two_theta`, `d_to_two_theta`	Diffraction-coordinate conversion.
`axis_angle_to_matrix`, `euler_to_matrix`, `matrix_to_euler`	Rotation representations.
`make_transform`, `compose_transforms`, `invert_transform`	Homogeneous transformations.
`apply_rotation_matrix`, `apply_transform`	Apply orientations or transforms to arrays.
`diffractometer_to_dict` and related serializers	Convert geometry objects to JSON-compatible dictionaries.

12.5 Return objects and cached state

Many methods update object attributes instead of returning a new array. Examples include `Detector.calculate_lab_grid()`, `LatticeStructure.calculate_q_hkls()`, and `LatticeStructure.check_Bragg_condition()`. Read the documented attribute after the method call.

High-level simulations generally return an `Experiment`; powder 1D and search workflows instead return documented dictionaries or dataclasses whose named fields should be inspected directly.

Chapter 13

Validation, reproducibility, and troubleshooting

13.1 Validation hierarchy

Validate a simulation from simple quantities to complex output:

1. Confirm energy–wavelength conversion.
2. Confirm cell parameters, symmetry, and several known d spacings.
3. Confirm allowed reflections and equivalent-reflection behavior.
4. Confirm detector size, physical extent, distance, and beam center.
5. Confirm zero-angle geometry in 3D.
6. Apply one motor at a time and check its axis and sign.
7. Confirm a known reflection or calibrant pattern.
8. Only then run multidimensional scans or inverse targeting.

13.2 Common problems

13.2.1 Import fails

Confirm that installation and execution use the same Python interpreter. Use `python -m pip show trxrdpy` and print `sys.executable`. GUI import failures often indicate that the `gui` extra was not installed.

13.2.2 CIF cannot be read

Check the path, required cell fields, `_space_group_IT_number`, and atom-site fractional columns. If the file contains several blocks, remember that the first complete non-zero cell is selected.

13.2.3 No reflections satisfy Bragg condition

Check energy in eV, wavelength in meters, q in Å^{-1}, lattice orientation, `qmax`, and bandwidth in percent. Visualize reciprocal space before changing tolerances.

13.2.4 No reflections appear on the detector

A reflection may satisfy Bragg but miss the active detector. Check detector distance, dimensions, PONI coordinates, orientation, and image conventions. Use the 3D plot to distinguish a Bragg failure from an interception failure.

13.2.5 Powder rings are displaced or mirrored

Check PONI axis mapping, pixel order, image origin, horizontal sign, and detector rotations. Do not correct a convention mismatch by arbitrarily changing the beam center.

13.2.6 Geometry scan is unexpectedly slow

The number of evaluations grows multiplicatively with scan dimensions. Reduce reflection count, use coarser steps, narrow the range, and refine only around candidates. Avoid creating a full-resolution detector grid inside an unnecessary inner loop.

13.2.7 Changing angles has no effect

For a detector, call `calculate_lab_grid()` after changing angles. Also check whether a custom transform is active; it takes precedence over legacy Euler angles. Call `clear_transform()` to restore the Euler path.

13.2.8 Unexpected lattice state after repeated rotations

Check `mode`. Absolute rotations start from the initial orientation; relative rotations accumulate from the current one. Record the initial orientation and avoid mixing the two modes unintentionally.

13.3 Reproducibility record

For each published or operational simulation, retain:

- XRDpy version and Python environment;
- source script or GUI session;
- CIF and its source or checksum;
- detector type, PONI file, and binning;
- energy and bandwidth;
- coordinate definitions and rotation orders;
- diffractometer geometry and motor-angle maps;
- reflection selection and `qmax`;
- scan bounds, steps, seeds, and acceptance settings; and
- numerical results before graphical post-processing.

13.4 Known interface cautions

- The PyPI and import name is `trxrdpy`, not `xrdpy`.
- The high-level horizontal pixel-count keyword is currently spelled `det_ntum_pixels_h`.
- Several calculation methods store their result on the object and return `None`.
- A PONI file's wavelength metadata does not replace every explicit beam-energy argument.
- Generic predefined geometries must be validated before representing a real instrument.
- Analysis modules are independent of the Simulation workflow described here.

Part II

Analysis

Chapter 14

Analysis branch overview

14.1 Scope of the Analysis branch

The Analysis branch is the experimental-data half of XRDpy. It does not require the Simulation branch and it does not assume that a simulated pattern is available. Its job is to take beamline data and reduce it to reproducible scientific summaries: corrected detector images, azimuthally integrated one-dimensional patterns, fitted peak parameters, differential intensity traces, and publication-ready figures or tables.

The package is organized around the fact that pump-probe diffraction experiments have two kinds of complexity. The first is facility-specific: every beamline stores raw detector frames, timing information, metadata, and dark/reference conditions differently. The second is facility-independent: once the experiment has been reduced to calibrated two-dimensional images and standardized `xy` patterns, plotting, differential analysis, and peak fitting are mostly the same. The manual therefore treats facility preparation and shared analysis as separate layers.

Facility preparation
: The facility modules know how to read the raw beamline products. They resolve scan or run identifiers, apply timing-reference information, reject invalid shots, create dark or reference images, and average detector frames into standardized two-dimensional arrays.

Shared reduction and analysis
: The shared modules know how to work with calibrated images and `xy` patterns. This layer contains detector/cake diagnostics, pyFAI-based azimuthal integration, one-dimensional pattern visualization, differential analysis, and peak fitting.

GUI orchestration
: The Analysis GUI is not a separate scientific method. It is an interface that collects parameters and calls the same backend routines. Its value is session management, path consistency, form validation, and a single workflow for users who prefer interactive execution.

14.2 Supported facilities

The Analysis branch contains dedicated namespaces for ESRF ID09, Max IV FemtoMAX, and SPring-8 / SACLA. These names are not simple labels; each namespace reflects a different raw-data model.

FemtoMAX is handled through a wrapper layer that accepts modern explicit path objects while remaining compatible with local configuration-driven workflows. The beamline and MAX IV context are described in Refs. [4, 5]. In practice, users may still keep experiment-specific defaults in a local `config.py` or equivalent setup file, but the reduction functions should receive resolved paths explicitly when a workflow is documented. The module resolves timing-reference pings,

builds standardized metadata HDF5 files, and exports averaged two-dimensional images for dark, delay, and fluence workflows.

ESRF ID09 uses BLISS/txs raw data. Time-resolved ID09 pump-probe operation and chopper-based timing infrastructure are described in Refs. [1, 2, 3]. The ID09 routines open the BLISS dataset, select frames by delay token, average selected frames, and write standardized dark or delay-resolved images. ID09 also has an integration layer that understands ID09 delay-token conventions and cached delay points.

SACLA support is more facility-legacy and infrastructure-dependent. The facility context and hard-X-ray XFEL timing diagnostics are described in Refs. [6, 7, 8]. The code relies on SACLA database and timing utilities, reads tag-level metadata, applies shutter and beam-status filters, constructs delay lists, creates background or dark images, and can process large runs in chunks suitable for HPC execution. The SACLA integration namespace then exposes the shared two-dimensional image integration workflow.

Because these facilities differ substantially, raw-data preparation is selected by facility. Identify the facility first, create the correct intermediate products, and only then move into the shared calibration, integration, differential-analysis, and fitting chapters.

14.3 Analysis data flow

The full analysis path is a staged pipeline. A user normally starts by defining where the experiment lives and which facility produced the data. The facility-specific step then converts raw detector frames into analysis-ready two-dimensional images. Once those images exist, the rest of the workflow is deliberately standardized.

1. define the experiment paths and facility;
2. create or locate metadata, dark frames, and corrected two-dimensional images;
3. validate detector calibration, PONI files, masks, and azimuthal conventions;
4. integrate the selected images into standardized `xy` patterns;
5. inspect absolute and differential patterns;
6. calculate differential integrals or fit selected peaks;
7. summarize delay, fluence, or multi-experiment trends; and
8. export figures, tables, and reproducibility records.

The important practical point is that the transition from facility-specific to shared analysis happens at the standardized two-dimensional image and `xy` pattern level. If a workflow fails before that point, the likely problem is raw data, metadata, timing, or path layout. If it fails after that point, the likely problem is detector geometry, mask/PONI setup, q range, azimuthal windows, normalization, fitting windows, or reference selection.

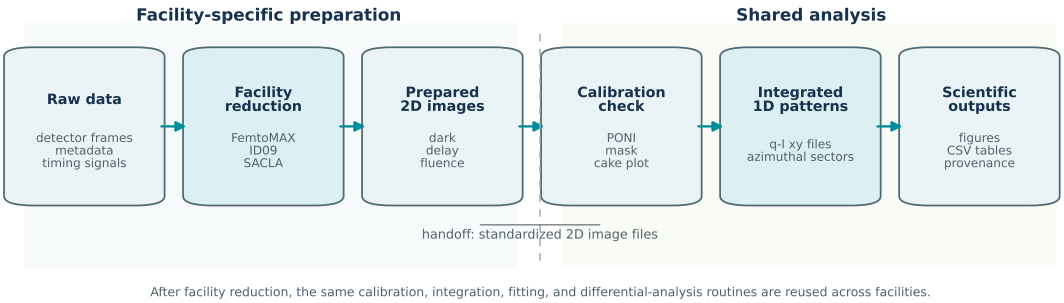


Figure 14.1: Analysis data flow from facility raw data to standardized images, azimuthal integration, one-dimensional viewing, differential analysis, fitting, and exported results. The dashed boundary marks the handoff from beamline-specific preparation to shared analysis routines.

14.4 Programmatic and GUI routes

Programmatic workflows are the reference implementation for reproducible analysis. They expose explicit paths, scan identifiers, q ranges, azimuthal windows, normalization settings, and output names. They are best for scripts, publications, and batch processing.

The GUI is useful when parameter choices are still being explored. It keeps the same state across tabs, logs backend calls, and helps users avoid inconsistent path and facility settings. A good practical workflow is to use the GUI to inspect detector images, cakes, and initial fits, then transfer the final parameter values into a scripted workflow for reproducibility.

When documenting or publishing an analysis, report the backend function and the parameter values. A GUI screenshot is useful for communication, but it is not a reproducibility record by itself.

Chapter 15

Data organization and file conventions

15.1 Experiment root and analysis root

Analysis functions distinguish raw inputs from generated products. The central object is `trxrdpy.analysis.common.paths.AnalysisPaths`. It stores one experiment root, a raw-data subdirectory, and an analysis-output subdirectory. The resulting `raw_root` is where facility readers look for beamline files; the resulting `analysis_root` is where XRDpy writes processed images, cached `xy` files, fit tables, and figures.

The preferred style is to pass `paths=AnalysisPaths(...)` explicitly. Many functions also support legacy arguments such as `path_root`, `raw_subdir`, and `analysis_subdir`. These routes are equivalent only if they resolve to the same filesystem locations. For reproducibility, avoid relying on the current working directory; make the raw and analysis roots visible in the script.

The `AnalysisPaths` class does not create directories by itself. It only resolves locations. Directory creation occurs in the facility or analysis routines when outputs are written.

`experiment_root/`	Experiment-level root folder.
`  RAW_DATA/`	Beamline raw files, scans, runs, or BLISS/SACLA/FemtoMAX inputs.
`  calibration/`	Shared calibration files.
`    detector.poni`	pyFAI detector geometry.
`    mask.edf`	Detector mask.
`  analysis/`	Generated and reusable analysis products.
`    <sample>/temperature_<T>K/`	Sample and temperature branch.
`      dark/<scan_tag>/`	Dark or calibration branch.
`        2D_images/`	Averaged dark or calibration images.
`        xy_files/`	Dark/calibration one-dimensional patterns.
`        metadata/`	Facility metadata and provenance.
`        figures/`	Detector, cake, fit, and diagnostic figures.
`      excitation_wl_<wl>nm/delay/`	Delay-scan branch.
`        fluence_<F>mJ/time_window_<tw>fs/`	One fluence and integration-time-window branch.
`          2D_images/`	Delay-resolved averaged images.
`          xy_files/`	Cached `q, intensity` patterns.
`          fitting/`	Peak-fit CSV tables.
`          figures/`	Viewer, differential, and fit plots.
`      excitation_wl_<wl>nm/fluence/`	Fluence-scan branch.
`        delay_<d>fs/time_window_<tw>fs/`	One delay and integration-time-window branch with the same `2D_images/`, `xy_files/`, `fitting/`, and `figures/` outputs.

Figure 15.1: Typical Analysis directory tree. Facility-specific raw data are kept below the raw root, while the analysis root stores standardized images, cached `xy` files, metadata, fit tables, and figures using sample, temperature, excitation, fluence, delay, and time-window tags.

15.2 Scan specifications and tags

Scan identity appears throughout the Analysis branch. For simple workflows a scan may be a single integer. For combined dark scans or averaged references it may be a list of integers. For cached outputs it may be a string such as `scan_167246` or a combined scan tag. Facility modules may also require a dataset, run number, BLISS scan number, or SACLA tag list.

The package uses stable scan tags to make filenames predictable. A single scan can be represented as a scan tag; a group of scans can be represented as a combined scan tag. These tags matter because later steps may reuse cached `xy` files or fit CSV files rather than recomputing them. If a user changes from scan `7` to scans `[7, 8]` but reuses an old output path, the analysis can silently compare the wrong products. Good examples show both the human scan selection and the tag or output folder it creates.

15.3 Standardized intermediate products

The shared pipeline is built around a small number of intermediate products. Keeping these products standardized is what allows different beamline preparations to use common integration, plotting, differential-analysis, and fitting code.

Two-dimensional image arrays
: Facility routines write averaged images, typically as NumPy arrays, for dark conditions, delay points, or fluence points. These images are the handoff point between raw-data reduction and pyFAI integration.

One-dimensional `xy` patterns
: Azimuthal integration writes two-column text files. The first column is the radial coordinate, normally q in $Å^{-1}$, and the second column is intensity. These files are cached so plotting and fitting do not need to reintegrate the detector image every time.

Metadata files
: Facility preparation may write HDF5 or CSV metadata describing scan identity, frame validity, delay bins, fluence values, temperature, excitation wavelength, and timing windows. These files are part of the scientific record, not temporary implementation details.

Result tables
: Differential analysis and peak fitting write CSV tables. These tables are the numerical outputs used for downstream plotting, error checking, and publication figures.

15.4 Reproducibility records

Every complete analysis records the raw-data selection, facility, sample name, temperature, excitation wavelength, fluence, time window, dark or reference selection, PONI file, mask file, azimuthal offset, polarization factor, q normalization range, azimuthal windows, fitting windows, output directory, and package version.

This information is not optional bookkeeping. Time-resolved diffraction results are sensitive to reference choice, azimuthal sector, mask, q range, and normalization. Two analyses can use the same raw scan and produce different conclusions if these values differ. Manual examples must therefore let the user reconstruct the path from raw data to final plot.

Chapter 16

Facility-specific 2D image production

Facility boundary

Facility modules are responsible for reaching a shared representation. The shared routines do not need to know the full raw-data layout of each beamline.

The examples in this chapter are launch examples. They show where each facility-specific backend is called and which products are expected before the workflow moves to shared calibration, integration, differential analysis, or fitting.

16.1 Max IV FemtoMAX

FemtoMAX is a time-resolved hard X-ray beamline at MAX IV Laboratory in Lund, Sweden [4, 5]. In the context of this manual, it represents synchrotron pump-probe experiments where detector images must be associated with timing-reference information before they can be averaged into delay or fluence conditions. The relevant facility-specific issues are therefore path layout, scan numbering, ping references, delay-bin definitions, and the distinction between dark, delay, and fluence scans.

FemtoMAX is the most modern facility wrapper in the Analysis branch. The user-facing module `trxrdpy.analysis.MaxIV_FemtoMAX.datared` is intentionally thin: it constructs an experiment object, resolves paths, loads timing-reference information, creates metadata HDF5 files, and exports averaged two-dimensional detector images. The wrapper accepts either an `AnalysisPaths` object or explicit path arguments. If a local `config.py` is used to collect site- or proposal-specific defaults, treat it as a convenience layer that prepares those values, not as the scientific record. The script or notebook should still expose the resolved raw path, analysis path, scan numbers, ping-reference table, and selected delays.

The FemtoMAX workflow begins with a scan or scan collection and a timing-reference table. The packaged reference table can be used by default, or a custom CSV can be supplied. The reference table is validated before use; overlapping scan ranges, missing columns, and malformed ping values are rejected. The timing reference is used to center corrected delay distributions and to define which frames belong to each delay bin.

FemtoMAX distinguishes three scan types. A `dark` scan creates metadata and averaged images without pump metadata. A `delay` scan uses excitation wavelength, fluence, and a time window to group frames by corrected delay. A `fluence` scan is organized at one or more explicit selected delays, with fluence values aligned to scans. The code intentionally rejects `selected_delays="auto"` for fluence exports, because a fluence series must be associated with a known delay.

The normal sequence is to inspect timing quality with `plot_pings_distribution`, create metadata with `create_h5_files`, and export averaged images with `generate_2D_imgs`. The export step

can run serially or in parallel. Parameters such as `batch_size`, `max_workers`, `chunk_size`, and multiprocessing start method control the balance between speed and memory use. Once the two-dimensional images exist, the FemtoMAX azimuthal-integration namespace delegates to the shared two-dimensional image workflow.

16.1.1 FemtoMAX delay-scan 2D image production

The example below shows the FemtoMAX reduction call used for a DET58 delay scan at 300 K, 1500 nm, and 52 mJ cm^{-2}. The first call plots the corrected ping distribution for the selected scans. The second call creates or validates the metadata HDF5 file and exports averaged two-dimensional images into the standardized analysis tree.

The delay source is `p2`. The selected delays are explicit because these values define the image folders that the later shared azimuthal-integration functions read. The same call can be run on a workstation or cluster node that has access to the raw FemtoMAX files.

```
from pathlib import Path

import numpy as np

from trxrdpy.analysis.common.paths import AnalysisPaths
from trxrdpy.analysis.MaxIV_FemtoMAX import datared as femtomax_datared

paths = AnalysisPaths(
    path_root=Path("/path/to/FemtoMAX2026"),
    raw_subdir="raw",
    analysis_subdir="analysis",
)

sample_name = "DET58"
temperature_K = 300
excitation_wl_nm = 1500
fluence_mJ_cm2 = 52
time_window_fs = 1000
scan_type = "delay"
delay_source = "p2"
nb_shot_threshold = None
overwrite = True

scans = np.arange(181060, 181092 + 1)
selected_delays = [
    -98972, -49132, -32079, -23519, -15817, -12481,
    -11000, -9448, -8190, -6684, -5268, -2871,
    -1536, 1253, 2622, 4090, 6000, 8732, 10792,
    12662, 15292, 17528, 19314, 21927, 26283,
    30691, 40272, 57032, 84809, 200391,
]

femtomax_datared.plot_pings_distribution(
    scans=scans,
    mode="overlay",
    delay_source=delay_source,
    unit="fs",
    view="scatter",
    require_both=True,
    paths=paths,
)
```

```
experiment, exported = femtomax_datared.generate_2D_imgs(
    scans=scans,
    sample_name=sample_name,
    temperature_K=temperature_K,
    excitation_wl_nm=excitation_wl_nm,
    fluence_mJ_cm2=fluence_mJ_cm2,
    time_window_fs=time_window_fs,
    scan_type=scan_type,
    selected_delays=selected_delays,
    delay_source=delay_source,
    require_both=True,
    nb_shot_threshold=nb_shot_threshold,
    overwrite=overwrite,
    batch_size=500,
    use_parallel=True,
    max_workers=5,
    chunk_size=1,
    start_method="fork",
    paths=paths,
)
```

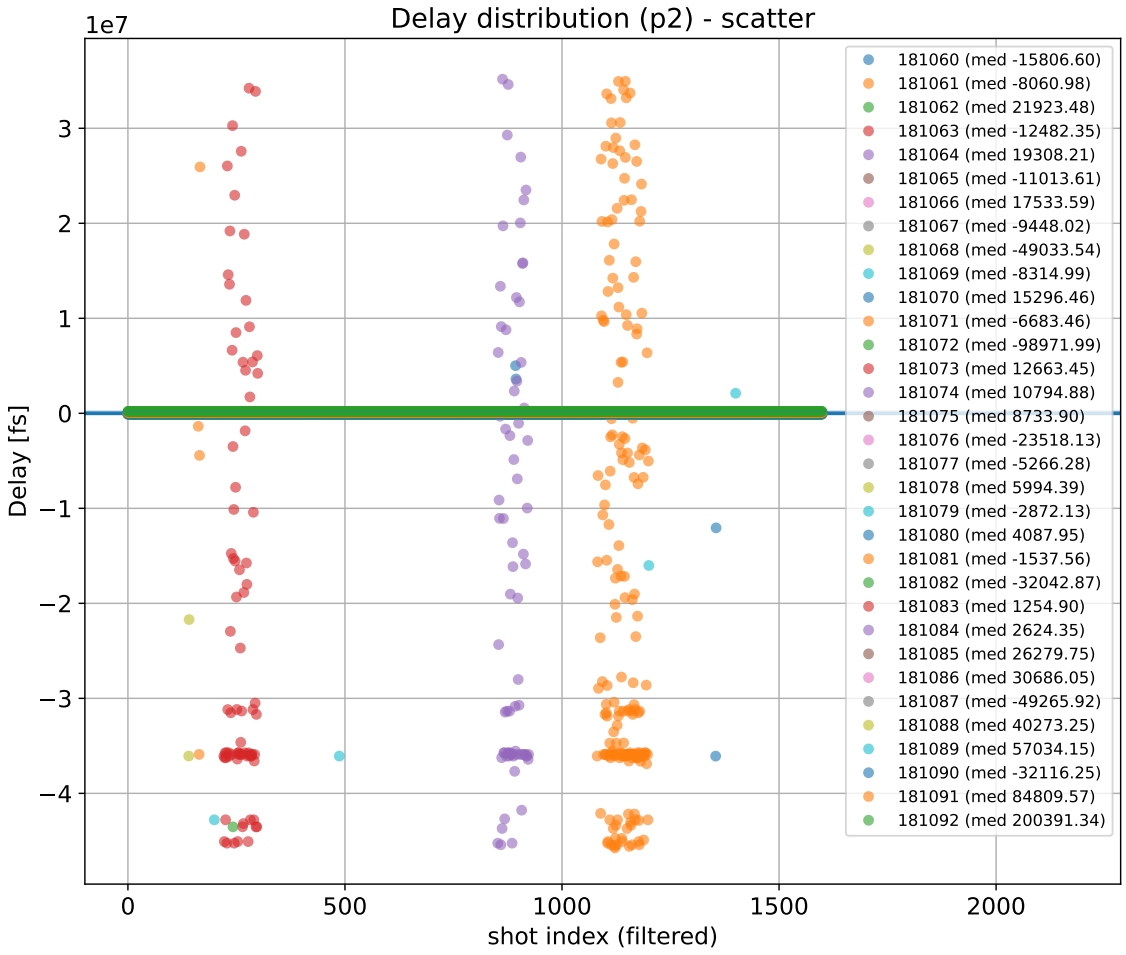


Figure 16.1: FemtoMAX DET58 p2 timing distribution for scans 181060–181092, shown over the full delay range.

