ImSwitch

unknown

Jun 22, 2023

SOFTWARE INFO

1 Installation **3**
- 1.1 Option A: Standalone bundles for Windows . . . 3
- 1.2 Option B: Install using pip . . . 3

2 Changelog **5**
- 2.1 1.2.1 . . . 5
- 2.2 1.2.0 . . . 5
- 2.3 1.1.0 . . . 6
- 2.4 1.0.0 . . . 6

3 How to contribute **7**
- 3.1 Reporting bugs . . . 7
- 3.2 User feedback . . . 7
- 3.3 Suggest a new feature . . . 7
- 3.4 Improving documentation . . . 7
- 3.5 Adding device support . . . 8
- 3.6 Automated tests . . . 8

4 Graphical user interface **9**
- 4.1 Detector Settings . . . 10
- 4.2 Recording and data storage . . . 12
- 4.3 Data visualization module . . . 13
- 4.4 Hardware control . . . 13
- 4.5 Alignment tools . . . 16

5 Use cases **21**
- 5.1 Parallelized confocal and RESOLFT (MoNaLISA) . . . 21
- 5.2 Point-scanning confocal and STED . . . 23
- 5.3 CoolLED control through USB and TTLs using a NIDAQ . . . 25

6 Scripting **27**

7 Module configuration **29**

8 HDF5 datafiles **31**
- 8.1 Datasets . . . 31
- 8.2 Object attributes . . . 31

9 Hardware control configurations **33**
- 9.1 How configurations are defined . . . 33
- 9.2 Configuration file specification . . . 34

9.3 Item types that may be included . . . 35
9.4 Available managers . . . 38
9.5 Available signal designers . . . 40

10 Adding support for more devices **43**
10.1 How device managers are implemented . . . 43
10.2 Base class documentation . . . 43
10.3 Available low-level managers . . . 49

11 Global-level functions **51**

12 api.imcontrol **53**

13 mainWindow **57**

Index **59**

`ImSwitch` is a software solution in Python that aims at generalizing microscope control by using an architecture based on the model-view-presenter (MVP) design pattern and enabling flexible control of multiple microscope modalities.

The constant development of novel microscopy methods with an increased number of dedicated hardware devices poses significant challenges to software development. ImSwitch is designed to be compatible with many different microscope modalities and customizable to the specific design of individual custom-built microscopes, all while using the same software. We would like to involve the community in further developing ImSwitch in this direction, believing that it is possible to integrate current state-of-the-art solutions into one unified software.

In this documentation page you will find all information you need about the installation, usage and development of ImSwitch, both from the user perspective (GUI description and use cases) as well as for developers (scripting and API modules, and hardware control and JSON config files).

CHAPTER

ONE

INSTALLATION

1.1 Option A: Standalone bundles for Windows

Windows users can download ImSwitch in standalone format from the releases page on GitHub. Further information is available there. An existing Python installation is *not* required.

1.2 Option B: Install using pip

ImSwitch is also published on PyPI and can be installed using pip. Python 3.7 or later is required. Additionally, certain components (the image reconstruction module and support for TIS cameras) require the software to be running on Windows, but most of the functionality is available on other operating systems as well.

To install ImSwitch from PyPI, run the following command:

```
pip install ImSwitch
```

(Developers installing ImSwitch from the source repository should run `pip install -r requirements-dev.txt` instead.)

You will then be able to start ImSwitch with this command:

```
imswitch
```

CHAPTER

TWO

CHANGELOG

2.1 1.2.1

Highlights:

- Snaps can now be saved to the image viewer (#64)
- Snaps can now be saved as tiff files (#75)
- Resolved the issue of not being able to run scans with only one positioner or only one laser
- Fixed the step up/down buttons not working properly for multi-axis positioners
- Fixed the api.imcontrol.setDetectorToRecord method not working

A list of all code changes is available on GitHub: https://github.com/kasasxav/ImSwitch/compare/v1.2.0...v1.2.1

2.2 1.2.0

Highlights:

- Saving multi-detector and timelapse recordings in a single file is now supported (#53)
- Selecting specific detectors to record is now supported (#52)
- It is now possible to edit values/on-off-state of non-involved lasers during scanning (#51)
- The image reconstruction module now allows reconstructing all loaded data files (e.g. multi-file timelapses) into a single reconstruction (#34)
- The documentation has been improved (#61, #63)
- Fixed the SLM widget causing crashes on macOS (#57)
- Fixed the module picker being empty in standalone Windows bundles (#55)

A list of all code changes is available on GitHub: https://github.com/kasasxav/ImSwitch/compare/v1.1.0...v1.2.0

2.3 1.1.0

Highlights:

- ImSwitch is now available to install from PyPI, and standalone Windows bundles are also available to download from the releases page on GitHub. (#38)
- User configuration files are now saved to an appropriate user directory. On Windows, this is the documents directory, and on other operating systems it's the user's home directory. (#40)
- Added a Tools menu item for setting active modules. (#27)
- Added an image shifting tool to the hardware control module. (#30)
- Added support for presets to laser widget. (#25)
- The laser widget is now a vertical list instead of a horizontal one. (#24)
- Resolved the issue of timelapse recordings sometimes containing too few frames. (#33)

A list of all code changes is available on GitHub: https://github.com/kasasxav/ImSwitch/compare/v1.0.0...v1.1.0

2.4 1.0.0

Initial release.

CHAPTER

THREE

HOW TO CONTRIBUTE

We want to encourage users and developers to give active feedback on their experience and feel free to contribute.

First, we feel very strongly about being inclusive and respectful to other contributors. Please follow the Python Community guidelines if you wish to contribute to our project.

There are different ways you can contribute:

3.1 Reporting bugs

If you encounter a bug, you can directly report it in the issues section. Please describe how to reproduce the bug and include as much information as possible that can be helpful for fixing it.

Have you written code that fixes the bug? You can open a new pull request or include your suggested fix in the issue.

3.2 User feedback

We would like to hear about your experience when using ImSwitch and suggestions for improvement. You can do that by starting a thread in the discussion section on GitHub.

3.3 Suggest a new feature

If you are missing features and want to develop ImSwitch further, you can start a discussion and brainstorm your suggestions. Feel free to open a pull request, but we believe it's essential to have feedback from the community if you want to add new functionality.

3.4 Improving documentation

We would like to include your use-case into the documentation! Feel free to open a discussion thread about what you wish to include or improve.

3.5 Adding device support

See *this page* for information on adding support for new devices. You can start a discussion if you want to get advice and help to get started.

3.6 Automated tests

We want to keep including automated tests into the development process, so if you have contributed to the project by fixing a bug or providing new functionality, we encourage you to write code that tests your contribution as well.

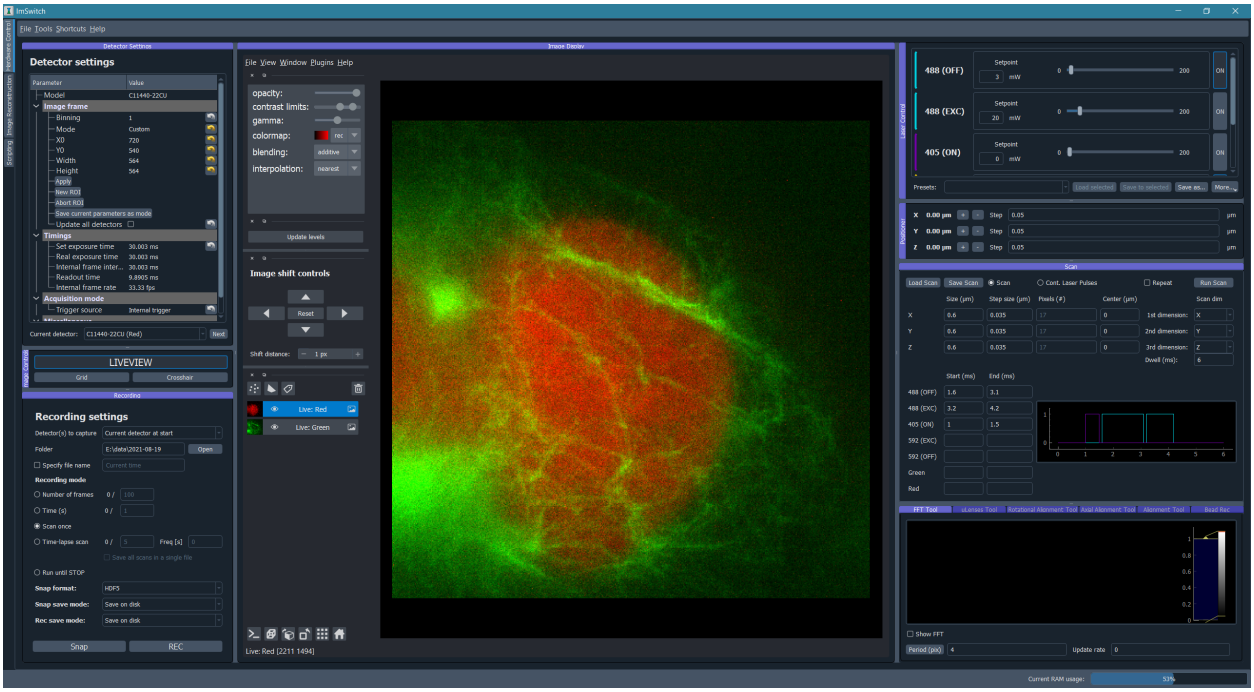
ImSwitch's pytests are located inside the `_test/unit/` and `_test/ui/` folders under each ImSwitch module's directory. So if you have added a new widget to the hardware control module and want to write UI tests for it, you should place the UI test files in the `/imswitch/imcontrol/_test/ui/` folder.