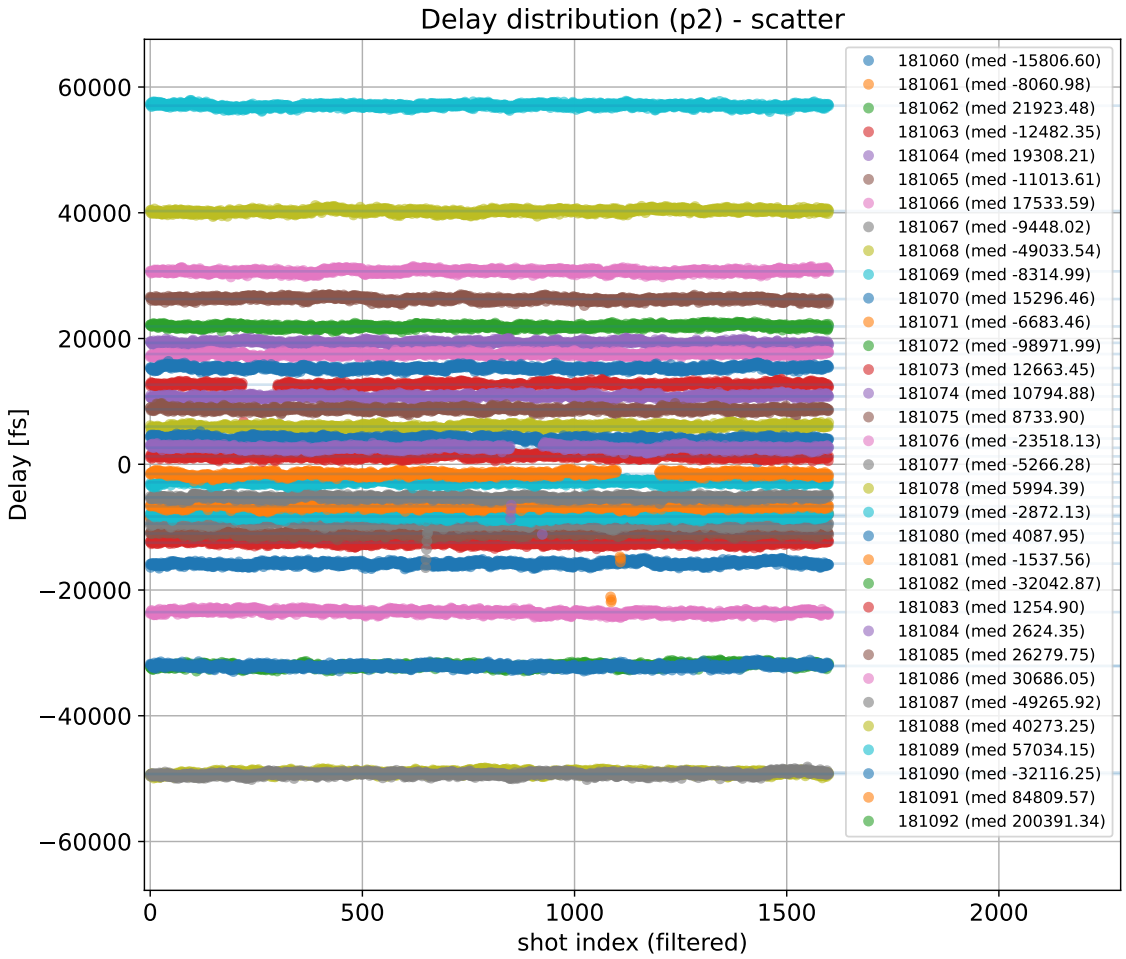


Figure 16.2: FemtoMAX DET58 `p2` timing distribution with the vertical axis restricted to the central delay region, making the main scan-to-scan timing structure visible.

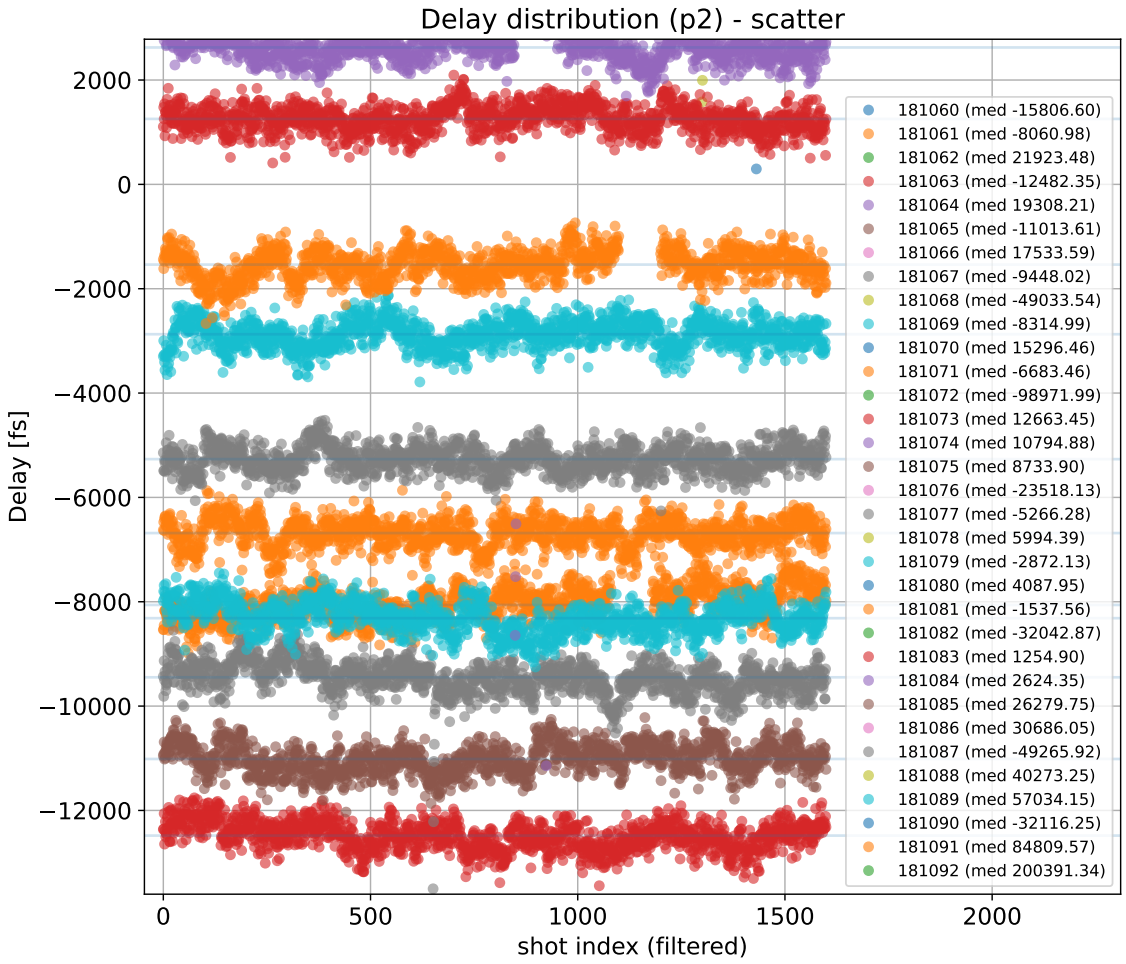


Figure 16.3: FemtoMAX DET58 p2 timing distribution zoomed around the near-time-zero region, where the shot-to-shot timing jitter is visible on the femtosecond scale.

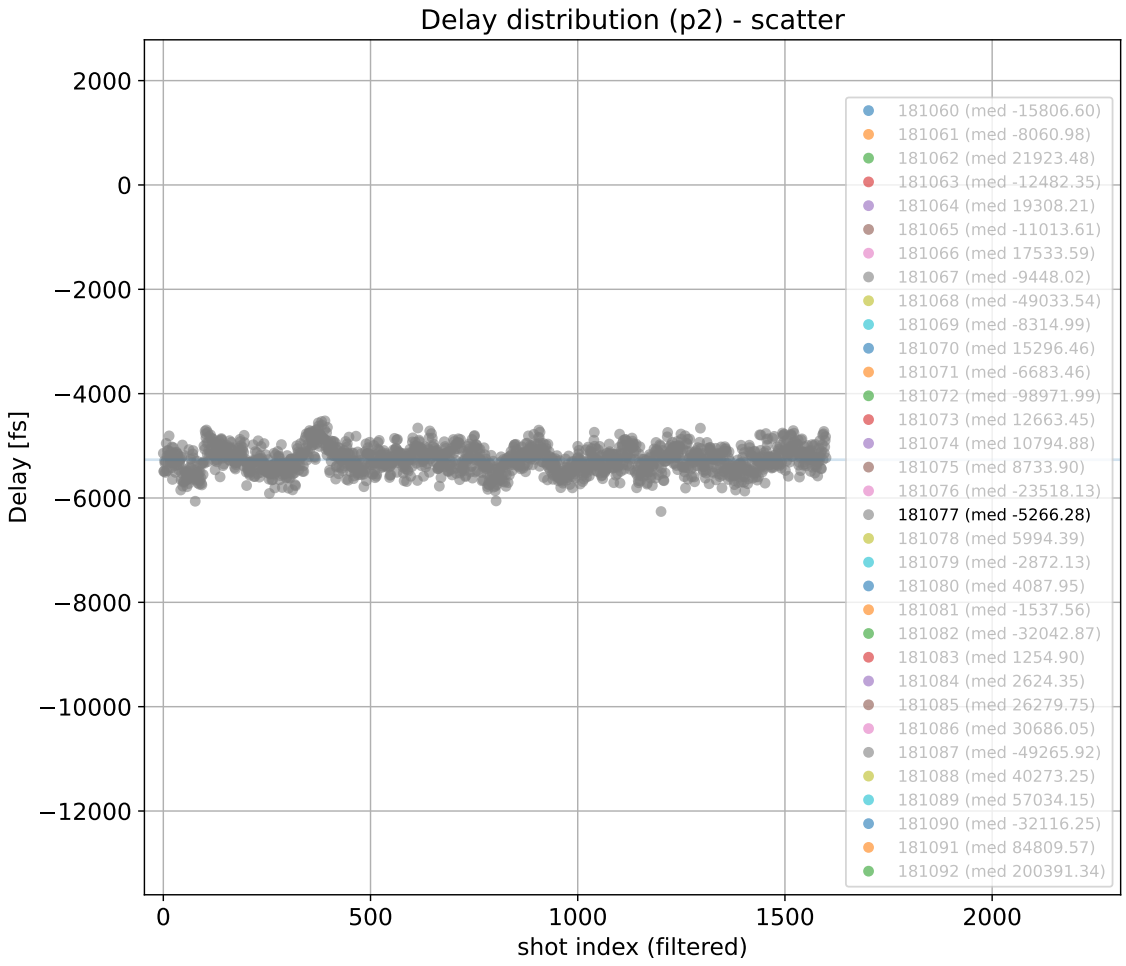


Figure 16.4: FemtoMAX DET58 `p2` timing distribution for the isolated scan 181077, used to inspect the local timing spread around the −5266.28 fs median delay.

16.2 ESRF ID09

ID09 is an ESRF beamline context associated here with time-resolved diffraction and scattering experiments at the European Synchrotron Radiation Facility in Grenoble, France [1, 2, 3]. The practical consequence for XRDpy users is that the raw data arrive through the ESRF data stack rather than through the FemtoMAX or SACLA file models. The module therefore focuses on BLISS datasets, txs delay metadata, scan keys, raw sample naming, and delay-token normalization.

The ESRF ID09 analysis namespace is built around BLISS/txs data access. The optional `pytxs` dependency is the ID09 local analysis tool used by XRDpy for parts of the raw-data handling layer; it is distributed separately as `pytxs` on PyPI [17]. The reduction functions open a BLISS dataset file located in the ID09 raw-data tree, select a scan key such as `<scan>.1`, read delay metadata, and average detector frames matching selected delay tokens. The raw sample name can be different from the analysis-facing sample name, so the functions support a `raw_sample_name` override.

ID09 delay handling is token-based. The raw metadata delays are formatted through txs utilities into canonical strings, and user selections are normalized to the same representation. This matters because the same physical delay may appear as a string token, a byte string, or a numeric value depending on where it enters the workflow. The reduction code standardizes these representations before selecting frames.

The function `get_2D_img` averages frames for one delay and returns the image without writing it. `create_dark_from_ref_delay` averages a reference-delay selection and writes it into the standardized dark-image structure. `create_final_2D_images` exports averaged detector images for all or selected delay points, converts delay tokens to femtoseconds, and writes the output using the shared delay-series naming convention.

The ID09 azimuthal-integration module adds ID09-specific logic on top of this prepared-image layer. It resolves calibration paths, lists available cached delay points, integrates selected delays, plots absolute and differential one-dimensional patterns, and can create a fluence scan from a set of delay scans. This makes ID09 slightly different from FemtoMAX and SACLA: it has both facility-specific image preparation and facility-specific integration conveniences.

16.2.1 ID09 2D image production example

The ID09 preparation example starts from a BLISS dataset and scan number. The reference-delay image is written as a standardized dark image, and selected laser-on delays are written as standardized delay images. Downstream chapters can then use the cached `xy` files if they already exist, or integrate from these two-dimensional images if the cache is missing.

```
from pathlib import Path

from trxrdpy.analysis.common.paths import AnalysisPaths
from trxrdpy.analysis.ESRF_ID09 import datared as id09_datared

paths = AnalysisPaths(
    path_root=Path("/path/to/ESRF/ID09/proposal"),
    raw_subdir="RAW_DATA",
    analysis_subdir="analysis",
)

dark_path = id09_datared.create_dark_from_ref_delay(
    sample_name="NiS2",
    raw_sample_name="NiS2",
    dataset=1,
```

```
    scan_nb=7,
    delay_ref="-5ns",
    temperature_K=300,
    paths=paths,
    overwrite=True,
)

delay_image_paths = id09_datared.create_final_2D_images(
    sample_name="NiS2",
    raw_sample_name="NiS2",
    dataset=1,
    scan_nb=7,
    temperature_K=300,
    excitation_wl_nm=1400.0,
    fluence_mJ_cm2=10.7,
    time_window_fs=100000,
    delays=["100ps", "500ps", "1ns"],
    paths=paths,
    overwrite=True,
)
```

16.3 SPring-8 / SACLA

SACLA is the X-ray free-electron laser facility operated at the RIKEN SPring-8 Center in Harima, Japan, next to the SPring-8 synchrotron [6, 7]. The experiments represented by this namespace are closer to XFEL-style shot or tag processing than to a simple synchrotron image stack: each detector frame must be interpreted together with facility tags, timing metadata, shutter state, beam status, and background or laser-off information.

The SACLA reduction code reflects a more legacy and facility-dependent environment. It uses SACLA-specific packages such as `stpy` and `dbpy`, and it bootstraps paths from environment variables: `XRDPY_PATH_ROOT`, `XRDPY_ANALYSIS_SUBDIR`, and `XRDPY_TIME_METADATA_SUBDIR`. This is different from the more self-contained FemtoMAX wrapper, so SACLA preparation is best treated as a beamline-environment workflow.

SACLA metadata are tag-based. The reduction reads tag lists and synchronized facility parameters, then merges them with timing-tool CSV metadata. For delay and fluence scans it filters invalid shots using X-ray shutter status, laser shutter status, and beam-status flags. The delay calculation combines timing-edge information and programmed motor position, with timing diagnostics related to the branching method described in Ref. [8]. In the current implementation the timing-edge contribution is scaled by -2.7 and the programmed delay motor by 6.67, producing a delay column in femtoseconds.

The SACLA preparation layer supports creating delay lists, processing background runs, creating dark images from laser-off frames, and creating final two-dimensional images. Because SACLA datasets can be large, the module also contains chunked processing and a PBS job-submission script for cluster execution. The goal is the same as for the other facilities: produce corrected two-dimensional images in the analysis tree so that shared azimuthal integration can operate on them.

For SACLA, document the reduction environment carefully. The database access layer, timing metadata folder, run number, beamline identifier, delay-bin definition, and background/dark strategy are all part of the analysis provenance.

16.3.1 SACLA 2D image production example

SACLA preparation depends on the local SACLA environment. Set the required path environment before importing the SACLA reduction module, create metadata for the selected run or run group, process the no-X-ray background run when one is used, and then create the final delay, fluence, laser-off, or differential image product.

```
import os

os.environ["XRDPY_PATH_ROOT"] = "/path/to/SACLA/proposal"
os.environ["XRDPY_ANALYSIS_SUBDIR"] = "analysis"
os.environ["XRDPY_TIME_METADATA_SUBDIR"] = "TM_data"

from trxrdpy.analysis.Spring8_SACLA import datared as sacla_datared

metadata_h5 = sacla_datared.create_metadata(
    bl=3,
    run=[1466583, 1466578],
    sample_name="sample",
    temperature_K=300,
    scan_type="delay",
    time_window_fs=250,
    excitation_wl_nm=800,
    fluence_mJ_cm2=1.0,
    time_step=250,
    overwrite=True,
)

sacla_datared.process_background_run(
    bl=3,
    run=1466500,
    overwrite=True,
)

delay_image = sacla_datared.create_final_2D_images(
    bl=3,
    scan_type="delay",
    metadata_h5_path=metadata_h5,
    delay=1000,
    background=1466500,
    detector_id="MPCCD-8N0-3-002",
    threshold_counts=40,
    save_laser_off=True,
    overwrite=True,
)
```

The SACLA one-dimensional viewer, differential, fitting, and multi-experiment analysis examples are presented later in Chapter 21, after the shared analysis pipeline chapters.

16.4 Facility comparison

The table below summarizes the practical differences between the three facility layers. It is not meant to hide the details; it is meant to show where each facility enters the shared workflow.

Facility	Raw input concept	Main preparation output	Shared handoff
FemtoMAX	scans plus ping references	metadata HDF5 and averaged 2D arrays	shared 2D azimuthal integration
ESRF ID09	delay-resolved scan source	dark/final images and cached delay patterns	ID09 integration or shared plotting
SACLA	beamline run/tag data	corrected 2D images by delay or condition	shared 2D azimuthal integration

Chapter 17

Calibration, masks, and detector diagnostics

17.1 Calibration inputs

Calibration and integration require a consistent detector geometry. In practice this means more than having a PONI file. The user must know which detector image is being integrated, which mask is being applied, how detector pixels are displayed, and how the package's azimuthal convention maps onto pyFAI's convention.

- a PONI file;
- an optional detector mask;
- a calibration or dark image;
- a radial coordinate convention; and
- an azimuthal coordinate convention.

The calibration module can discover or accept explicit PONI and mask paths. EDF masks and related detector-image files are read through fabio where appropriate [11, 12]. The compatibility loader can patch narrowly scoped legacy PONI issues, such as unsupported PONI-version syntax or detector `orientation` metadata. This patching is intentionally conservative and records the changes applied. A patched PONI file makes pyFAI loading possible; it does not prove that the experiment geometry is scientifically correct.

The parameter `azim_offset_deg` is central. It maps the package display-coordinate azimuth convention to the pyFAI coordinate convention. The default value in many workflows is $-90°$, but verify this against a calibration image or a known anisotropic feature. The optional `polarization_factor` follows pyFAI's convention and must be either `None` or a finite number between -1 and 1.

17.2 Detector and cake diagnostics

Detector and cake plots are the first quality-control step after path and calibration setup. The detector panel shows the image in pixel coordinates. The cake panel shows the same image after pyFAI remapping into radial coordinate and azimuth. If the PONI file, mask, axis convention, or azimuthal offset is wrong, the cake plot usually exposes the problem before it contaminates one-dimensional integrations.

The function `calibration.plot_detector_and_cake` computes this diagnostic. It supports independent detector and cake color limits, logarithmic display, optional detector-axis flips, row-wise normalization over a q interval, explicit radial and azimuthal ranges, and saving. Axis flips affect only the displayed detector panel; they do not change the integration arrays. This distinction is important because display correction is not the same as changing the scientific geometry.

The example in Figure 17.1 uses the ESRF ID09 NiS2 dark scan at 300 K with the corresponding room-temperature PONI file and EDF mask. The plotted settings follow the saved GUI state for this dataset: the EDF mask is applied, the detector display uses the saved y-axis flip, the detector color limits are $(0, 100)$, and the cake color limits are $(0, 2)$. These are display choices for inspection; they do not modify the calibrated geometry used later for integration.

```
from pathlib import Path

from trxrdpy.analysis import calibration

fig, axes, data = calibration.plot_detector_and_cake(
    sample_name="NiS2",
    scan=7,
    temperature_K=300,
    npt_rad=1000,
    npt_azim=360,
    azimuthal_range=(-90.0, 90.0),
    normalize=True,
    q_norm_range=(2.9308, 3.2827),
    use_mask=True,
    poni_path=Path("/path/to/NiS2_Room_T.poni"),
    mask_edf_path=Path("/path/to/Mask_NiS2_Room_T.edf"),
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    detector_clim=(0, 100),
    cake_clim=(0, 2),
    invert_detector_y=True,
    save=True,
    path_root=Path("/path/to/ESRF2023"),
    analysis_subdir="analysis",
)
```

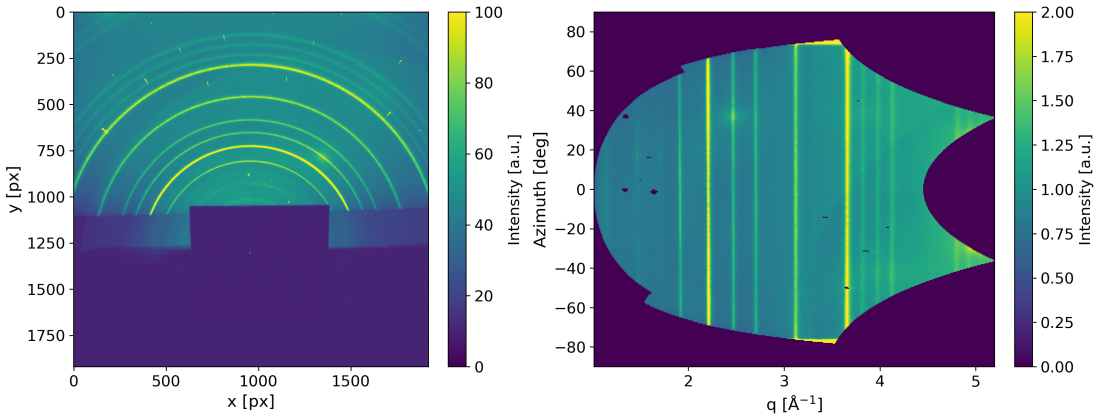


Figure 17.1: ID09 NiS2 detector image and two-dimensional cake for the 300 K dark scan, using the saved GUI display settings: detector y-axis flip, EDF mask, detector color limits $(0, 100)$, and cake color limits $(0, 2)$.

17.3 Azimuthal windows

Many workflows operate over azimuthal windows rather than over the full detector. Windows are used to separate anisotropic scattering, avoid detector artifacts, or compare symmetry-related sectors. The package supports two common representations. Edge-based windows use an ordered

list of azimuthal edges and create adjacent sectors. Explicit window lists use individual (ϕ_0, ϕ_1) pairs.

Calibration and integration routines can also include a full-range pattern in addition to segmented sectors. The full-range pattern is convenient for quick inspection, but sector-resolved patterns are often more diagnostic. If a sector is used for differential analysis or fitting, the sector definition must be reported with the result. A peak amplitude integrated over $-90°$ to $90°$ is not directly comparable to one integrated over a narrower sector unless the normalization and interpretation are explicit.

Figure 17.2 uses the same NiS2 calibration context as the detector/cake diagnostic. The full range is $(-90°, 90°)$, and the segmented windows follow the saved GUI edge list $[-90, -70, -50, -30, -10, 10, 30, 50, 70, 90]$. The function `calibration.plot_cake_azimuthal_windows` computes the cake with the same pyFAI path as `calibration.plot_detector_and_cake`, overlays adjacent azimuthal sectors, and optionally adds the sector side bar used in the figure.

```
from trxrdpy.analysis import calibration

fig, axes, data = calibration.plot_cake_azimuthal_windows(
    sample_name="NiS2",
    scan=7,
    temperature_K=300,
    npt_rad=1000,
    npt_azim=360,
    azimuthal_range=(-90.0, 90.0),
    normalize=True,
    q_norm_range=(2.9308, 3.2827),
    use_mask=True,
    poni_path="/path/to/NiS2_Room_T.poni",
    mask_edf_path="/path/to/Mask_NiS2_Room_T.edf",
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    azimuthal_edges=[-90, -70, -50, -30, -10, 10, 30, 50, 70, 90],
    full_range=(-90, 90),
    cake_clim=(0, 2),
    show_side_bar=True,
    save=True,
    path_root="/path/to/ESRF2023",
    analysis_subdir="analysis",
)
```

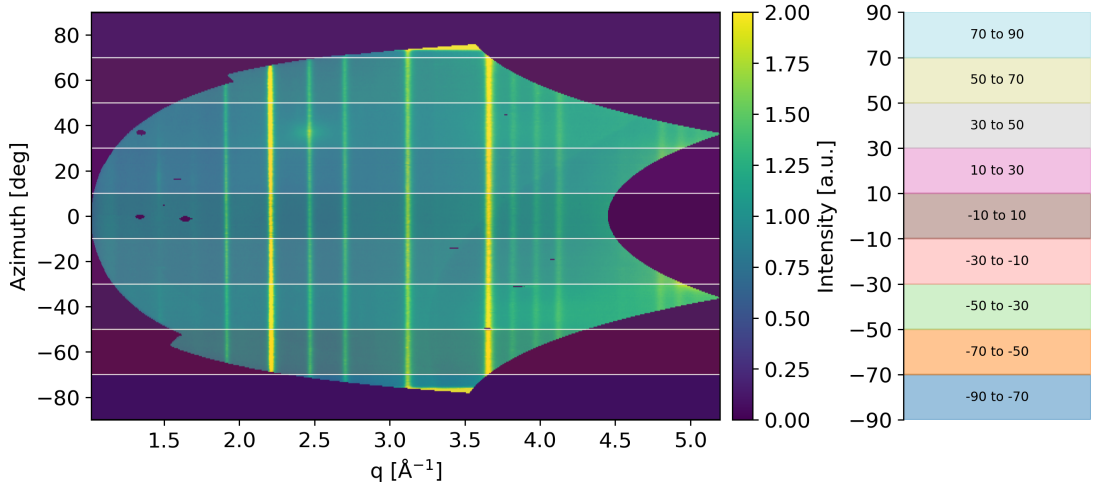


Figure 17.2: ID09 NiS2 cake with the saved segmented azimuthal windows overlaid. The side bar lists each sector from the GUI edge definition, while the full $(-90°, 90°)$ range is retained as the reference full-range integration window.

17.4 Calibration peak fitting

Calibration peak fitting quantifies whether a calibration or dark image behaves consistently across azimuth. The function `calibration.do_peak_fitting` integrates the calibration image into one-dimensional patterns, fits a linear background plus pseudo-Voigt peak in each azimuthal window, and writes a CSV table of fit parameters and quality metrics. Choose and record the fitting range, pseudo-Voigt fraction, normalization range, and fit method.

The most useful diagnostic is often not the individual fit overlay, but the trend of fitted peak position, FWHM, amplitude, or area versus azimuth. Smooth trends can indicate geometry or sample anisotropy; abrupt jumps usually indicate bad sectors, masking problems, or a fitting range that is too narrow. The functions `plot_property_vs_azimuth`, `plot_1D_plus_fit`, and `compare_1D_patterns` support this review.

```
from trxrdpy.analysis import calibration

fit_table = calibration.do_peak_fitting(
    sample_name="NiS2",
    scan=7,
    temperature_K=300,
    q_fit_range=(3.11, 3.135),
    azimuthal_ranges=[-90, -70, -50, -30, -10, 10, 30, 50, 70, 90],
    include_full=True,
    full_range=(-90, 90),
    npt=1000,
    normalize=True,
    q_norm_range=(2.9308, 3.2827),
    poni_path="/path/to/NiS2_Room_T.poni",
    mask_edf_path="/path/to/Mask_NiS2_Room_T.edf",
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    out_csv_name="peak_fits.csv",
    path_root="/path/to/ESRF2023",
    analysis_subdir="analysis",
)
```

```
fig_pos, ax_pos = calibration.plot_property_vs_azimuth(
    "NiS2", 7, 300,
    _property="pv_center",
    out_csv_name="peak_fits.csv",
    ylim=(3.11, 3.135),
    path_root="/path/to/ESRF2023",
    analysis_subdir="analysis",
)

fig_fwhm, ax_fwhm = calibration.plot_property_vs_azimuth(
    "NiS2", 7, 300,
    _property="pv_fwhm",
    out_csv_name="peak_fits.csv",
    ylim=(0.01, 0.06),
    path_root="/path/to/ESRF2023",
    analysis_subdir="analysis",
)
```

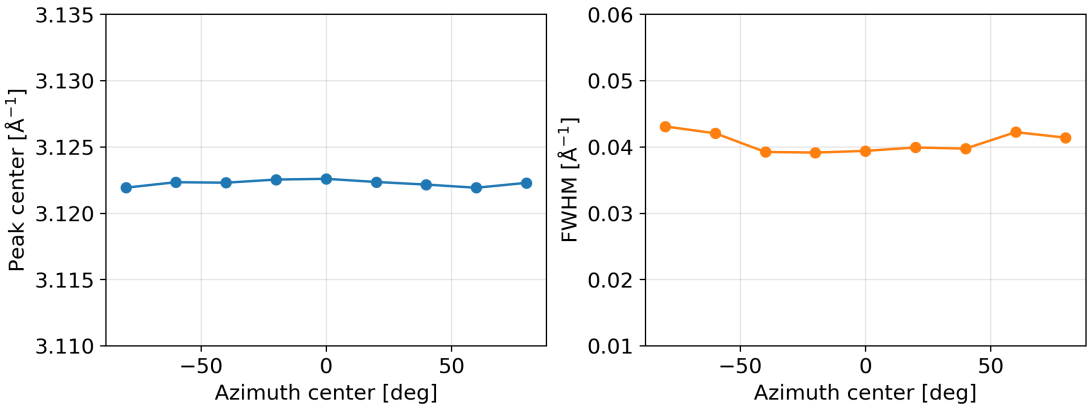


Figure 17.3: ID09 NiS2 calibration peak properties versus azimuthal sector for the 300 K dark scan. The full-range sector is excluded from the trend so the segmented azimuth windows can be compared directly.

17.5 Cake azimuthal-distribution analysis

The calibration image can also be used to quantify how intensity in a selected q band is distributed over azimuth. This is different from fitting a peak independently in each azimuthal sector. Instead, the workflow starts from a two-dimensional cake, integrates a narrow q band for every azimuthal bin, subtracts a scaled background band, and normalizes the remaining signal into an azimuthal fraction or percentage.

The user-facing function is `calibration.analyze_cake_azimuthal_distribution`. It uses the same calibration context as the detector/cake diagnostic: sample name, scan, temperature, PONI path, mask path, radial binning, azimuthal range, normalization range, azimuthal offset, and optional polarization correction. The new scientific selectors are:

- `q_value`: center of the signal q band in Å^{-1};
- `q_width`: full width of the signal q band;
- `bg_mode`: background placement, one of `left`, `right`, `average`, or `manual`;
- `bg_q_range`: explicit background interval used when `bg_mode="manual"`;
- `phi_windows`: optional $(\phi_{\text{center}}, \phi_{\text{halfwidth}})$ selectors used to summarize selected angular sectors;

and

- `mirror_mode`: `none`, `separate`, or `together`, controlling whether positive and negative angular sectors are treated independently or combined.

The signal band is interpreted as $q_{\text{value}} \pm q_{\text{width}}/2$. For automatic background modes, `left` uses the adjacent lower-q band, `right` uses the adjacent higher-q band, and `average` averages the scaled lower- and higher-q backgrounds. Manual background mode is useful when nearby peaks, fluorescence structure, or detector artifacts make the adjacent bands unsuitable. The selected background is scaled to the signal-band width before subtraction.

```
from trxrdpy.analysis import calibration

result = calibration.analyze_cake_azimuthal_distribution(
    sample_name="sample",
    scan=167246,
    temperature_K=300,
    q_value=2.2038,
    q_width=0.0757,
    bg_mode="right",
    phi_windows=[(0, 10)],
    mirror_mode="none",
    npt_rad=1000,
    npt_azim=360,
    azimuthal_range=(-90, 90),
    normalize=True,
    q_norm_range=(2.9308, 3.2827),
    use_mask=True,
    save=True,
)

profile = result["profile_df"]
summary = result["summary_df"]
```

The returned `profile_df` contains one row per azimuthal bin. The main columns are `azimuth_deg`, `full_intensity`, `background_raw`, `background_scaled`, `sample_intensity`, `fraction`, and `percent`. Metadata columns record the signal q limits, background mode, background range, scale factor, and number of q bins used. The returned `summary_df` contains one row per requested phi selector and reports the corrected intensity sum, fraction, and percent in that angular window.

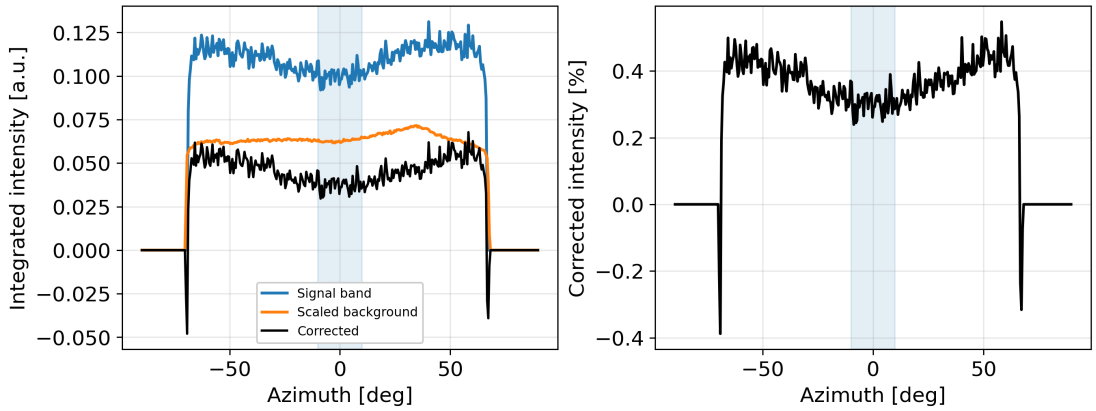


Figure 17.4: ID09 NiS2 cake azimuthal-distribution diagnostic for $q = 2.2038$ Å^{-1} with a 0.0757 Å^{-1} band and the background band placed on the high-q side. The left panel compares the signal-band integral, scaled background, and corrected response; the right panel shows the normalized corrected distribution with the selected phi window highlighted.

When called from scripts, the function can create Matplotlib figures directly. When called from the Analysis GUI, the expensive cake calculation is run with `make_plots=False`; the GUI then calls `calibration.plot_cake_azimuthal_distribution_profiles` on the main Qt thread. This split is intentional because pyFAI computation is safe to run in a worker, while Matplotlib window creation belongs on the GUI thread.

If `save=True`, the workflow writes the profile and summary tables as CSV files in the selected figures directory, together with the profile and normalized-distribution figures. Treat those CSV files as numerical outputs, not only as plotting byproducts. They define the q band, background treatment, selected phi sectors, and integrated fractions used in later interpretation.

Chapter 18

Azimuthal integration and standardized 1D patterns

18.1 Shared 2D integration workflow

The shared azimuthal-integration API is implemented in `trxrdpy.analysis._shared_2d.azimint` and reusable classes in `trxrdpy.analysis.common.azimint_utils`. It is the common route for beamlines whose preparation stage produces standardized two-dimensional images. FemtoMAX and SACLA expose this layer through their facility namespaces, and ID09 has additional facility-specific integration utilities.

The integration functions create `AzimIntegrator` objects configured with a PONI file, EDF mask, radial bin count, normalization setting, q normalization range, azimuthal offset, and optional polarization correction. Dataset classes such as `DarkDataset`, `DelayDataset`, and `FluenceDataset` describe where the input two-dimensional image is located and where `xy` cache files are written.

The shared workflow uses the same conceptual parameters for dark, delay, and fluence datasets:

- image source;
- reference or dark selection;
- azimuthal window;
- q or two-theta limits;
- normalization range;
- polarization correction; and
- cache overwrite behavior.

Existing `xy` files are reused unless overwrite is requested. This is a major performance advantage, but it also means that changes to PONI files, masks, azimuthal windows, radial bin counts, or normalization settings require attention. If the integration settings change, regenerate the corresponding `xy` cache.

```
from pathlib import Path
import numpy as np

from trxrdpy.analysis.common.paths import AnalysisPaths
from trxrdpy.analysis.MaxIV_FemtoMAX import azimint as femtomax_azimint

paths = AnalysisPaths(
    path_root=Path("/path/to/FemtoMAX/proposal"),
    analysis_subdir="analysis",
)

integrator, datasets = femtomax_azimint.integrate_delay_1d(
```

```
    sample_name="sample",
    temperature_K=300,
    excitation_wl_nm=800.0,
    fluence_mJ_cm2=1.5,
    time_window_fs=250,
    delays_fs=[-1000, 0, 1000, 5000],
    poni_path="/path/to/detector.poni",
    mask_edf_path="/path/to/mask.edf",
    azimuthal_edges=np.array([-90, -45, 0, 45, 90]),
    include_full=True,
    full_range=(-90, 90),
    npt=1000,
    normalize=True,
    q_norm_range=(2.65, 2.75),
    overwrite_xy=False,
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    paths=paths,
)
```

18.2 Delay-scan patterns

Delay-scan integration turns a set of delay-resolved two-dimensional images into a time series of one-dimensional patterns. The function `integrate_delay_1d` accepts one delay, a list of delays, or `"all"`. The helper routines discover available delay points from the standardized analysis tree and normalize the delay selection to integer femtoseconds.

Plotting functions can show absolute patterns, difference patterns relative to a reference, or both. A reference may be a selected delay, a dark pattern, or another supported reference specification depending on the facility wrapper. The reference choice is part of the scientific result. For example, subtracting a negative-delay reference and subtracting a dark pattern answer different questions.

When interpreting delay patterns, inspect the absolute patterns before the differences. A difference pattern can look clean even when the absolute pattern contains a masked-detector artifact, a normalization problem, or an unexpected baseline. The robust sequence is detector/cake diagnostic, absolute pattern inspection, reference selection, then differential plotting.

Figure 18.1 uses the ESRF ID09 NiS2 viewer dataset and is generated through the ID09 one-dimensional viewer backend. The cached `xy` files are loaded with the saved PONI geometry so the horizontal axis is displayed as q. The -5 ns reference is displayed at 0 ns after applying the saved $+5$ ns delay offset, matching the GUI convention.

```
from trxrdpy.analysis.ESRF_ID09 import azimint as id09_azimint

fig, axes = id09_azimint.plot_1D_abs_and_diffs_delay(
    sample_name="NiS2",
    raw_sample_name="NiS2",
    dataset=1,
    scan_nb=7,
    temperature_K=300,
    excitation_wl_nm=1400.0,
    fluence_mJ_cm2=10.7,
    time_window_fs=100000,
    delays_fs="all",
    ref_type="delay",
    ref_value=-5000000,
```

```
    ref_delay="-5ns",
    path_root="/path/to/ESRF2023",
    raw_subdir="RAW_DATA",
    analysis_subdir="analysis",
    poni_path="/path/to/NiS2_Room_T.poni",
    mask_edf_path="/path/to/Mask_NiS2_Room_T.edf",
    azim_window=(-90.0, 90.0),
    npt=1000,
    q_norm_range=(2.9308, 3.2827),
    normalize=True,
    compute_if_missing=False,
    xlim=(1.5, 4.5),
    fs_or_ps="ns",
    delay_offset_fs=5000000,
    polarization_factor=0.99,
    save_plots=True,
)
```

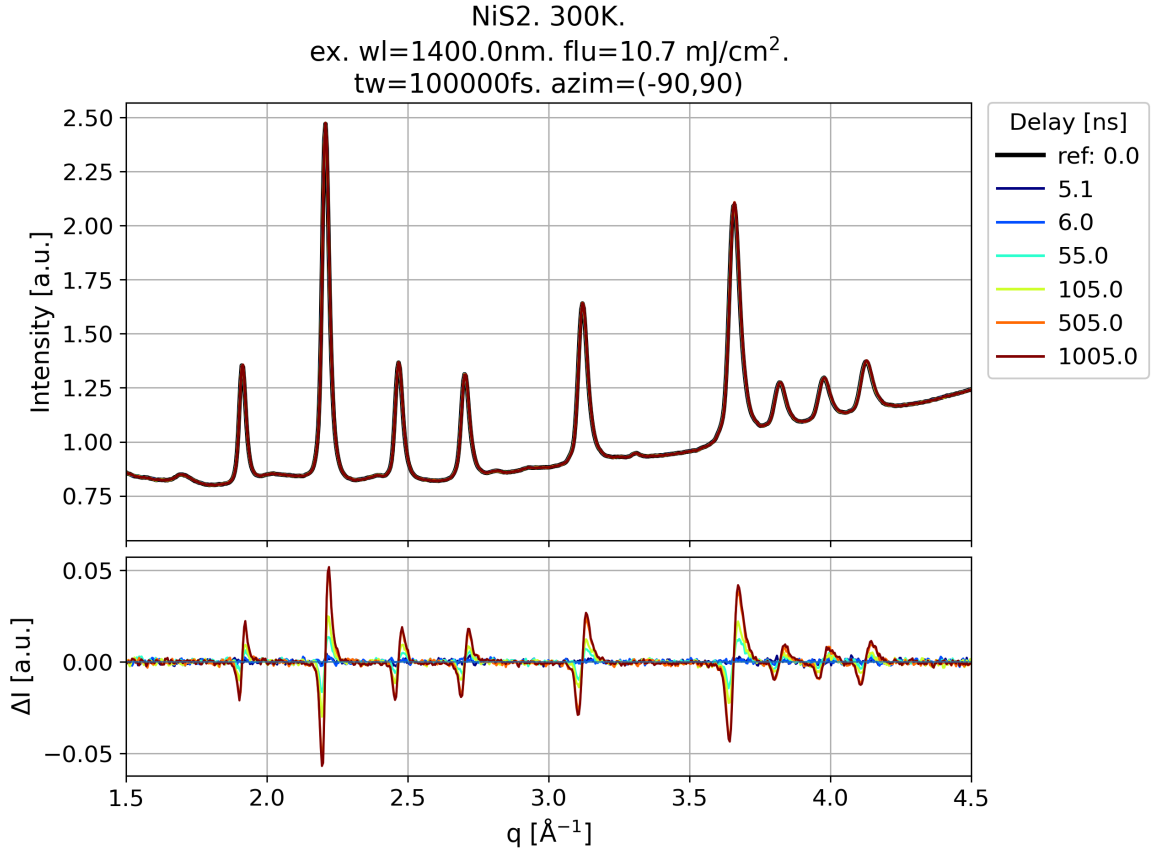


Figure 18.1: ID09 one-dimensional viewer example for NiS2 at 300 K. The top panel compares absolute full-azimuth patterns at the negative-delay reference and selected laser-on delays; the bottom panel shows differences relative to the −5 ns pattern after the saved +5 ns display offset.