The test suite will automatically be run on commits and pull requests on GitHub. You can also run it manually by executing the command `python -m pytest --pyargs imswitch` in the root directory of the repository.

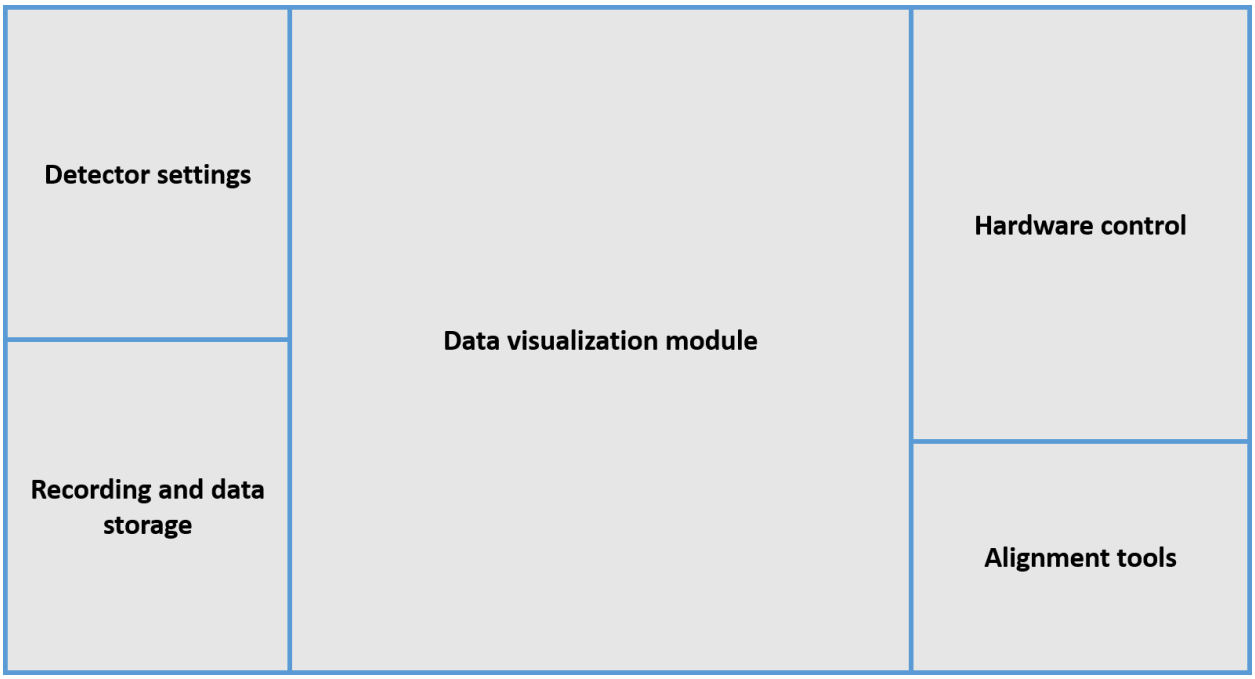
CHAPTER

FOUR

GRAPHICAL USER INTERFACE



The ImSwitch GUI is divided in different modules in order to make it intuitive to explore for both users and developers.