18.3 Fluence-scan patterns

Fluence-scan integration compares one-dimensional patterns at different pump fluences, usually at a fixed delay. The dataset identity therefore includes sample name, temperature, excitation wavelength, fluence, time window, and delay. The package normalizes fluence labels and constructs

conventional output paths so that fluence series can be reloaded for plotting, differential analysis, or fitting.

A fluence series is not just another delay series. In a delay scan the independent variable is time at fixed excitation condition. In a fluence scan the independent variable is excitation density, and the selected delay defines the time slice being compared. Captions and filenames must make that fixed-delay condition explicit.

```
from trxrdpy.analysis.MaxIV_FemtoMAX import azimint as femtomax_azimint

fig, axes = femtomax_azimint.plot_1D_abs_and_diffs_fluence(
    sample_name="sample",
    temperature_K=300,
    excitation_wl_nm=800.0,
    delay_fs=1000,
    time_window_fs=250,
    fluences_mJ_cm2=[0.5, 1.0, 2.0, 4.0],
    ref_type="dark",
    ref_value="scan_12344",
    poni_path="/path/to/detector.poni",
    mask_edf_path="/path/to/mask.edf",
    azim_window=(-90.0, 90.0),
    npt=1000,
    normalize=True,
    q_norm_range=(2.65, 2.75),
    compute_if_missing=True,
    overwrite_xy=False,
    xlim=(1.5, 4.5),
    fluence_digits=3,
    save_plots=True,
    path_root="/path/to/FemtoMAX/proposal",
    analysis_subdir="analysis",
)
```

18.4 Quality checks before fitting

Before differential analysis or fitting, verify:

- stable q grids or correctly resampled patterns;
- acceptable signal-to-background ratio;
- consistent normalization;
- absence of masked-detector artifacts in the integrated windows; and
- correct reference selection.

These checks are faster than debugging failed fits later. A good rule is that a peak must be visually identifiable in the integrated patterns before fitting is attempted. If a peak is only visible after aggressive subtraction or normalization, the fitting model and uncertainty estimate need extra scrutiny.

Chapter 19

Differential analysis

19.1 Peak and background specifications

Differential analysis summarizes changes in selected q regions without necessarily fitting a full peak model. A peak specification defines the q interval of interest and how the background region is selected. The background can be placed before or after the peak window, or otherwise specified by the analysis utilities. This makes the result easier to interpret than a black-box integral: the user can state exactly which q interval contributed to the signal and which interval estimated the background.

The differential routines operate on differences relative to a reference. As in the integration chapter, reference choice matters. A dark reference, a negative-delay reference, and an average of multiple reference delays can produce different physical interpretations.

19.2 Delay differential traces

The main delay differential entry point, `differential_analysis.plot_differential_integrals`, computes signed ΔI and absolute $|\Delta I|$ integrals for a selected peak region and matched background. Error bars, when requested, are derived from the background response at the same delay. This gives a direct visual check of whether the peak-region response is larger than the local background variation.

Delay traces can be plotted in femtoseconds or converted to other time units. A display offset can be applied when the experimental time zero is refined after reduction. The output is both a table and a figure, so the numerical trace can be reused for further analysis.

Figure 19.1 uses the ID09 NiS2 delay dataset shown in the one-dimensional viewer example. The selected peak is the (202) region, integrated over $q = (3.0182, 3.2232)$ Å^{-1}, with the matched background placed after the peak window. The reference is the -5 ns pattern, and the displayed delay axis includes the saved $+5$ ns offset.

```
from trxrdpy.analysis import differential_analysis

peak_specs = {
    "202": {
        "q_range": (3.0182, 3.2232),
        "bg_mode": "after",
    }
}

df, fig = differential_analysis.plot_differential_integrals(
    sample_name="NiS2",
```

```
    temperature_K=300,
    excitation_wl_nm=1400.0,
    fluence_mJ_cm2=10.7,
    time_window_fs=100000,
    delays_fs="all",
    ref_type="delay",
    ref_value=-5000000,
    poni_path="/path/to/NiS2_Room_T.poni",
    mask_edf_path="/path/to/Mask_NiS2_Room_T.edf",
    azim_window=(-90.0, 90.0),
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    peak="202",
    peak_specs=peak_specs,
    npt=1000,
    normalize_xy=True,
    q_norm_range=(2.9308, 3.2827),
    compute_if_missing=False,
    unit="ns",
    delay_offset_fs=5000000,
    show_errorbars=True,
    save=True,
    path_root="/path/to/ESRF2023",
    analysis_subdir="analysis",
)
```

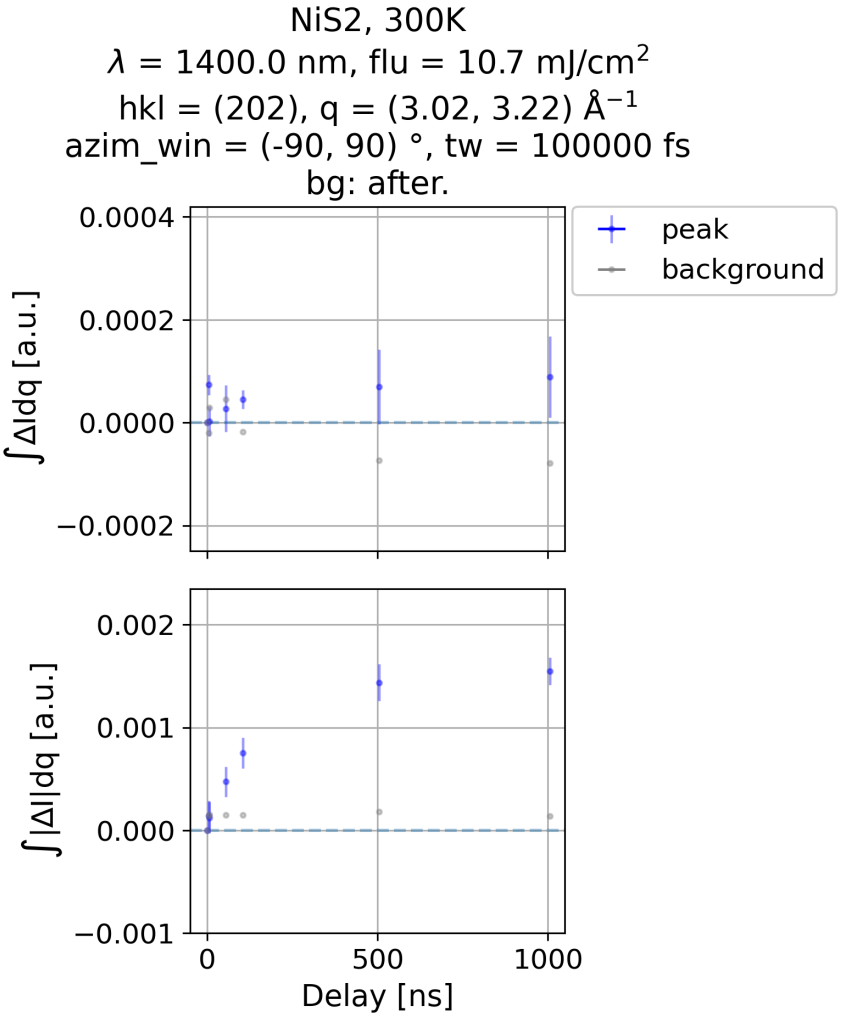


Figure 19.1: ID09 NiS2 differential-integral example for the (202) peak region. The upper panel shows signed $\int \Delta I dq$, and the lower panel shows $\int |\Delta I| dq$; the gray background trace provides the error-bar estimate for the blue peak-region response.

The same delay-domain differential trace can be transformed with `differential_analysis.plot_differential_fft`.

```
df, (fft_peak, fft_background), fig = differential_analysis.plot_differential_fft(
    sample_name="NiS2",
    temperature_K=300,
    excitation_wl_nm=1400.0,
    fluence_mJ_cm2=10.7,
    time_window_fs=100000,
    delays_fs="all",
    ref_type="delay",
    ref_value=-5000000,
    poni_path="/path/to/NiS2_Room_T.poni",
    mask_edf_path="/path/to/Mask_NiS2_Room_T.edf",
    azim_window=(-90.0, 90.0),
    peak="202",
    peak_specs=peak_specs,
    region="peak",
    kind="diff",
    time_window_select_ps=(0, 1000),
    poly_order=1,
    freq_unit="cm^-1",
    compute_if_missing=False,
    save=True,
    path_root="/path/to/ESRF2023",
    analysis_subdir="analysis",
)
```

19.3 Fluence differential traces

Fluence differential traces summarize the same kind of integrated intensity change, but as a function of pump fluence. They are useful for identifying thresholds, saturation, or non-linear response. As with fluence fitting, the fixed delay must be stated. If the fixed delay changes, the fluence trace is a different experiment.

```
peak_specs = {"116": {"q_range": (3.55, 3.75), "bg_mode": "after"}}

df, fig = differential_analysis.plot_differential_integrals_fluence(
    sample_name="DET58_2",
    temperature_K=80,
    excitation_wl_nm=1400.0,
    delay_fs=500000,
    time_window_fs=100000,
    fluences_mJ_cm2=[1.3, 2.29, 5.49, 13.9],
    ref_type="dark",
    ref_value="scan_7",
    poni_path="/path/to/DET58_80K_2_0001_0035.poni",
    mask_edf_path="/path/to/DET58_80K_mask.edf",
    azim_window=(-90.0, 90.0),
    azim_offset_deg=90.0,
    polarization_factor=0.99,
    peak="116",
    peak_specs=peak_specs,
    npt=1000,
    normalize_xy=True,
    q_norm_range=(2.65, 2.75),
    compute_if_missing=True,
```

```
    show_errorbars=True,
    save=True,
    path_root="/path/to/ESRF2026/March",
    analysis_subdir="analysis",
)
```

19.4 Multi-experiment comparison

Multi-experiment differential analysis compares traces from several experiments or conditions. Each experiment can carry its own path configuration, or the function can receive a shared path configuration. The key requirement is explicit labeling: sample name, temperature, excitation wavelength, fluence, time window, reference choice, and azimuthal window must be visible either in the plot legend or in the saved metadata.

Comparisons can be made on an absolute scale or after normalization. Absolute-scale comparisons preserve amplitude information but require consistent integration and normalization. Shape-normalized comparisons are useful for timing or frequency comparisons, but they do not support claims about absolute intensity differences.

```
base_experiment = {
    "sample_name": "DET58_2",
    "temperature_K": 80,
    "excitation_wl_nm": 1400.0,
    "time_window_fs": 100000,
    "ref_type": "dark",
    "ref_value": "scan_7",
}

experiments = [
    {**base_experiment, "label": "1.3 mJ/cm^2", "fluence_mJ_cm2": 1.3},
    {**base_experiment, "label": "13.9 mJ/cm^2", "fluence_mJ_cm2": 13.9},
]

result = differential_analysis.plot_differential_integrals_multi(
    experiments=experiments,
    delays_fs="all",
    poni_path="/path/to/DET58_80K_2_0001_0035.poni",
    mask_edf_path="/path/to/DET58_80K_mask.edf",
    azim_window=(-90.0, 90.0),
    azim_offset_deg=90.0,
    polarization_factor=0.99,
    peak="116",
    peak_specs=peak_specs,
    npt=1000,
    normalize_xy=True,
    q_norm_range=(2.65, 2.75),
    compute_if_missing=True,
    unit="ps",
    show_errorbars=True,
    save=True,
    path_root="/path/to/ESRF2026/March",
    analysis_subdir="analysis",
)
```

Chapter 20

Peak fitting

20.1 Fitting philosophy

Peak fitting is a model-based compression of the one-dimensional pattern. It is never the first diagnostic. First verify the detector/cake geometry, then inspect absolute and differential patterns, and only then fit peaks. The fit is useful when a peak can be described by a stable model over a defined q interval and when the fitted parameters have a physical interpretation.

The fitting utilities use peak specifications: each named peak defines a q range and optional model settings. Delay and fluence fitting routines then apply those specifications to every selected delay or fluence point and every selected azimuthal group. The default model is based on a pseudo-Voigt peak plus background handling implemented through lmfit models [16]. The user controls the q window, pseudo-Voigt fraction, fitting method, azimuthal treatment, reference behavior, and whether missing `xy` files may be computed on demand.

20.2 Delay peak fitting

The main delay fitting entry point is `fitting.run_delay_peak_fitting`. It locates the selected delay patterns, optionally integrates missing `xy` files from two-dimensional images, fits each requested peak, and writes a CSV table. The output can include reference rows, and reference values can be combined or kept separate depending on the analysis question.

Azimuthal handling is explicit. With `phi_mode="separate_phi"`, sectors are fit independently. With `phi_mode="phi_avg"`, symmetry-related or selected sectors can be reduced before fitting, using `phi_reduce="sum"` or `"mean"`. These choices affect amplitudes and areas, so they must be included in filenames and captions. The code tags output CSV names with the azimuthal mode to reduce accidental reuse of incompatible results.

Diagnostic fit figures are optional because a full delay series can produce many plots. During method development, saving or showing fit overlays is useful. Once the model is validated, the CSV table and evolution plots are usually the main outputs.

The ID09 fitting examples in this section use the 2026 DET58_2 dataset at 80 K and 1400 nm excitation with a 100 ps time window. The delay comparison follows the saved multi-delay fitting state and uses the available local fluence folders from 0.9 to 13.9 mJ cm^{-2}. The position and FWHM evolutions are plotted separately using `fitting.plot_time_evolution_multi`; the display then uses a logarithmic delay axis to match the GUI view.

```
from trxrdpy.analysis import fitting

overlay = fitting.plot_fit_overlay_from_csv(
    sample_name="DET58_2",
```

```
    temperature_K=80,
    excitation_wl_nm=1400.0,
    fluence_mJ_cm2=13.9,
    time_window_fs=100000,
    peak="110",
    delay_fs=500000,
    group="Full",
    phi_mode="phi_avg",
    phi_reduce="sum",
    csv_path="/path/to/peak_fits_delay_phiavg_sum.csv",
    ensure_csv=False,
    show=False,
    poni_path="/path/to/DET58_80K_2_0001_0035.poni",
    mask_edf_path="/path/to/DET58_80K_mask.edf",
    azim_offset_deg=90.0,
    compute_if_missing=False,
    path_root="/path/to/ESRF2026/March",
    analysis_subdir="analysis",
)
```

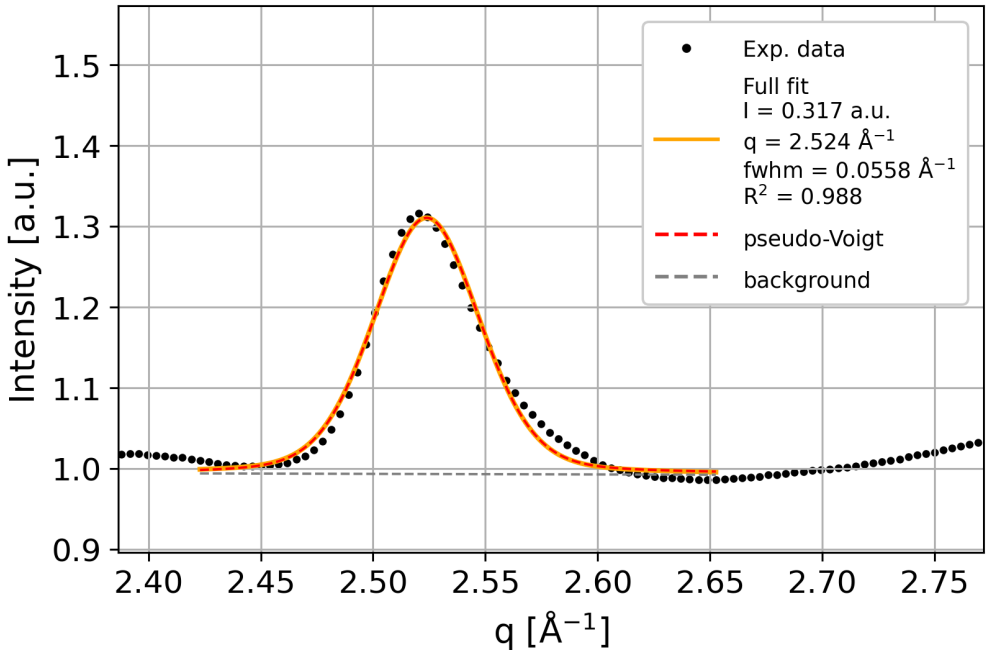


Figure 20.1: ID09 DET58_2 delay-fit overlay reconstructed from the saved fit CSV for the (110) peak at 500 ps and 13.9 mJ cm^{-2}. The plot shows the experimental points, pseudo-Voigt component, background, and total fit for the full azimuthal group after `phi_avg` summation.

```
experiments = [
    {
        "label": "V2O3, 80, 1400, 1.3, 100000",
        "sample_name": "DET58_2",
        "temperature_K": 80,
        "excitation_wl_nm": 1400.0,
        "fluence_mJ_cm2": 1.3,
        "time_window_fs": 100000,
        "phi_mode": "phi_avg",
        "phi_reduce": "sum",
        "ref_type": "dark",
        "csv_path": "/path/to/fluence_1p3mJ/peak_fits_delay_phiavg_sum.csv",
```

```
    },
    {
        "label": "V2O3, 80, 1400, 13.9, 100000",
        "sample_name": "DET58_2",
        "temperature_K": 80,
        "excitation_wl_nm": 1400.0,
        "fluence_mJ_cm2": 13.9,
        "time_window_fs": 100000,
        "phi_mode": "phi_avg",
        "phi_reduce": "sum",
        "ref_type": "dark",
        "csv_path": "/path/to/fluence_13p9mJ/peak_fits_delay_phiavg_sum.csv",
    },
]

position_result = fitting.plot_time_evolution_multi(
    experiments=experiments,
    peak="116",
    _property="hkl_pos",
    unit="ps",
    phi_mode="phi_avg",
    phi_reduce="sum",
    phi_window="Full",
    as_lines=True,
    show_baseline_sigma=True,
    baseline_mode="errorbar",
    cmap="jet",
    show=False,
)
position_result["ax"].set_xscale("log")

fwhm_result = fitting.plot_time_evolution_multi(
    experiments=experiments,
    peak="116",
    _property="hkl_fwhm",
    unit="ps",
    phi_mode="phi_avg",
    phi_reduce="sum",
    phi_window="Full",
    as_lines=True,
    show_baseline_sigma=True,
    baseline_mode="errorbar",
    cmap="jet",
    show=False,
)
fwhm_result["ax"].set_xscale("log")
```

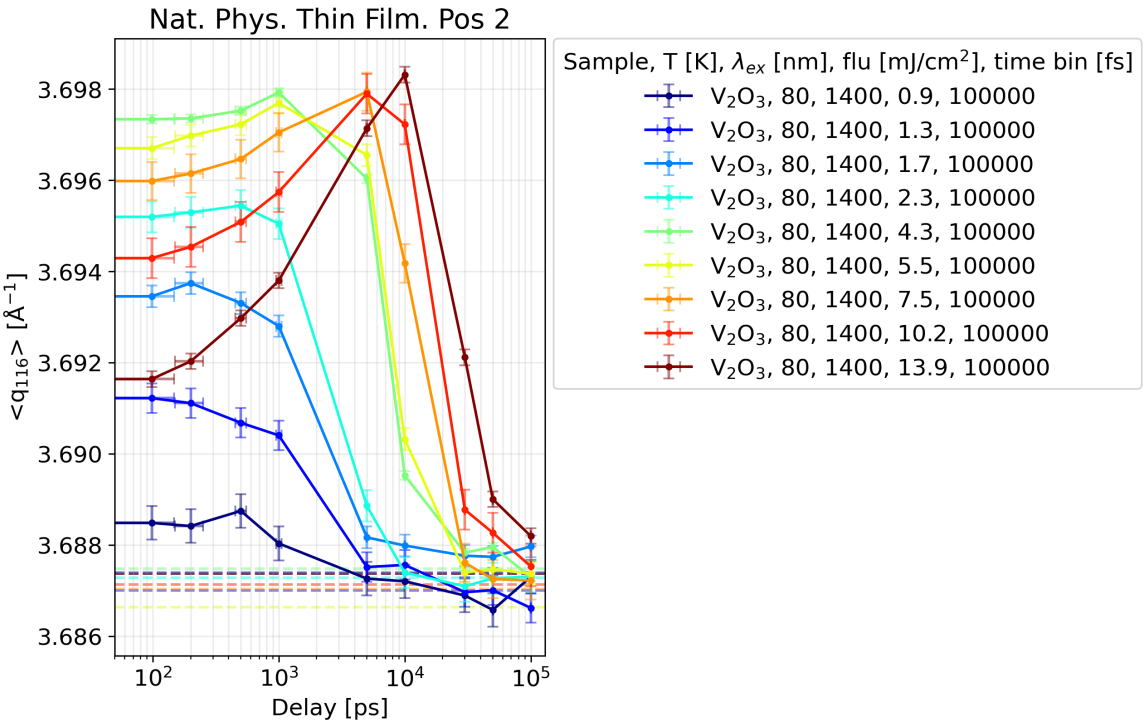


Figure 20.2: ID09 DET58_2 fitted (116) position versus delay for the local delay-scan fluence series, generated with the existing multi-experiment fitting plotting facility. The delay axis is displayed logarithmically, matching the GUI view.

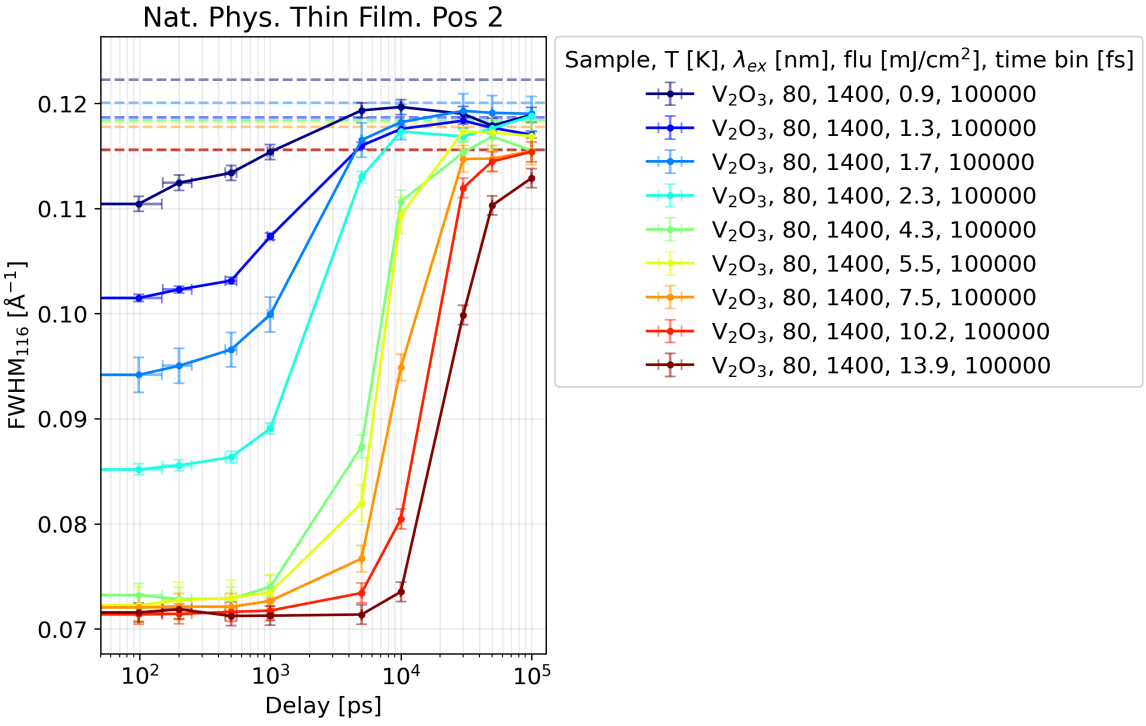


Figure 20.3: ID09 DET58_2 fitted (116) FWHM versus delay for the local delay-scan fluence series, generated with the existing multi-experiment fitting plotting facility. The delay axis is displayed logarithmically, matching the GUI view.

20.3 Fluence peak fitting

Fluence peak fitting follows the same model logic as delay fitting, but the independent variable is fluence rather than time. The fluence fitter resolves fluence-specific `xy` files, writes fluence-specific CSV tables, and provides evolution plots as a function of excitation fluence.

The main interpretive risk is confusing fluence dependence with time dependence. Captions and filenames must always state the fixed delay used for the fluence series. If multiple delays are analyzed as separate fluence series, give each delay its own output path or clearly tagged CSV file.

The fluence example uses the shared 2026 DET58_2 fixed-delay folders at 500 ps, 5 ns, and 10 ns. Each curve is loaded from its corresponding `peak_fits_fluence_phiavg_sum.csv` file, and the plotted quantity follows the GUI setting for the shared state: (116) FWHM in the full `phi_avg` group.

```
from trxrdpy.analysis import fitting

experiments = [
    {
        "label": "500 ps",
        "sample_name": "DET58_2",
        "temperature_K": 80,
        "excitation_wl_nm": 1400.0,
        "delay_fs": 500000,
        "time_window_fs": 100000,
        "phi_mode": "phi_avg",
        "ref_type": "dark",
        "csv_path": "/path/to/delay_500000fs/peak_fits_fluence_phiavg_sum.csv",
    },
    {
        "label": "10 ns",
        "sample_name": "DET58_2",
        "temperature_K": 80,
        "excitation_wl_nm": 1400.0,
        "delay_fs": 10000000,
        "time_window_fs": 100000,
        "phi_mode": "phi_avg",
        "ref_type": "dark",
        "csv_path": "/path/to/delay_10000000fs/peak_fits_fluence_phiavg_sum.csv",
    },
]

fig, ax, saved_path = fitting.plot_fluence_evolution_multi(
    experiments=experiments,
    peak="116",
    prop="hkl_fwhm",
    group_by="phi_label",
    group="Full",
    fluence_unit="mJ/cm$^2$",
    only_success=True,
    include_reference=False,
    legend_title="Delay",
    as_lines=True,
    show=False,
    save=True,
)
```

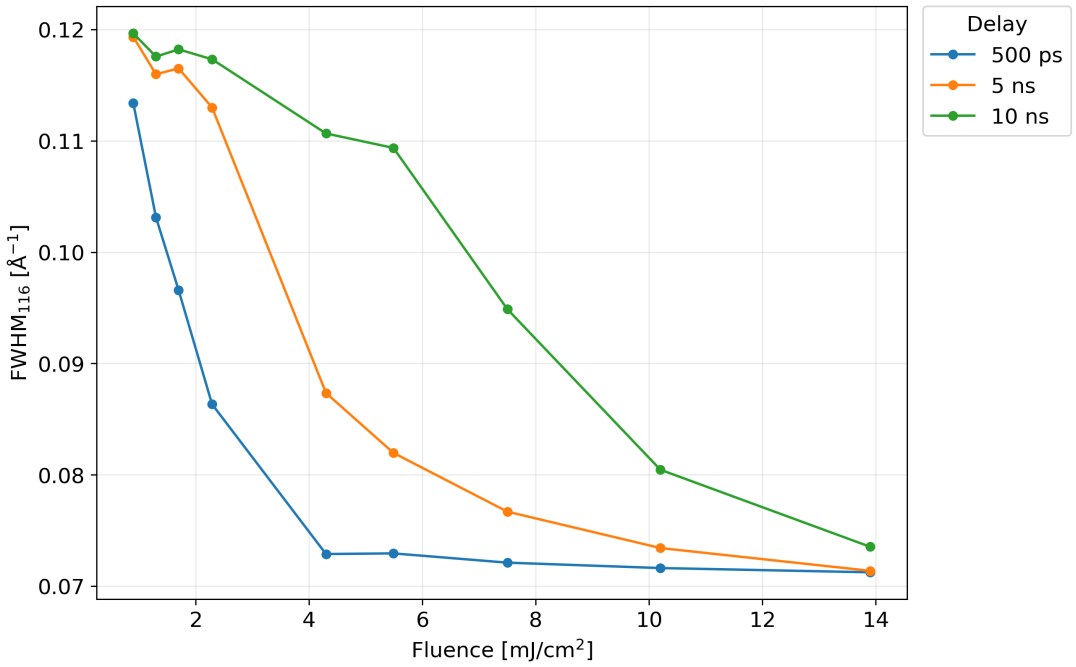


Figure 20.4: ID09 DET58_2 fitted (116) FWHM versus fluence for three fixed-delay fluence scans. The fixed delay is encoded by the curve label, so the figure should not be interpreted as a time trace.

20.4 Fit outputs and diagnostics

Fit CSV files are the numerical record of the fitting step. They normally include experiment identity, independent-variable value, peak name, azimuthal group, best-fit parameters, derived width or area values, and quality metrics such as R^2 where available. Failed fits remain visible in the table rather than being silently dropped, because missing points can bias time or fluence trends.

The overlay functions can reconstruct diagnostic plots from saved CSV files. This is useful when a user wants to inspect one suspicious point without rerunning the entire fit series. The correct workflow is to keep the CSV, the peak specification, and the integration settings together.

Chapter 21

Facility-specific worked analysis examples

The shared chapters above define the common analysis pipeline: detector diagnostics, azimuthal integration, differential analysis, and peak fitting. The examples in this chapter apply those shared tools to facility-specific datasets after the facility preparation step has already produced corrected two-dimensional images or cached one-dimensional `xy` files.

21.1 Max IV FemtoMAX

21.1.1 FemtoMAX single-delay-scan example

This example follows a FemtoMAX DET58 delay scan at 300 K, 1500 nm, and 52 mJ cm^{-2}. The reduction step has already produced averaged two-dimensional images and cached one-dimensional `xy` files for a 1000 fs time window. The reference used for the delay scan is not a dark scan; it is the -98972 fs delay point. For display, a $+4000$ fs offset is applied, so the reference appears at -94.97 ps.

21.1.1.1 Detector and 2D cake diagnostic from the FemtoMAX dark scan

The calibration diagnostic starts from dark scan 180951. The detector image is plotted with the detector y axis inverted, the detector color scale limited to $(0, 0.05)$, and the cake color scale limited to $(0, 2)$. The cake is normalized over $q = (1.9, 2.0)$ $\mathrm{Å}^{-1}$.

```
from pathlib import Path

from trxrdpy.analysis import calibration

root = Path("/path/to/FemtoMAX2026")
poni_path = root / "calibration/DET58_180951.poni"
mask_path = root / "calibration/DET58_180951_mask.edf"

fig, axes, data = calibration.plot_detector_and_cake(
    sample_name="DET58",
    scan=180951,
    temperature_K=300,
    npt_rad=1000,
    npt_azim=360,
    azimuthal_range=(-90.0, 90.0),
    normalize=True,
    q_norm_range=(1.9, 2.0),
    use_mask=True,
    poni_path=str(poni_path),
```

```
    mask_edf_path=str(mask_path),
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    detector_clim=(0.0, 0.05),
    cake_clim=(0.0, 2.0),
    invert_detector_y=True,
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

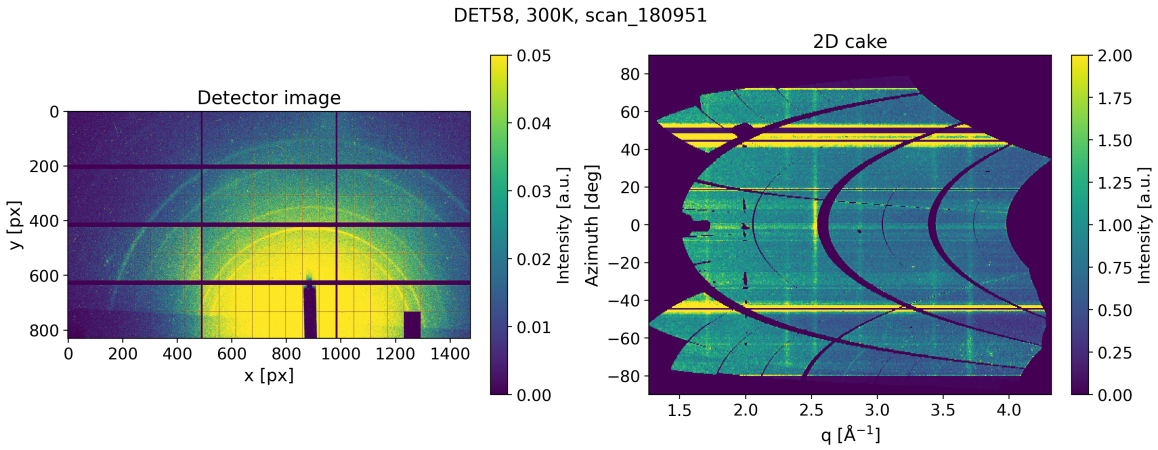


Figure 21.1: FemtoMAX DET58 dark-scan detector image and 2D cake diagnostic for scan 180951. The detector panel uses the inverted y-axis convention used in the GUI state, and the cake panel uses the normalized q-azimuth representation used for later integration checks.

21.1.1.2 One-dimensional viewer for the FemtoMAX delay scan

The one-dimensional viewer compares all cached delay patterns against the selected delay reference. The absolute-pattern and differential-pattern y limits are passed directly to the plotting function, so the figure can be reproduced from the GUI without manual Matplotlib edits after plotting.

```
from pathlib import Path

from trxrdpy.analysis.MaxIV_FemtoMAX import azimint as femtomax_azimint

root = Path("/path/to/FemtoMAX2026")
poni_path = root / "calibration/DET58_180951.poni"
mask_path = root / "calibration/DET58_180951_mask.edf"

q_ref, I_ref, fig, axes = femtomax_azimint.plot_1D_abs_and_diffs_delay(
    sample_name="DET58",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    fluence_mJ_cm2=52.0,
    time_window_fs=1000,
    delays_fs="all",
    ref_type="delay",
    ref_value=-98972,
    poni_path=str(poni_path),
    mask_edf_path=str(mask_path),
    azim_window=(-90.0, 90.0),
```

```
    npt=1000,
    normalize=True,
    q_norm_range=(1.9, 2.0),
    compute_if_missing=False,
    overwrite_xy=False,
    xlim=(1.5, 4.0),
    ylim_top=(0.3285, 1.5136),
    ylim_diff=(-0.55, 0.55),
    fs_or_ps="ps",
    digits=2,
    delay_offset_fs=4000.0,
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    save_plots=True,
    from_2D_imgs=False,
    path_root=root,
    analysis_subdir="analysis",
)
```

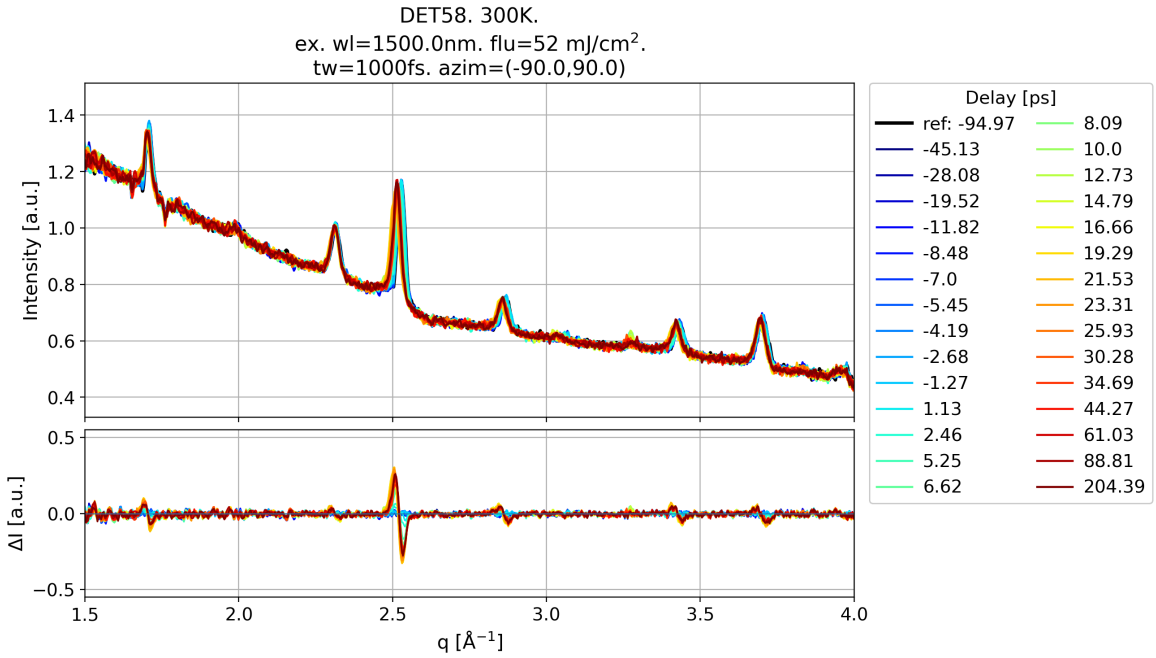


Figure 21.2: FemtoMAX DET58 one-dimensional viewer example for the 300 K, 1500 nm delay scan. The reference is the −98972 fs delay pattern, displayed as −94.97 ps after the +4000 fs offset.

21.1.1.3 Differential-integral analysis for the FemtoMAX (110) peak

The differential-integral plot uses the (110) peak over $q = (2.4584, 2.5733)$ Å^{-1}, with the matched background window placed after the peak. The displayed delay axis and both y limits are provided as plotting arguments.

```
from pathlib import Path

from trxrdpy.analysis import differential_analysis

root = Path("/path/to/FemtoMAX2026")
```

```
poni_path = root / "calibration/DET58_180951.poni"
mask_path = root / "calibration/DET58_180951_mask.edf"

peak_specs = {
    "110": {"q_range": (2.4584, 2.5733), "bg_side": "right"},
}

df, fig = differential_analysis.plot_differential_integrals(
    sample_name="DET58",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    fluence_mJ_cm2=52.0,
    time_window_fs=1000,
    delays_fs="all",
    ref_type="delay",
    ref_value=-98972,
    poni_path=str(poni_path),
    mask_edf_path=str(mask_path),
    azim_window=(-90.0, 90.0),
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    peak="110",
    peak_specs=peak_specs,
    npt=1000,
    normalize_xy=True,
    q_norm_range=(1.9, 2.0),
    compute_if_missing=False,
    overwrite_xy=False,
    unit="ps",
    delay_offset_fs=4000.0,
    show_errorbars=True,
    errorbar_scale=1.0,
    xlim=(-15.0, 65.0),
    ylim_signed=(-0.008, 0.008),
    ylim_abs=(-0.0027, 0.018),
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

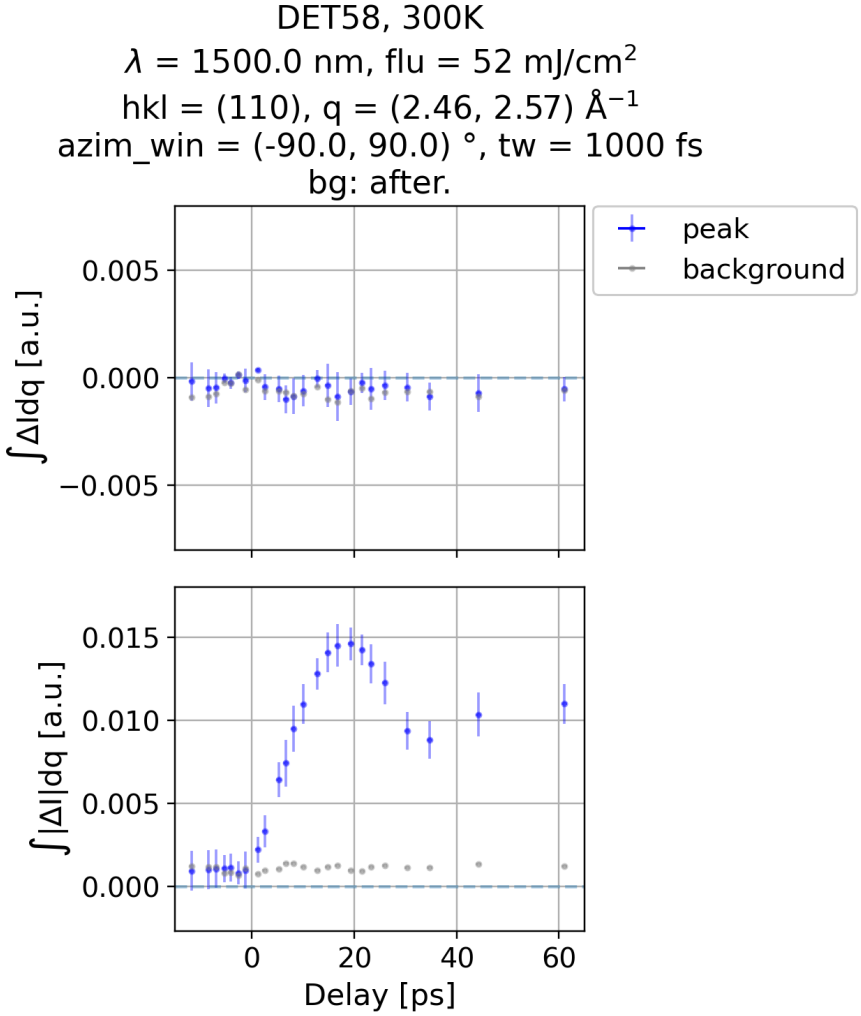


Figure 21.3: FemtoMAX DET58 differential-integral example for the (110) peak region. The upper panel shows signed $\int \Delta I dq$, and the lower panel shows $\int |\Delta I| dq$ over the displayed -15 to 65 ps window.

21.1.1.4 Fitting analysis for different FemtoMAX azimuthal sectors

The fitting example reads the existing `peak_fits_delay_phiavg_sum.csv` table and plots the fitted (110) peak position for the full azimuthal range and the $|\Phi| = 60°, 30°, 0°$ sector groups. The dashed horizontal lines are the reference values from the selected delay reference.

```
from pathlib import Path

from trxrdpy.analysis import fitting

root = Path("/path/to/FemtoMAX2026")
csv_path = (
    root / "analysis/DET58/temperature_300K/excitation_wl_1500nm"
    / "delay/fluence_52p0mJ/time_window_1000fs"
    / "fitting/peak_fits_delay_phiavg_sum.csv"
)

out = fitting.plot_time_evolution(
    sample_name="DET58",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    fluence_mJ_cm2=52.0,
    time_window_fs=1000,
    peak="110",
    _property="hkl_pos",
    unit="ps",
    group_by="azim_range_str",
    groups=["Full", 60, 30, 0],
    only_success=True,
    include_reference=True,
    as_lines=True,
    delay_offset_fs=4000.0,
    show_baseline_sigma=True,
    baseline_sigma=1.0,
    baseline_alpha=1.0,
    baseline_mode="errorbar",
    xlim=(-15.0, 65.0),
    ylim=(2.50, 2.535),
    csv_path=str(csv_path),
    phi_mode="phi_avg",
    phi_reduce="sum",
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

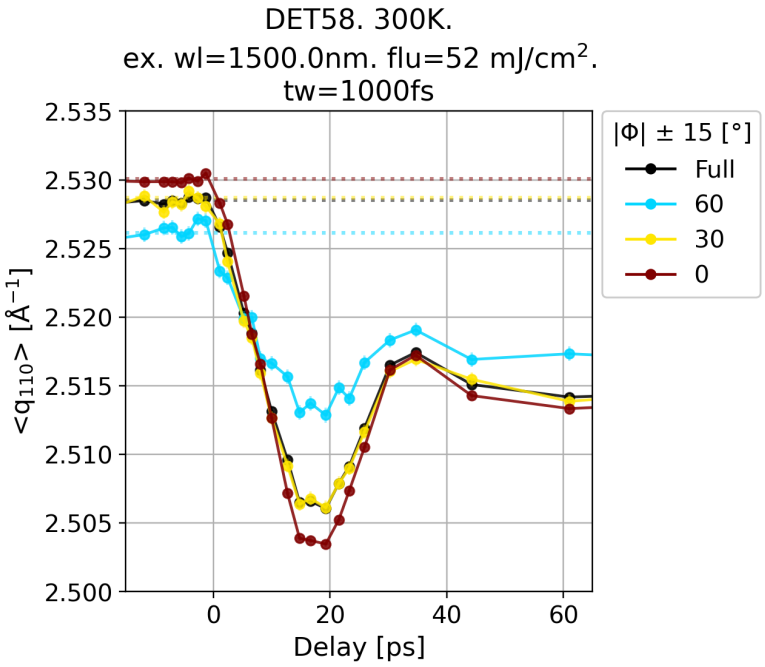


Figure 21.4: FemtoMAX DET58 fitted (110) peak position versus delay for different symmetric azimuthal sectors, generated from the existing `peak_fits_delay_phiavg_sum.csv` file.

21.2 SPring-8 / SACLA

21.2.1 SACLA delay-scan examples from cached one-dimensional data

Once the SACLA preparation step has produced corrected two-dimensional images or cached one-dimensional `xy` files, the downstream analysis uses the shared two-dimensional integration backend exposed through `trxrdpy.analysis.Spring8_SACLA.azimint`. The examples below use a DET70 delay scan at 300 K and 1500 nm. The local folders are stored as 25 and 10 mJ cm^{-2}, while the GUI display applies a fluence scale of 0.4. Therefore the plotted labels appear as 10 and 4 mJ cm^{-2}. This scale is a display convention only; the data lookup still uses the folder fluence.

21.2.1.1 One-dimensional viewer for a SACLA delay scan

The one-dimensional viewer compares cached delay patterns against a dark reference and plots both the absolute patterns and their differences. In this example the reference is dark scan 1466609, the integration uses the full $(-90°, 90°)$ azimuthal range, and the displayed delay axis includes a $+9.6$ ps offset.