4.1 Detector Settings

This section interacts with the different detectors of the system. These can be either Cameras or Point Detectors. The user has access to different parameters like the subarray size (to determine the field-of-view used in a camera), a set of ROIs to use (which can be updated in real time by saving the current ROI), and other properties like the trigger type and exposure time. The parameters shown will depend on the type of detector selected, and additional parameters can be added by the developer in the specific DetectorManagers.

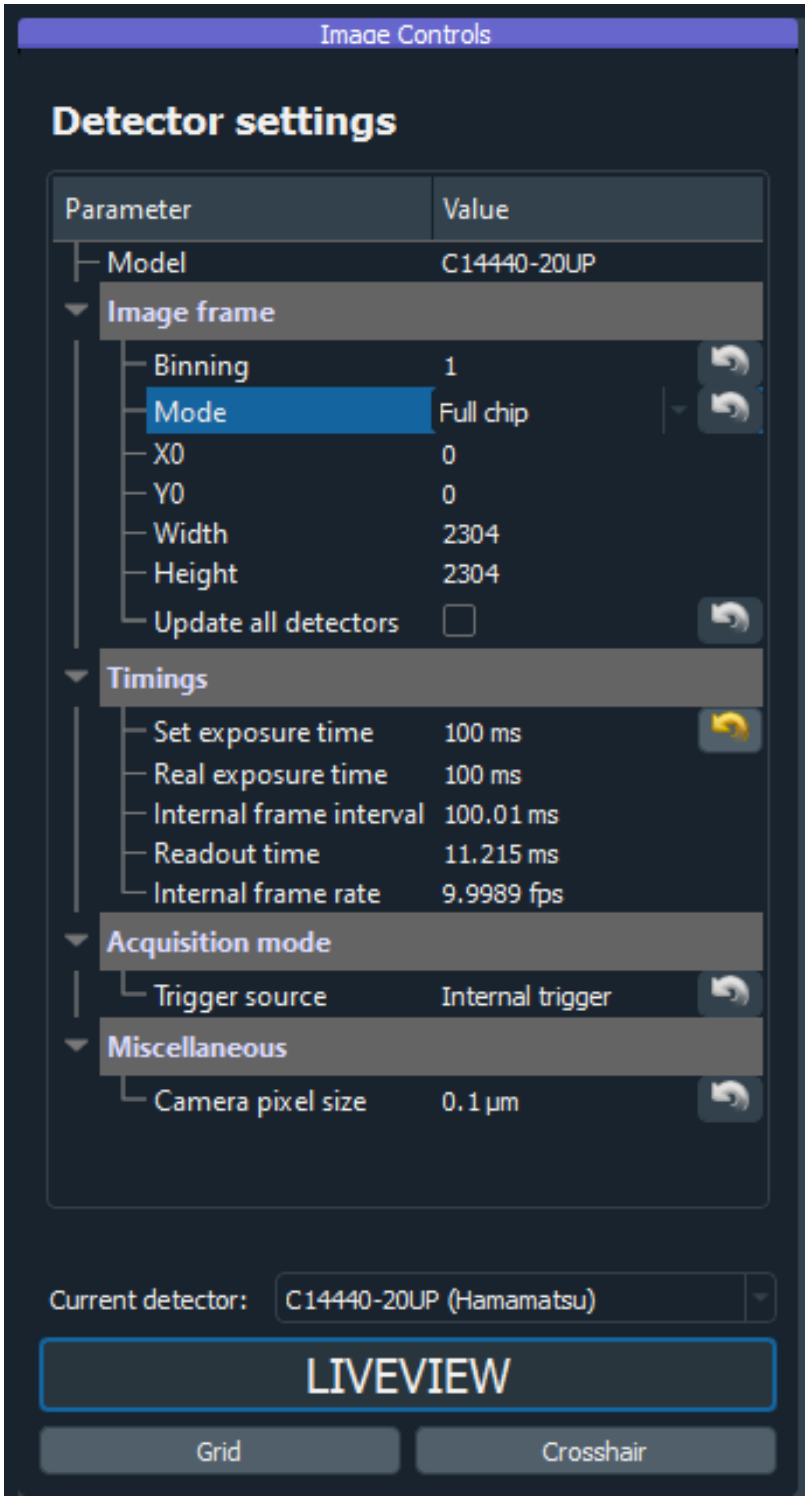


Image Controls
Detector settings
Parameter
Value
Model
C14440-20UP
Image frame
Binning
1
Mode
Full chip
X0
0
Y0
0
Width
2304
Height
2304
Update all detectors
Timings
Set exposure time
100 ms
Real exposure time
100 ms
Internal frame interval
100.01 ms
Readout time
11.215 ms
Internal frame rate
9.9989 fps
Acquisition mode
Trigger source
Internal trigger
Miscellaneous
Camera pixel size
0.1 µm
Current detector:
C14440-20UP (Hamamatsu)