```
from pathlib import Path

from trxrdpy.analysis.Spring8_SACLA import azimint as sacla_azimint

root = Path("/path/to/SACLA2024")
poni_path = root / "calibration/DET70_1466610.poni"
mask_path = root / "calibration/DET70_300K_mask.edf"

q_ref, I_ref, fig, axes = sacla_azimint.plot_1D_abs_and_diffs_delay(
    sample_name="DET70",
    temperature_K=300,
    excitation_wl_nm=1500.0,
```

```
    fluence_mJ_cm2=25.0,
    time_window_fs=250,
    delays_fs="all",
    ref_type="dark",
    ref_value=1466609,
    poni_path=str(poni_path),
    mask_edf_path=str(mask_path),
    azim_window=(-90.0, 90.0),
    npt=1000,
    normalize=True,
    q_norm_range=(2.65, 2.75),
    compute_if_missing=True,
    overwrite_xy=False,
    xlim=(1.5, 4.0),
    fs_or_ps="ps",
    digits=2,
    delay_offset_fs=9600.0,
    fluence_scale=0.4,
    max_curves=30,
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    save_plots=True,
    from_2D_imgs=False,
    path_root=root,
    analysis_subdir="analysis",
)
```

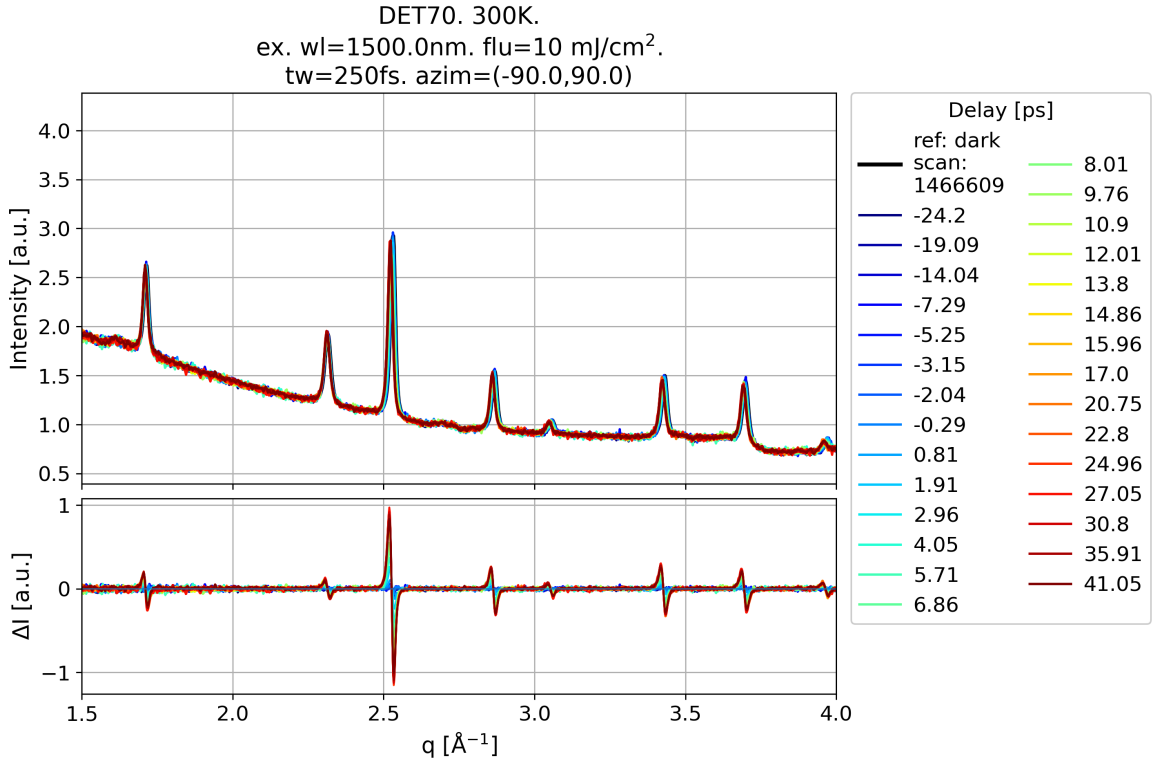


Figure 21.5: SACLA DET70 one-dimensional viewer example for the 300 K, 1500 nm delay scan. The top panel compares absolute full-azimuth patterns against dark scan 1466609, and the bottom panel shows the differences relative to the dark reference after the +9.6 ps display offset.

21.2.1.2 Differential-integral analysis for the SACLA (110) peak

The differential-analysis function integrates the signed and absolute differential response in the selected peak window. Here we use the (110) peak, $q = (2.45, 2.60)$ Å^{-1}, with the matched background window placed after the peak.

```
from pathlib import Path

from trxrdpy.analysis import differential_analysis

root = Path("/path/to/SACLA2024")
poni_path = root / "calibration/DET70_1466610.poni"
mask_path = root / "calibration/DET70_300K_mask.edf"

peak_specs = {
    "110": {"q_range": (2.45, 2.60), "bg_side": "right"},
}

df, fig = differential_analysis.plot_differential_integrals(
    sample_name="DET70",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    fluence_mJ_cm2=25.0,
    time_window_fs=250,
    delays_fs="all",
    ref_type="dark",
    ref_value=1466609,
    poni_path=str(poni_path),
    mask_edf_path=str(mask_path),
    azim_window=(-90.0, 90.0),
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    peak="110",
    peak_specs=peak_specs,
    npt=1000,
    normalize_xy=True,
    q_norm_range=(2.65, 2.75),
    compute_if_missing=True,
    overwrite_xy=False,
    unit="ps",
    delay_offset_fs=9600.0,
    fluence_scale=0.4,
    show_errorbars=True,
    errorbar_scale=1.0,
    plot_abs_and_diffs=True,
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

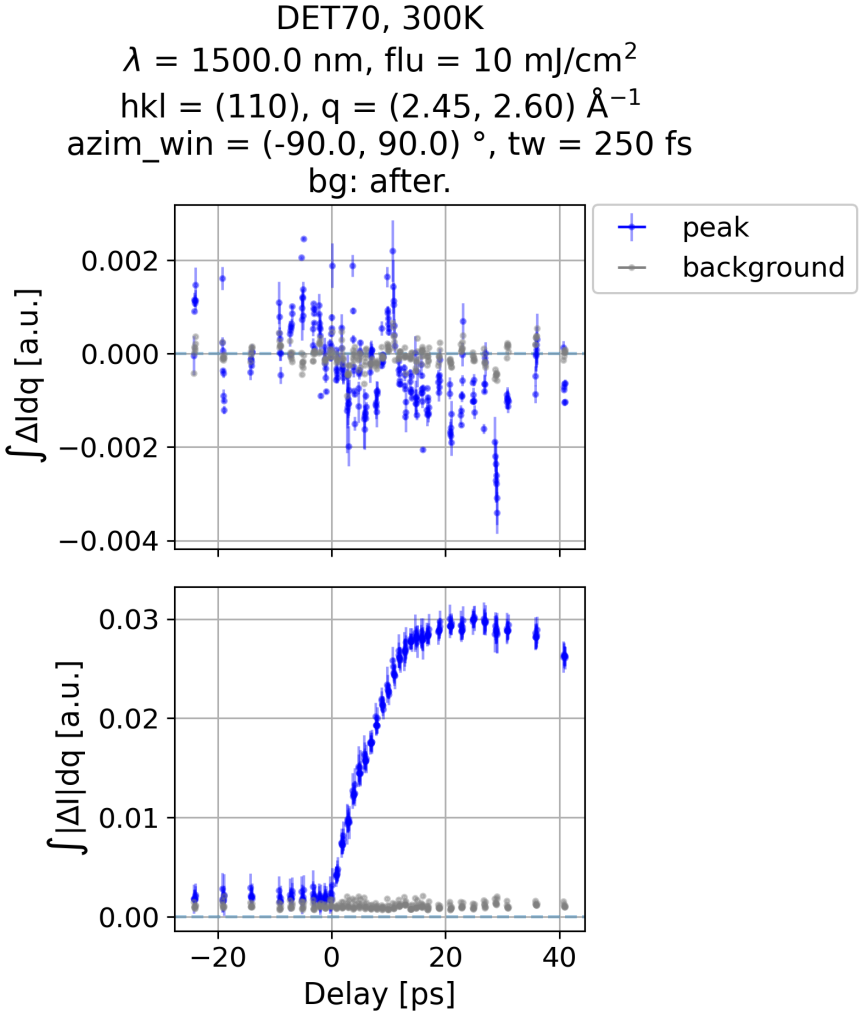


Figure 21.6: SACLA DET70 differential-integral example for the (110) peak region. The upper panel shows signed $\int \Delta I dq$, and the lower panel shows $\int |\Delta I| dq$; the gray background trace provides the error-bar estimate for the blue peak-region response.

21.2.1.3 Fitting analysis for different azimuthal sectors

For the same delay scan, the fitting analysis can plot a fitted peak property for several symmetric azimuthal sectors. The example below reads the existing `phi_avg` fitting CSV and plots the (110) peak position for the full sector and the $|\Phi| = 60°, 30°, 0°$ sector groups.

```
from pathlib import Path

from trxrdpy.analysis import fitting

root = Path("/path/to/SACLA2024")
csv_path = (
    root / "analysis/DET70/temperature_300K/excitation_wl_1500nm"
    / "delay/fluence_25p0mJ/time_window_250fs"
    / "fitting/peak_fits_delay_phiavg_sum.csv"
)

out = fitting.plot_time_evolution(
    sample_name="DET70",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    fluence_mJ_cm2=25.0,
    time_window_fs=250,
    peak="110",
    _property="hkl_pos",
    out_csv_name="peak_fits_delay.csv",
    unit="ps",
    group_by="azim_range_str",
    groups=["Full", 60, 30, 0],
    only_success=True,
    include_reference=True,
    as_lines=False,
    delay_offset_fs=9600.0,
    fluence_scale=0.4,
    show_baseline_sigma=True,
    baseline_sigma=1.0,
    baseline_alpha=0.18,
    baseline_mode="errorbar",
    csv_path=str(csv_path),
    phi_mode="phi_avg",
    phi_reduce="sum",
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

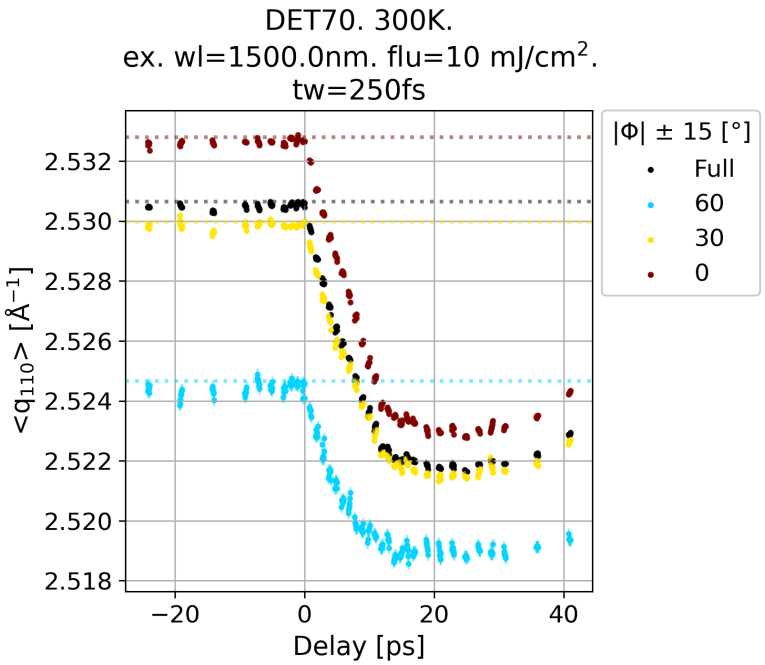


Figure 21.7: SACLA DET70 fitted (110) peak position versus delay for different symmetric azimuthal sectors. The figure is generated by the existing fitting time-evolution plotting facility using the cached `peak_fits_delay_phiavg_sum.csv` file.

21.2.1.4 Merged multi-delay fitting comparison

The multi-experiment fitting plotter can compare a merged time-resolution series with another fluence condition. In the example below, the 25 mJ cm^{-2} folder is displayed as 10 mJ cm^{-2} and combines 40 fs and 250 fs time-window CSVs. The 10 mJ cm^{-2} folder is displayed as 4 mJ cm^{-2}.

```
from pathlib import Path

from trxrdpy.analysis import fitting

root = Path("/path/to/SACLA2024")
delay_root = (
    root / "analysis/DET70/temperature_300K/excitation_wl_1500nm/delay"
)

experiments = [
    {
        "label": r"V$_2$O$_3$, 300, 1500, 10",
        "merge": [
            {
                "sample_name": "DET70",
                "temperature_K": 300,
                "excitation_wl_nm": 1500.0,
                "fluence_mJ_cm2": 25.0,
                "time_window_fs": 250,
                "ref_type": "dark",
                "ref_value": 1466609,
                "delay_offset_fs": 9600.0,
                "csv_path": str(
                    delay_root
                    / "fluence_25p0mJ/time_window_250fs"
```

```
                    / "fitting/peak_fits_delay_phiavg_sum.csv"
                ),
            },
            {
                "sample_name": "DET70",
                "temperature_K": 300,
                "excitation_wl_nm": 1500.0,
                "fluence_mJ_cm2": 25.0,
                "time_window_fs": 40,
                "ref_type": "dark",
                "ref_value": 1466610,
                "delay_offset_fs": 9600.0,
                "csv_path": str(
                    delay_root
                    / "fluence_25p0mJ/time_window_40fs"
                    / "fitting/peak_fits_delay_phiavg_sum.csv"
                ),
            },
        ],
    },
    {
        "label": r"V$_2$O$_3$, 300, 1500, 4",
        "sample_name": "DET70",
        "temperature_K": 300,
        "excitation_wl_nm": 1500.0,
        "fluence_mJ_cm2": 10.0,
        "time_window_fs": 250,
        "ref_type": "dark",
        "ref_value": 1466613,
        "delay_offset_fs": 9600.0,
        "csv_path": str(
            delay_root
            / "fluence_10p0mJ/time_window_250fs"
            / "fitting/peak_fits_delay_phiavg_sum.csv"
        ),
    },
]

result = fitting.plot_time_evolution_multi(
    experiments=experiments,
    peak="110",
    _property="hkl_pos",
    out_csv_name="peak_fits_delay.csv",
    unit="ps",
    phi_mode="phi_avg",
    phi_reduce="sum",
    phi_window="Full",
    only_success=True,
    include_reference=True,
    as_lines=False,
    show_baseline_sigma=True,
    baseline_sigma=1.0,
    baseline_mode="errorbar",
    cmap="jet_r",
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

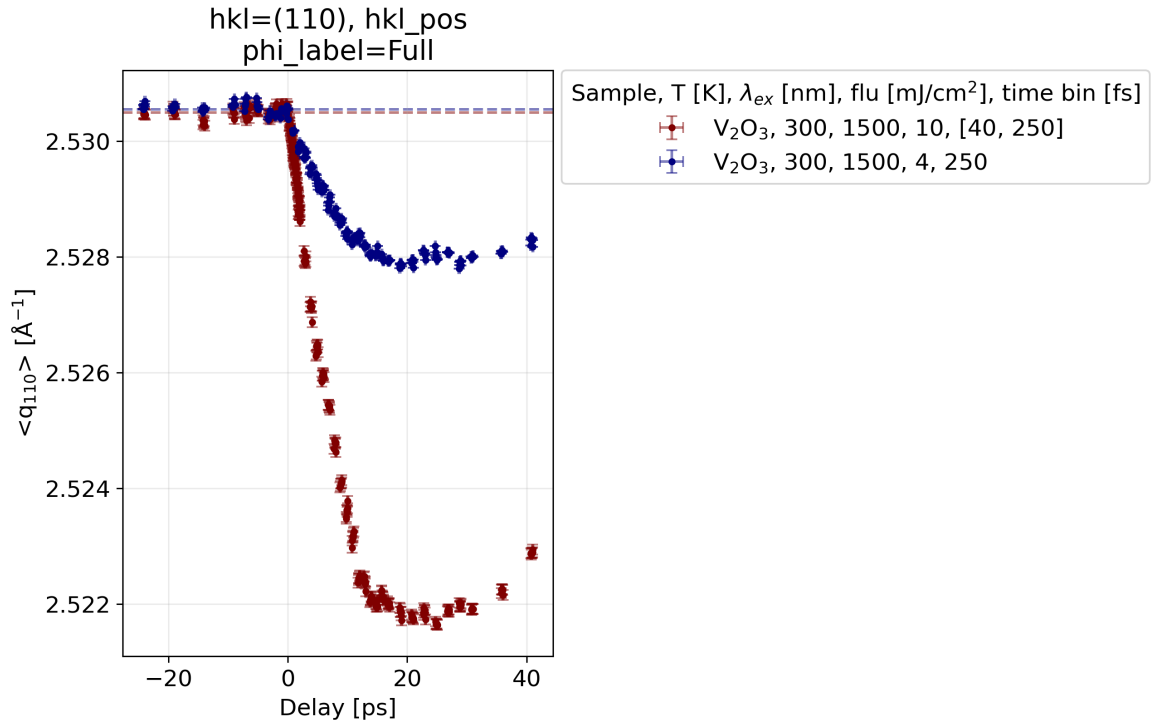


Figure 21.8: SACLA DET70 multi-delay fitting comparison for the fitted (110) position. The red series merges 40 fs and 250 fs time-window fits for the higher-fluence condition, while the blue series shows the lower-fluence 250 fs dataset.

21.2.2 SACLA fluence-scan examples from cached one-dimensional data

The SACLA fluence examples use a DET71 dataset at 300 K and 1500 nm. The single-experiment examples use the fixed-delay branch `delay_23979fs/time_window_250fs` with dark scan 1466631. A +9600 fs display offset is applied, so the fixed delay is reported as 33.6 ps. As in the SACLA delay examples, the stored fluence values are multiplied by 0.4 for display.

21.2.2.1 One-dimensional viewer for a SACLA fluence scan

The fluence viewer compares the fixed-delay patterns against a dark reference. The independent-variable labels in the legend use the display-scaled fluence values. The `fluence_digits` argument controls the precision of those labels without changing the delay precision used in the figure title.

```
from pathlib import Path

from trxrdpy.analysis.Spring8_SACLA import azimint as sacla_azimint

root = Path("/path/to/SACLA2024")
poni_path = root / "calibration/DET71_1466629.poni"
mask_path = root / "calibration/DET71_300K_mask.edf"

fig, axes = sacla_azimint.plot_1D_abs_and_diffs_fluence(
    sample_name="DET71",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    delay_fs=23979,
    time_window_fs=250,
    fluences_mJ_cm2="all",
```

```
    ref_type="dark",
    ref_value=1466631,
    poni_path=str(poni_path),
    mask_edf_path=str(mask_path),
    azim_window=(-90.0, 90.0),
    npt=1000,
    normalize=True,
    q_norm_range=(2.65, 2.75),
    compute_if_missing=True,
    overwrite_xy=False,
    xlim=(1.5, 4.0),
    fluence_scale=0.4,
    fluence_digits=3,
    delay_offset_fs=9600.0,
    fs_or_ps="ps",
    digits=1,
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    save_plots=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

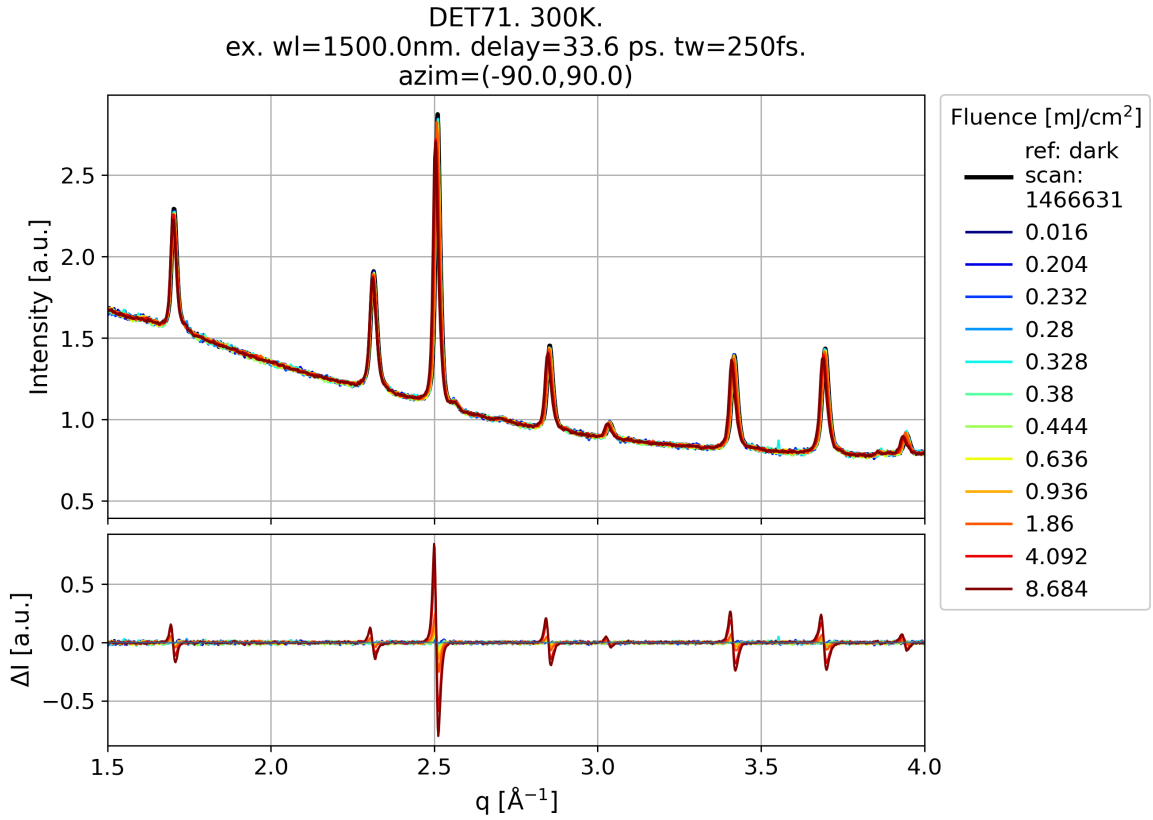


Figure 21.9: SACLA DET71 one-dimensional fluence viewer example at a displayed fixed delay of 33.6 ps. The top panel compares absolute full-azimuth patterns against dark scan 1466631, and the bottom panel shows fluence-dependent differences relative to the dark reference.

21.2.2.2 Differential-integral analysis for a SACLA fluence scan

The fluence differential analysis uses the same fixed-delay branch and dark reference. The (110) peak is integrated over $q = (2.4518, 2.55)$ Å^{-1}, with the background window placed after the peak.

```
from pathlib import Path

from trxrdpy.analysis import differential_analysis

root = Path("/path/to/SACLA2024")
poni_path = root / "calibration/DET71_1466629.poni"
mask_path = root / "calibration/DET71_300K_mask.edf"

peak_specs = {
    "110": {"q_range": (2.4518, 2.55), "bg_side": "right"},
}

df, fig = differential_analysis.plot_differential_integrals_fluence(
    sample_name="DET71",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    delay_fs=23979,
    time_window_fs=250,
    fluences_mJ_cm2="all",
    ref_type="dark",
    ref_value=1466631,
    poni_path=str(poni_path),
    mask_edf_path=str(mask_path),
    azim_window=(-90.0, 90.0),
    azim_offset_deg=-90.0,
    polarization_factor=0.99,
    peak="110",
    peak_specs=peak_specs,
    npt=1000,
    normalize_xy=True,
    q_norm_range=(2.65, 2.75),
    compute_if_missing=True,
    overwrite_xy=False,
    fluence_unit="mJ/cm$^2$",
    fluence_scale=0.4,
    delay_offset_fs=9600.0,
    fs_or_ps="ps",
    digits=1,
    show_errorbars=True,
    errorbar_scale=1.0,
    plot_abs_and_diffs=True,
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

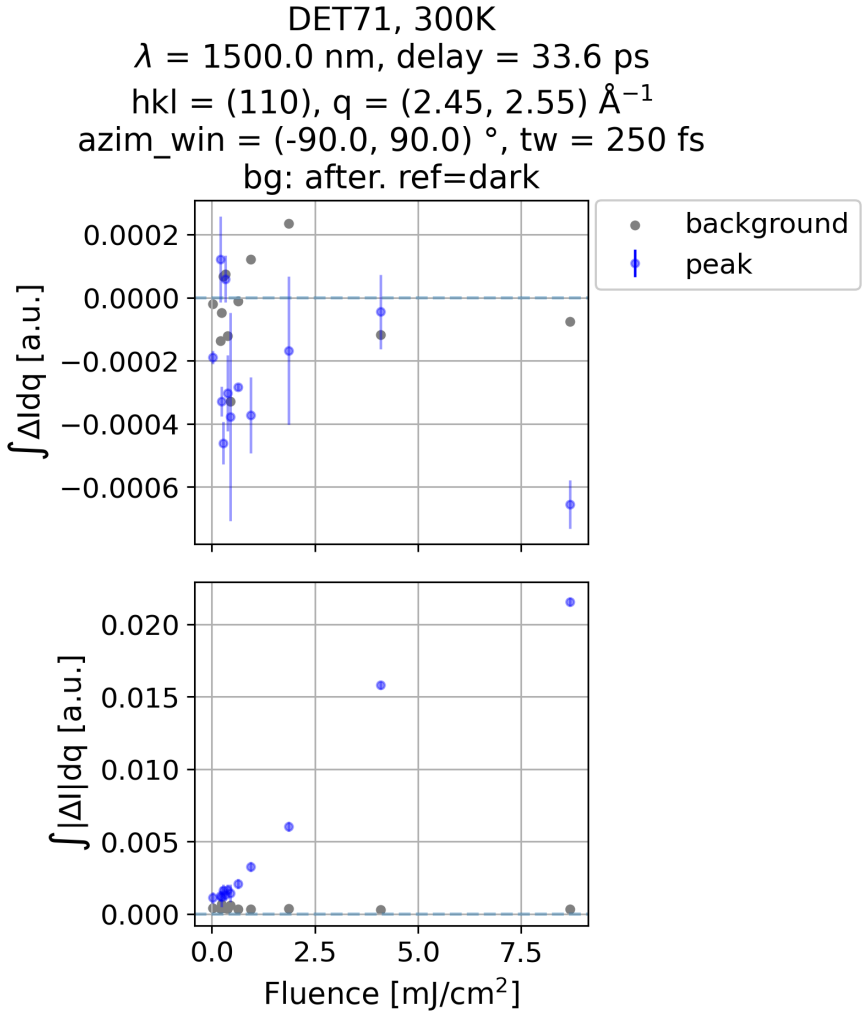


Figure 21.10: SACLA DET71 fluence-dependent differential-integral example for the (110) peak region at a displayed delay of 33.6 ps. The upper panel shows signed $\int \Delta I dq$, and the lower panel shows $\int |\Delta I| dq$.

21.2.2.3 Fitting analysis for different azimuthal sectors

The single-experiment fluence fitting example reads the cached `phi_avg` fitting CSV and plots the fitted (110) position for the full sector and selected symmetric $|\Phi|$ groups. The dark/reference fits are displayed as horizontal dashed lines.

```
from pathlib import Path

from trxrdpy.analysis import fitting

root = Path("/path/to/SACLA2024")

fig, ax, csv_path = fitting.plot_fluence_evolution(
    sample_name="DET71",
    temperature_K=300,
    excitation_wl_nm=1500.0,
    delay_fs=23979,
    time_window_fs=250,
    peak="110",
    _property="hkl_pos",
    unit="mJ/cm$^2$",
    out_csv_name="peak_fits_fluence.csv",
    groups=["Full", 60, 30, 0],
    phi_mode="phi_avg",
    phi_reduce="sum",
    as_lines=True,
    fluence_scale=0.4,
    fluence_offset=0.0,
    delay_offset_fs=9600.0,
    fs_or_ps="ps",
    digits=1,
    show_baseline_sigma=True,
    baseline_sigma=1.0,
    baseline_alpha=0.18,
    baseline_mode="errorbar",
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

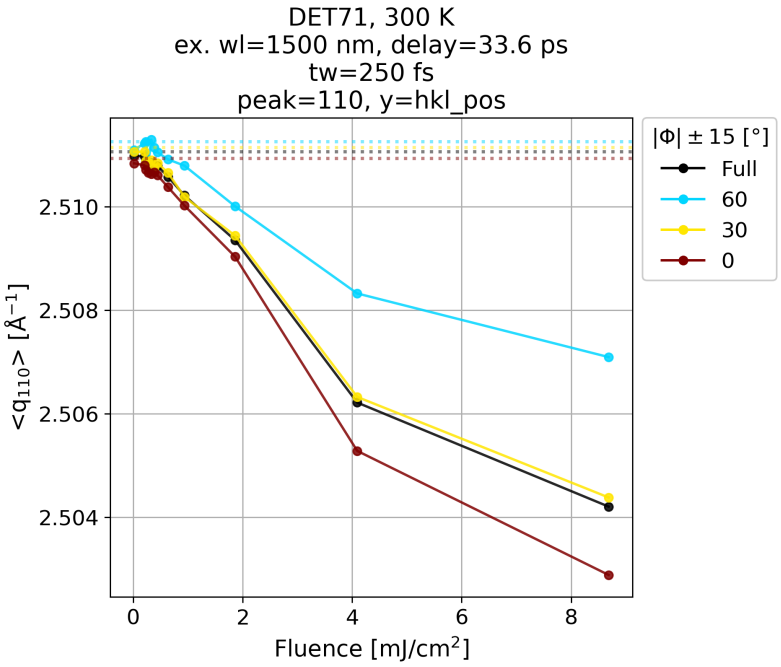


Figure 21.11: SACLA DET71 fitted (110) peak position versus fluence at a displayed delay of 33.6 ps. The plotted groups are the full $(-90°, 90°)$ range and selected `phi_avg` symmetric azimuthal sectors.

21.2.2.4 Multiple-experiment fluence fitting comparison

The multi-fluence comparison overlays two fixed-delay fluence scans. The first branch has stored delay −4042 fs and is displayed as 5.6 ps after the +9600 fs offset. The second branch has stored delay 23979 fs and is displayed as 33.6 ps. The dark/reference rows are included and plotted as horizontal dashed lines for each series.

```
from pathlib import Path

from trxrdpy.analysis import fitting

root = Path("/path/to/SACLA2024")
fluence_root = (
    root / "analysis/DET71/temperature_300K/excitation_wl_1500nm/fluence"
)

experiments = [
    {
        "label": r"(V,Cr)$_2$O$_3$, 300, 1500, 5.6, 250",
        "sample_name": "DET71",
        "temperature_K": 300,
        "excitation_wl_nm": 1500.0,
        "delay_fs": -4042,
        "time_window_fs": 250,
        "phi_mode": "phi_avg",
        "phi_reduce": "sum",
        "ref_type": "dark",
        "ref_value": 1466630,
        "fluence_scale": 0.4,
        "delay_offset_fs": 9600.0,
        "csv_path": str(
```

```
            fluence_root
            / "delay_-4042fs/time_window_250fs"
            / "fitting/peak_fits_fluence_phiavg_sum.csv"
        ),
    },
    {
        "label": r"(V,Cr)$_2$O$_3$, 300, 1500, 33.6, 250",
        "sample_name": "DET71",
        "temperature_K": 300,
        "excitation_wl_nm": 1500.0,
        "delay_fs": 23979,
        "time_window_fs": 250,
        "phi_mode": "phi_avg",
        "phi_reduce": "sum",
        "ref_type": "dark",
        "ref_value": 1466631,
        "fluence_scale": 0.4,
        "delay_offset_fs": 9600.0,
        "csv_path": str(
            fluence_root
            / "delay_23979fs/time_window_250fs"
            / "fitting/peak_fits_fluence_phiavg_sum.csv"
        ),
    },
]