LIVEVIEW
Grid
Crosshair

4.2 Recording and data storage

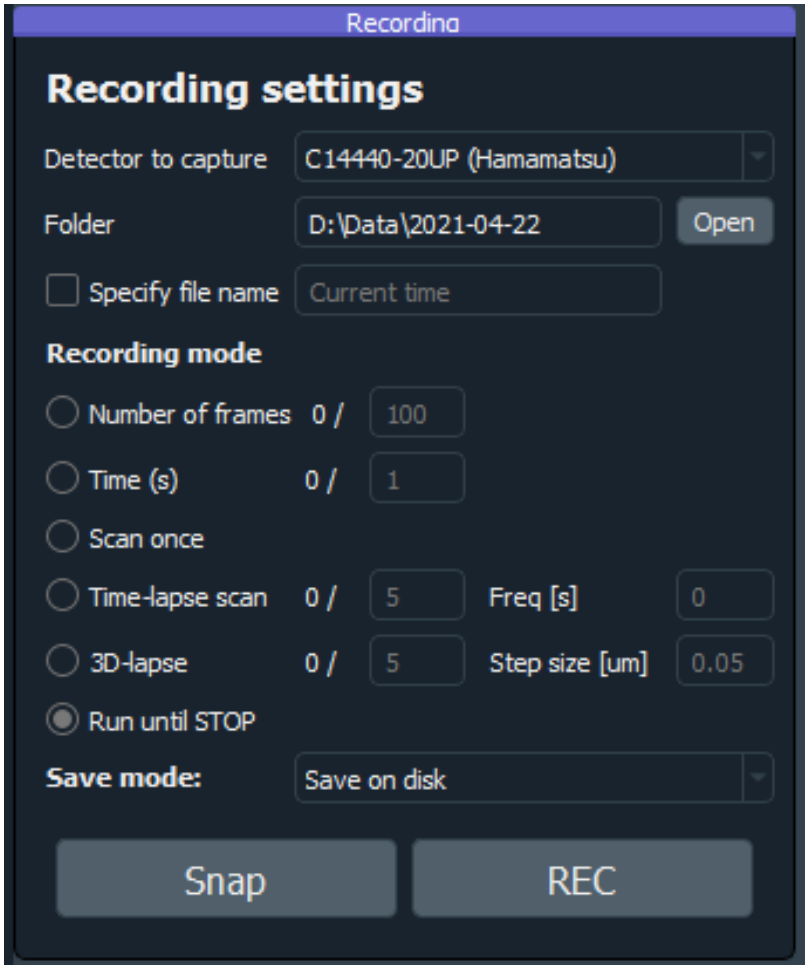
Either during a scan or free running mode, the recording section will make sure to retrieve all the incoming images from the DetectorManagers selected and save them.

The images can be saved in the disk, in RAM to be shared to the Image Processing module, or both. There are different recording modules depending on the type of recording:

- **Number of frames**: the user specifices the number of frames to be saved.
- **Time (s)**: similar as above but specifying the time instead.
- **Scan once**: the recording will stop once the current scan does.
- **Timelapse**: there will be sequential recordings spaced by the time that the user inputs. Each image from the scan (or raw frames) will be saved in a different file.
- **3D Lapse**: same as timelaps but moving the positioner in between, alternative way to perform a 3D scan.
- **Run until stop**: the recording thread will run until it's stopped by the user.