fig, ax, saved_path = fitting.plot_fluence_evolution_multi(
    experiments=experiments,
    peak="110",
    prop="hkl_pos",
    group_by="azim_range_str",
    group="Full",
    fluence_unit="mJ/cm$^2$",
    fluence_scale=0.4,
    fs_or_ps="ps",
    digits=1,
    only_success=True,
    include_reference=True,
    as_lines=False,
    show_baseline_sigma=True,
    baseline_mode="errorbar",
    baseline_sigma_scale=1.0,
    show=False,
    legend_outside=True,
    save=True,
    path_root=root,
    analysis_subdir="analysis",
)
```

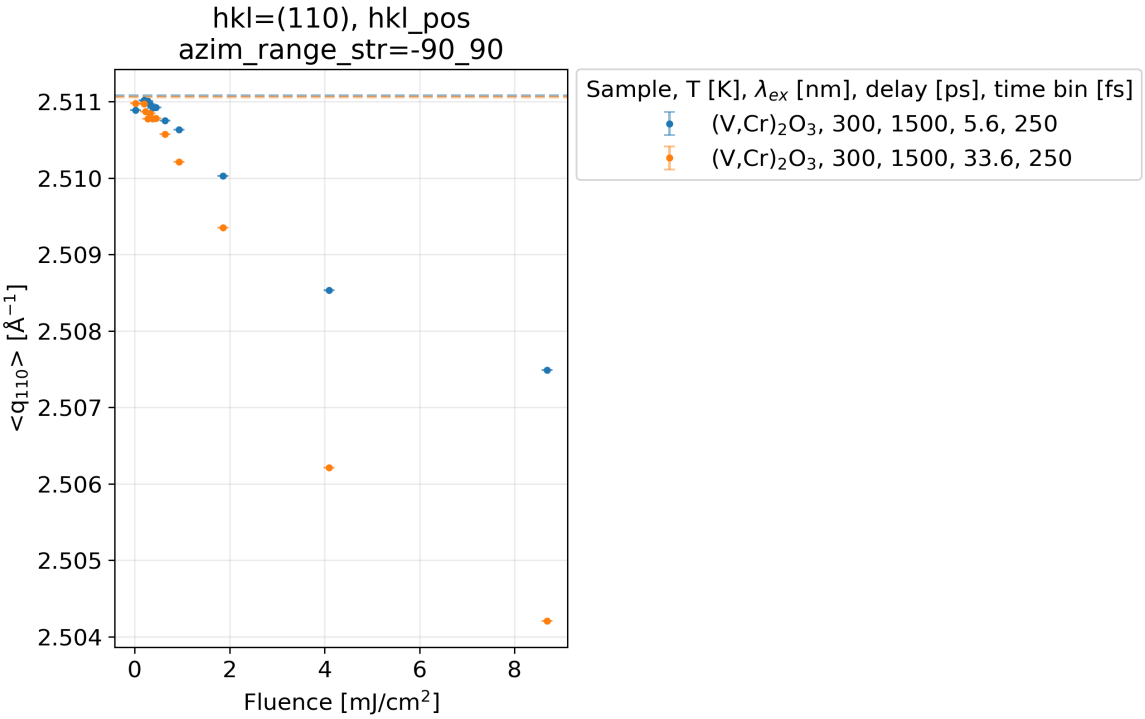


Figure 21.12: SACLA DET71 multi-experiment fluence comparison for the fitted (110) position in the full azimuthal range. The two series correspond to displayed delays of 5.6 ps and 33.6 ps, and the horizontal dashed lines show the included dark/reference fits.

Chapter 22

Analysis GUI workflows

22.1 GUI workflow map

The Analysis GUI is built around a shared `AnalysisGuiState`, a facility service, a path service, and backend services for preparation, calibration, integration, differential analysis, and fitting. The main window contains the following tabs: Session, 2D Image Creation, Calibration, 1D Pattern Creation, 1D Viewer, Differential Analysis, and Fitting Analysis.

The practical tab order is:

1. session and path setup;
2. preparation;
3. calibration;
4. pattern creation and viewer;
5. differential analysis;
6. fitting analysis; and
7. export and reproducibility recording.

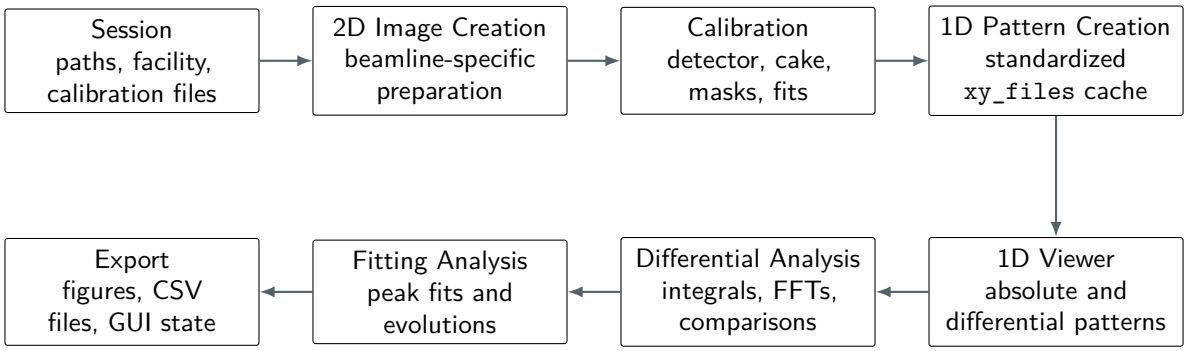


Figure 22.1: Analysis GUI workflow. The GUI first creates or locates beamline-specific images, validates calibration, and builds the standardized `xy_files` cache. The shared analysis path then drops to the 1D Viewer and continues through Differential Analysis, Fitting Analysis, and export.

22.2 Session and facility setup

The Session tab establishes the state used by the rest of the GUI. It selects the facility, experiment root, raw and analysis subdirectories, optional FemtoMAX ping-reference configuration, PONI path, mask path, azimuthal offset, and polarization factor. These values are not cosmetic. They determine which backend module is called and where files are read or written.

Fill the Session tab before running any analysis action. Changing the facility or analysis root after

generating files can make later tabs point to a different dataset than expected. For careful work, save the GUI state after the paths and detector settings are finalized.

A saved GUI state can be restored either with *Load GUI State...* or by dropping a saved `xrdpy_gui_state.json` file into the GUI-state path field and pressing *Load Path*. After loading, the field is updated to the resolved JSON path so the active state file is visible.

22.3 Preparation tab

The 2D Image Creation tab is the GUI route into facility-specific preparation. For FemtoMAX it exposes metadata creation, ping-reference handling, and averaged-image export. For ID09 it exposes delay selection and final-image creation. For SACLA it is an interface to the facility-dependent preparation routines and any required local environment.

Check the output of this tab before moving on. At minimum, identify which dark image and which delay or fluence images were created. If these images are missing or written under an unexpected sample/temperature folder, calibration and integration tabs will either fail or analyze the wrong data.

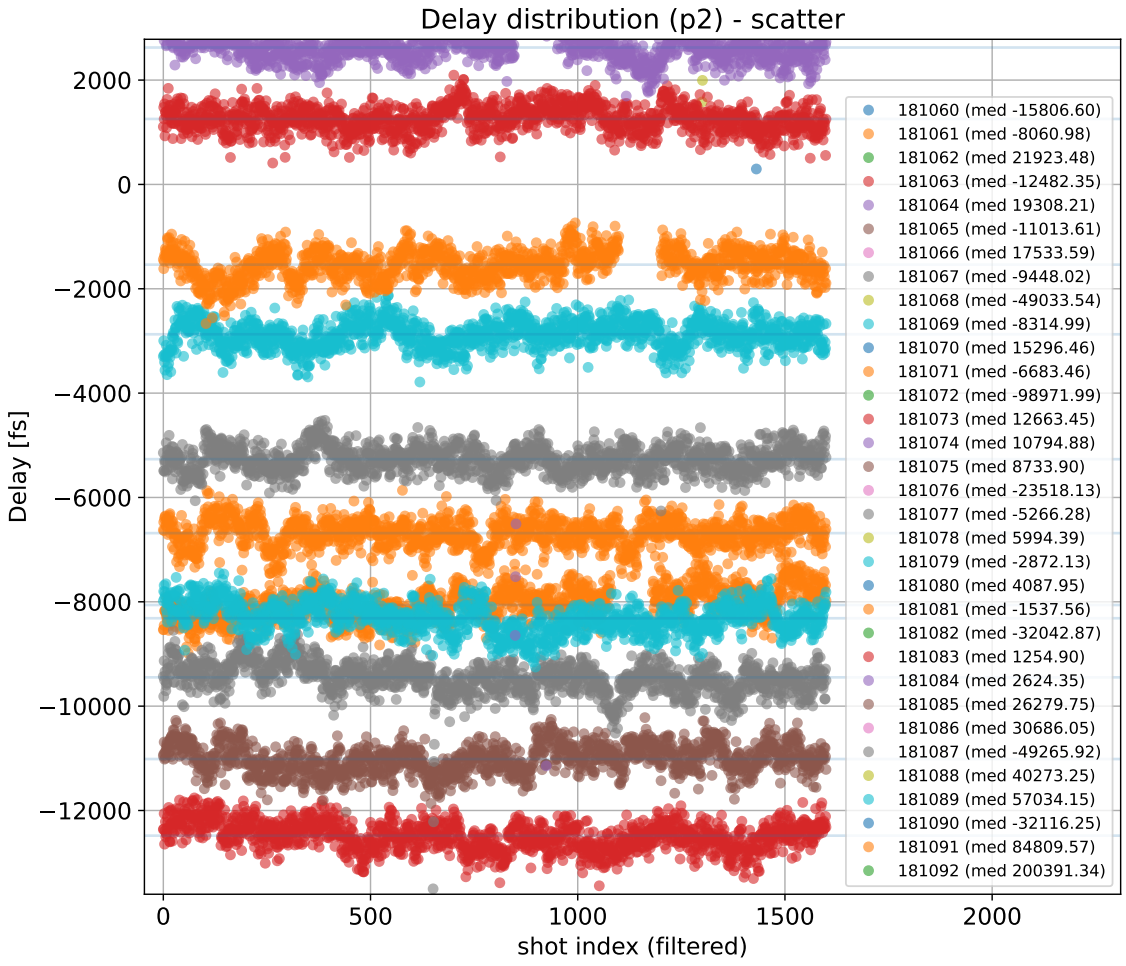


Figure 22.2: Analysis GUI preparation output example: FemtoMAX timing distribution inspected before delay-bin image production.

22.4 Calibration and pattern tabs

The Calibration tab wraps detector/cake plotting, calibration `xy` creation, calibration peak fitting, q-band cake azimuthal-distribution analysis, caked pattern plotting, and property-versus-azimuth visualization. The controls map directly to calibration arguments: scan selector, azimuthal edges, full-range inclusion, q range, normalization range, mask usage, azimuthal offset, polarization factor, q-band/background selectors, phi-window selectors, and save options.

The 1D Pattern Creation tab computes standardized `xy` files for dark, delay, or fluence datasets. The 1D Viewer tab loads or computes patterns and plots absolute and differential views. Use these two tabs together: creation makes the cache explicit, and the viewer checks whether the resulting patterns are scientifically usable.

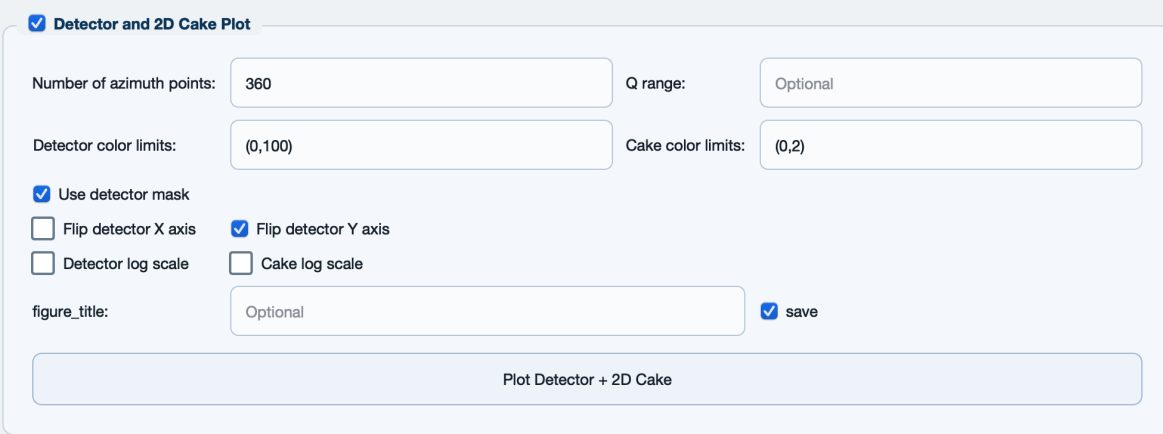


Figure 22.3: Analysis GUI calibration controls for detector/cake plotting. The same panel controls detector-axis inversion, mask use, detector and cake color limits, q range, azimuthal binning, and save behavior.

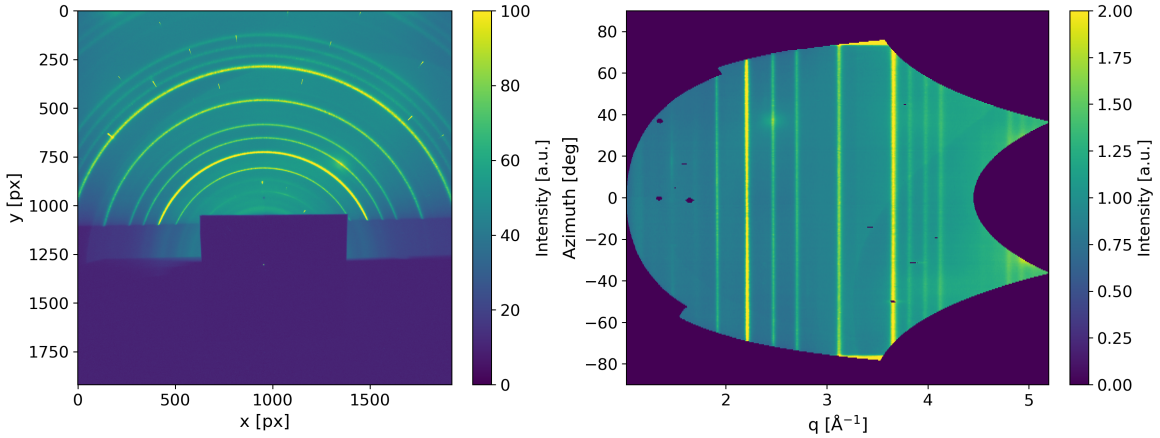


Figure 22.4: Analysis GUI calibration output example: detector image and 2D cake diagnostic produced before one-dimensional integration.

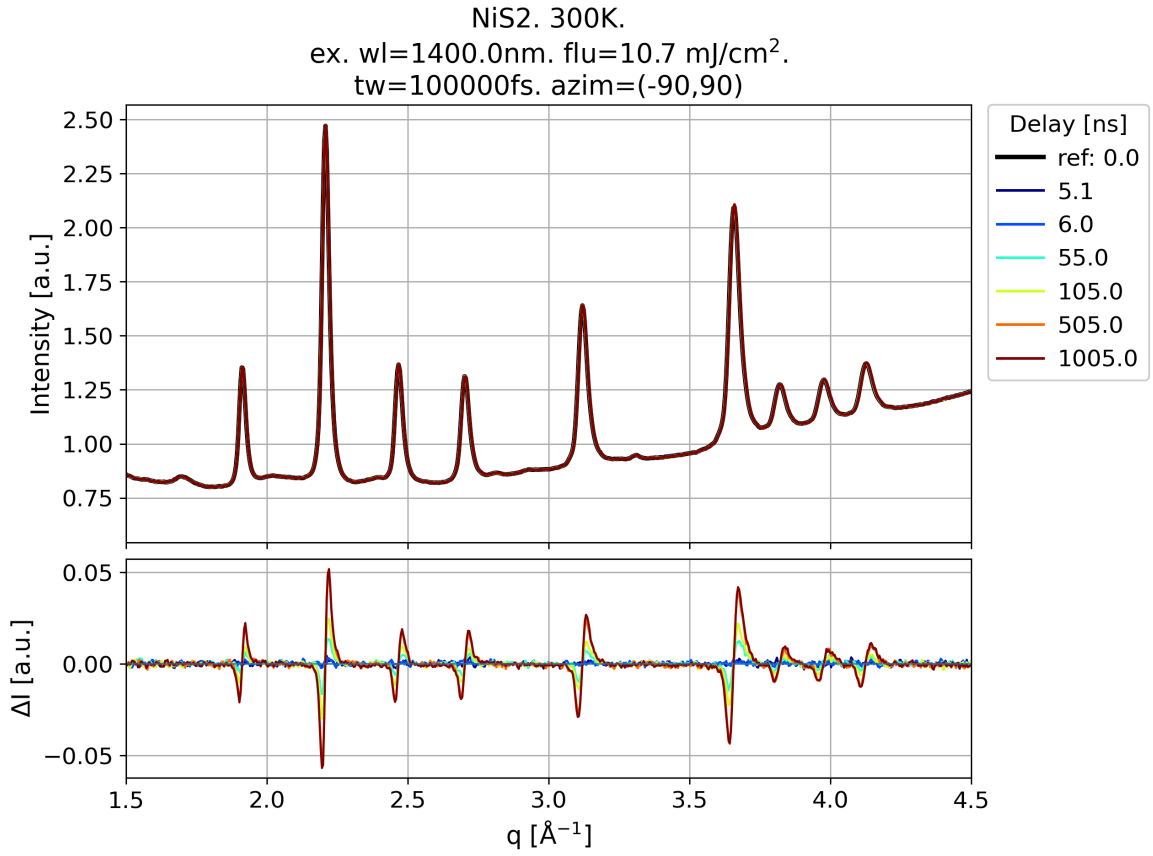


Figure 22.5: Analysis GUI 1D Viewer output example: absolute delay patterns and their differential traces after cached `xy` files have been selected or generated.

22.5 Differential and fitting tabs

The Differential Analysis tab collects peak/background definitions and reference settings before plotting integrals, FFTs, or multi-experiment comparisons. The Fitting Analysis tab collects peak specifications, azimuthal windows, reference settings, and output choices before calling the fitting backend. Both tabs depend on the pattern cache and detector settings established earlier.

Do not use the GUI as a black box. After a differential or fit plot is produced, record the peak specification, reference choice, azimuthal window, q normalization range, and output CSV path. Those values are the bridge from interactive exploration to reproducible analysis.

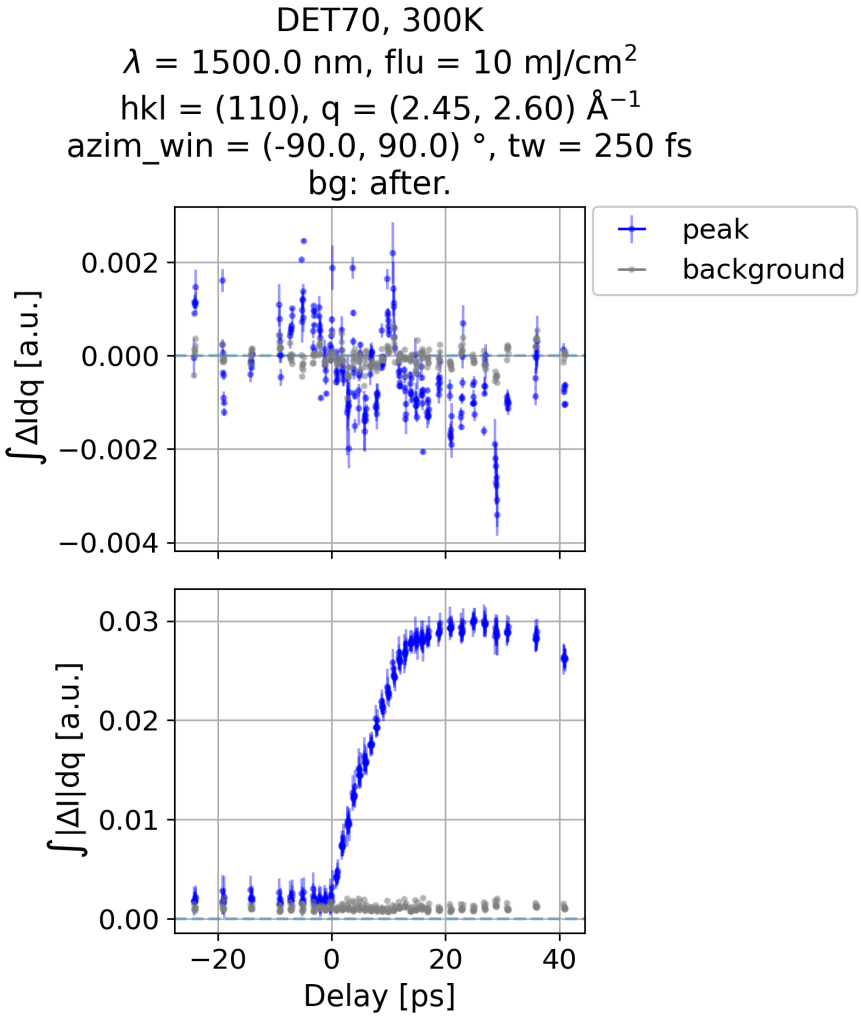


Figure 22.6: Analysis GUI Differential Analysis output example: signed and absolute differential integrals for a selected peak/background definition.

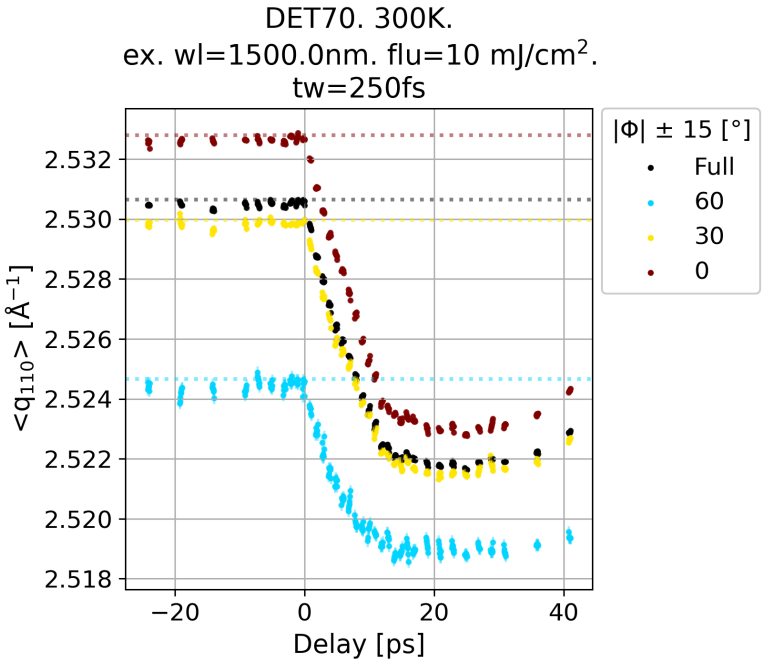


Figure 22.7: Analysis GUI Fitting Analysis output example: fitted peak position versus delay for several azimuthal sectors.

Chapter 23

Analysis API reference and troubleshooting

23.1 User-facing API map

The Analysis API is large. This manual documents the user-facing surface needed for real workflows and avoids turning internal helpers into public commitments. Internal utilities appear only when they define path naming, conventions, or important validation behavior.

Module	Documentation role
`trxrdpy.analysis.calibration`	User-facing calibration, cake diagnostics, q-band cake azimuthal-distribution analysis, calibration fitting, and pattern comparison.
`trxrdpy.analysis.MaxIV_FemtoMAX`	FemtoMAX metadata, dark/delay/fluence preparation, and facility wrappers.
`trxrdpy.analysis.ESRF_ID09`	ID09 raw-data reduction, delay handling, integration, and plotting.
`trxrdpy.analysis.Spring8_SACLA`	SACLA metadata, dark/background processing, final-image creation, and shared integration exposure.
`trxrdpy.analysis.fitting`	Delay and fluence peak fitting, fit overlays, and evolution plots.
`trxrdpy.analysis.differential_analysis`	Differential traces, FFTs, and multi-experiment comparison.
`trxrdpy.analysis.gui`	Graphical workflow controller for the same backend operations.

23.2 Common analysis failures

Most analysis failures fall into one of a few categories. The error message usually points to a missing file or invalid selector, but the root cause is often an earlier mismatch in paths, calibration, or reference choice.

- missing or inconsistent raw-data paths;
- scan tags that do not match cached outputs;
- wrong PONI or detector mask;
- inverted detector axes or azimuthal offset mistakes;
- empty integrations caused by q or azimuthal limits;
- empty q bands or background bands in cake azimuthal-distribution analysis;
- failed peak fits caused by poor initial windows or baseline choices;
- inconsistent reference delay or fluence choices; and
- GUI/backend environment mismatches.

When a failure occurs, debug in pipeline order. First confirm that the raw files exist and that the path configuration points to them. Then confirm that facility preparation produced the expected two-dimensional images. Then confirm that the PONI and mask load correctly and produce a sensible cake. Then confirm that q bands, background bands, and azimuthal limits actually intersect the computed cake grid. Then confirm that the `xy` files correspond to the current integration settings. Only after those checks is differential-analysis or fitting code the likely source of the problem.

23.3 Publication checklist

Before publishing an analysis result, record the facility, scan identifiers, raw and analysis paths, calibration file, mask file, polarization factor, azimuthal windows, q ranges, normalization method, reference choice, fitting windows, q-band/background definitions for cake azimuthal-distribution analysis, software version, and output filenames.

For time-resolved work, also record the delay units, time-zero offset, time-window width, timing-reference source, and whether invalid shots were rejected. For fluence work, record the fluence units, fixed delay, excitation wavelength, and how fluence values were assigned to scans. For multi-experiment comparisons, record whether traces were plotted on an absolute scale or normalized for comparison.

For software releases, publish the compiled `XRDpy_manual.pdf` together with the package version. The local source figures and experimental folders used to build the examples are not required in the source distribution when the released PDF is included.

Part III

Shared reference and appendices

Chapter 24

Project and contribution information

Project record

Use the package version, repository URL, Zenodo DOI, and saved analysis or simulation parameters when citing or reproducing results made with XRDpy.

24.1 Versioning and compatibility

Record the installed XRDpy package version with every simulation or analysis result. A saved figure should be traceable to a script, GUI state, parameter file, or CSV table that includes the package version and the relevant experimental or simulated geometry.

24.2 Reporting issues

Useful issue reports include the operating system, Python version, package version, installation route, minimal script or GUI state, complete error message, and the smallest input file needed to reproduce the behavior. For analysis issues, include the facility, path layout, PONI file, mask file, selected scan or delay, and whether cached `xy` files were reused.

24.3 Citation, license, and acknowledgments

The project repository and Zenodo concept DOI are listed in the front matter. Version-specific Zenodo records should be used when a publication depends on a specific release. The package is distributed under the Creative Commons Attribution 4.0 International license.

Bibliography

[1] Michael Wulff et al. "The Realization of Sub-Nanosecond Pump and Probe Experiments at the ESRF." *Faraday Discussions* 122 (2003), pp. 13–26.

[2] Marco Cammarata et al. "Chopper System for Time Resolved Experiments with Synchrotron Radiation." *Review of Scientific Instruments* 80.1 (2009), 015101.

[3] European Synchrotron Radiation Facility. *ID09 beamline: Time-resolved studies*. https://www.esrf.fr/home/UsersAndScience/Experiments/CBS/ID09/o.html.

[4] Henrik Enquist et al. "FemtoMAX – an X-ray Beamline for Structural Dynamics at the Short-Pulse Facility of MAX IV." *Journal of Synchrotron Radiation* 25.2 (2018), pp. 570–579.

[5] Aymeric Robert et al. "MAX IV Laboratory." *The European Physical Journal Plus* 138.6 (2023), 495.

[6] M. Yabashi et al. "Compact XFEL and AMO Sciences: SACLA and SCSS." *Journal of Physics B: Atomic, Molecular and Optical Physics* 46.16 (2013), 164001.

[7] Makina Yabashi, Hitoshi Tanaka, and Tetsuya Ishikawa. "Overview of the SACLA Facility." *Journal of Synchrotron Radiation* 22.Pt 3 (2015), pp. 477–484.

[8] Tetsuo Katayama et al. "A Beam Branching Method for Timing and Spectral Characterization of Hard X-ray Free-Electron Lasers." *Structural Dynamics* 3.3 (2016), 034301.

[9] Jérôme Kieffer and Dimitrios Karkoulis. "PyFAI, a Versatile Library for Azimuthal Regrouping." *Journal of Physics: Conference Series* 425.20 (2013), 202012.

[10] pyFAI developers. *pyFAI documentation*. https://pyfai.readthedocs.io/en/stable/pyFAI.html.

[11] E. B. Knudsen, H. O. Sørensen, J. P. Wright, G. Goret, and J. Kieffer. "FabIO: easy access to two-dimensional X-ray detector images in Python." *Journal of Applied Crystallography* 46.2 (2013), pp. 537–539.

[12] FabIO developers. *FabIO: an I/O library for images produced by 2D X-ray detectors*. PyPI project page, https://pypi.org/project/fabio/.

[13] James Hester. *PyCifRW: CIF/STAR file support for Python*. PyPI project page, https://pypi.org/project/PyCifRW/.

[14] Dominik Kriegner, Eugen Wintersberger, and Julian Stangl. "xrayutilities: a versatile tool for reciprocal space conversion of scattering data recorded with linear and area detectors." *Journal of Applied Crystallography* 46 (2013), pp. 1162–1170.

[15] xrayutilities developers. *xrayutilities documentation*. https://xrayutilities.sourceforge.io/.

[16] Matthew Newville, Till Stensitzki, Renee Otten, and others. *LMFIT: Non-Linear Least-Squares Minimization and Curve-Fitting for Python*. Zenodo concept DOI, https://doi.org/10.5281/zenodo.598352; documentation, https://lmfit.github.io/lmfit-py/.

[17] `pytxs` developers. ***`pytxs`**: time-resolved x-ray scattering*. PyPI project page, https://pypi.org/project/pytxs/.

Chapter A

Glossary

Bragg condition Geometrical condition under which elastic scattering from lattice planes is constructive.

CIF Crystallographic Information File containing structured crystallographic metadata.

Direct lattice Real-space periodic basis of the crystal.

Ewald sphere Reciprocal-space construction representing elastic-scattering solutions at fixed wavelength.

HKL Miller-index triplet (h, k, l) identifying a reciprocal-lattice vector or lattice-plane family.

PONI Point of normal incidence in pyFAI detector geometry; also the calibration-file format used to store that geometry.

Reciprocal lattice Fourier-space lattice used to represent diffraction vectors.

q Magnitude of the scattering vector, expressed here in Å^{-1}.

Two theta Angle between incident and diffracted beam directions.

Chapter B

Package map

Module	Responsibility
`trxrdpy.simulation.utils`	Unit conversion, rotation matrices, and homogeneous transforms.
`trxrdpy.simulation.cif`	CIF parsing and validation.
`trxrdpy.simulation.sample`	Lattice, reflections, Bragg checks, and powder intensities.
`trxrdpy.simulation.poni`	PONI parsing and normalization.
`trxrdpy.simulation.detector`	Area-detector geometry and lab grid.
`trxrdpy.simulation.geometry`	Motors, motor chains, and paired geometries.
`trxrdpy.simulation.diffractometers`	Predefined geometry factories and serialization.
`trxrdpy.simulation.experiment`	Combined experiment calculations and plots.
`trxrdpy.simulation.polycrystalline`	High-level 1D, 2D, and 3D powder workflows.
`trxrdpy.simulation.single_crystal`	High-level single-crystal simulation, scans, and targeting.
`trxrdpy.simulation.plot`	Simulation visualization functions.
`trxrdpy.simulation.gui`	Qt graphical simulation workflows.

Chapter C

Citation and license

If XRDpy contributes to academic work, cite the Zenodo record corresponding to the version used. The concept DOI for all versions is https://doi.org/10.5281/zenodo.18634909. Version-specific citation metadata is available from Zenodo.

The project is distributed under the Creative Commons Attribution 4.0 International license (CC BY 4.0). Consult the repository's `LICENSE` file for the complete terms.