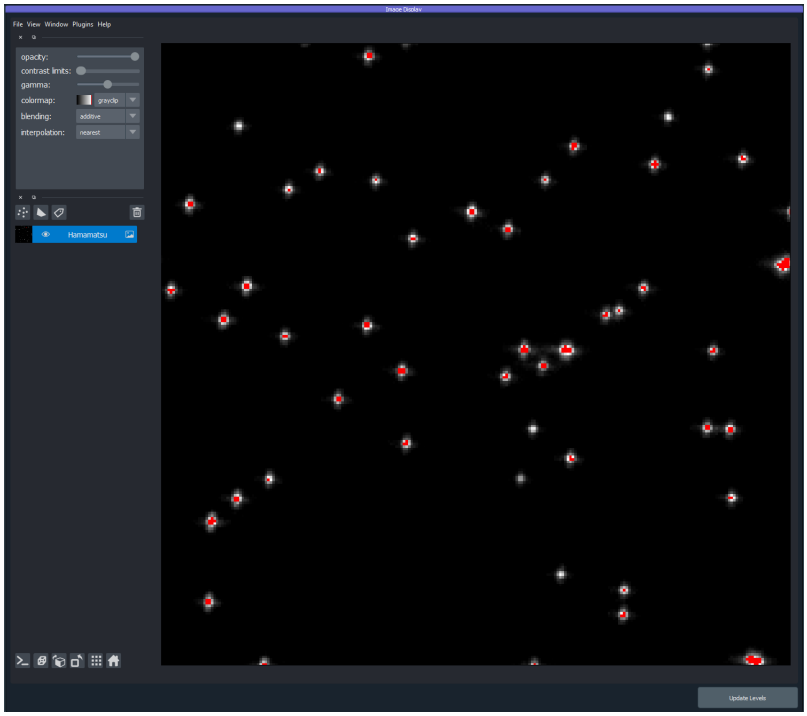
The data will be saved in hdf5 together with all user-interactable parameters of ImSwitch (laser power, scan parameters, etc).

Images can additionally be saved after a scan, using the Snap button. This is the use-case for procedurally-updated images while using for example point-detectors as in confocal or STED imaging.



4.3 Data visualization module

We incorporated Napari for visualizing the real-time images from the detectors. It works with both point detectors and cameras, and Napari offers good support for displaying multiple channels. Additionally and one could add plugins for data analysis or visualization tools. This module enables the use of multiple cameras and point detectors simultaneously. Point detector images are updated on a line-by-line basis during the acquisition.

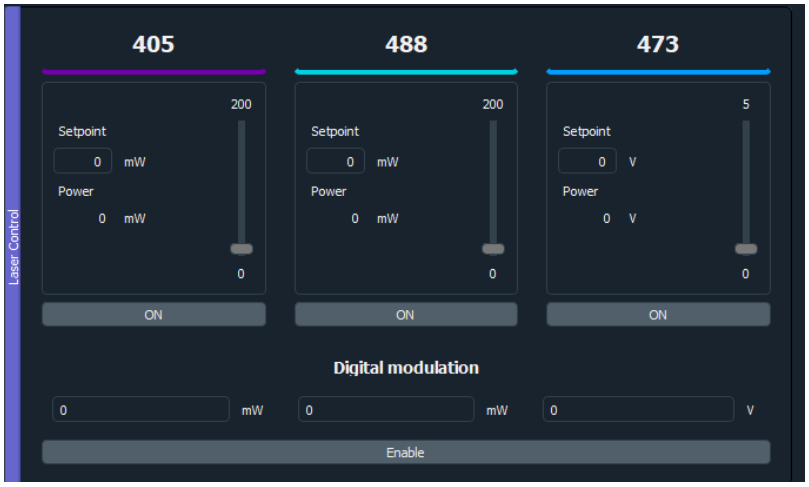


4.4 Hardware control

In this section the modules for hardware control are implemented. The main components are the laser widget, scanning widget, positioner widget, focus lock widget, and SLM widget. Developers can easily add new modules following the main structure of ImSwitch, for controlling additional hardware or controlling differently current hardware. The hardware control widgets necessary are automatically loaded depending on the user-defined settings in the JSON configuration files.

4.4.1 Laser widget

There are two different ways we normally use the lasers, *offline* and triggered only by the buttons and sliders in this widget, or trigered by an acquisition card controlled by the scanning widget. In the latter case we press the *Digital Modulation* button and set the desired powers during the scan.

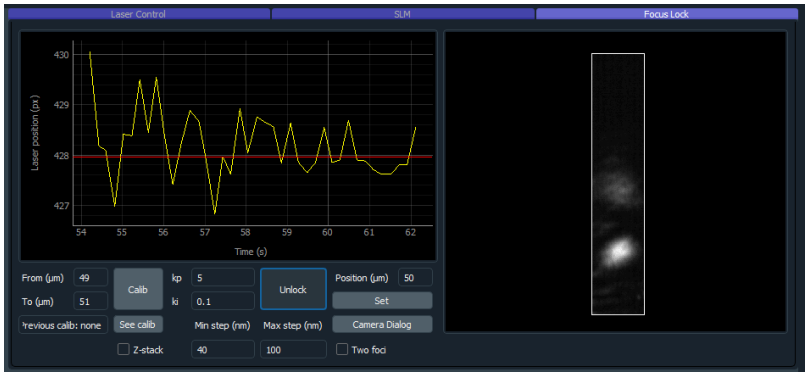


4.4.2 SLM widget

In the SLM widget you can control the phase masks which you use to shape the laser line that is incident on it. The SLM widget is configured to control two simultaneous phase masks applied on a beam, such as for shaping a STED laser beam into an overlayed donut and tophat pattern for 3DSTED, but can readily be reprogrammed to deal with other beam shaping for different methods. Through the widget you can control what type of mask you want to show in each of the two sides (donut, tophat, gaussian, half/quad/hex/split patterns for alignment purposes), the position of the masks, and their respective Zernike-polynomial-based aberration correction parameters for correcting stationary aberrations in the setup (implemented are tip/tilt, defocus, spherical, vertical/horizontal coma, and vertical/oblique astigmatism, additional polynomials can be readily implemented). It also has controls for saving/loading all the parameters to/from a pickled file.

4.4.3 Focus lock widget

In the focus lock widget you can control a reflection-based focus lock which operates by reflecting a laser beam off the cover slip in total internal reflection and is detected on a camera. Movement of the sample in z corresponds to lateral movement of the laser spot on the camera. The center of the spot is tracked and through a feedback loop (PI controller) commands is sent to the connect z-positioner to move the sample to counter-act the detected movement. The widget has controls for locking/unlocking the sample in the current position, setting the z-position of the connected z-positioner, a setting for handling double reflections from the sample, and settings for the proportional and integral gain of the PI controller.

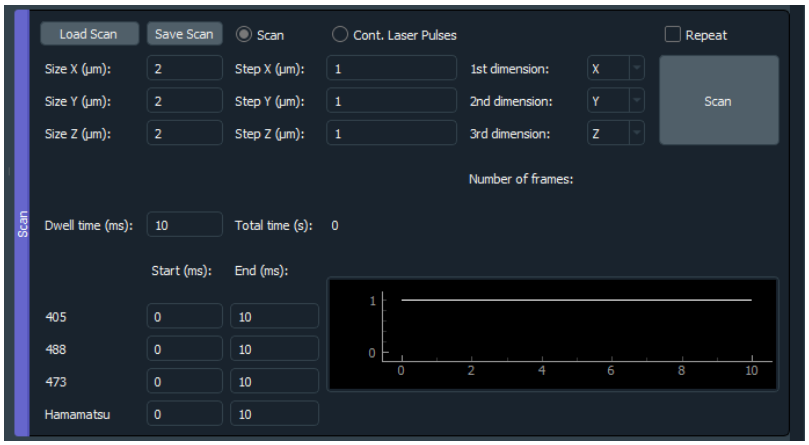


4.4.4 Positioner widget

For positioner we mean any type of scanning device that we wish to move either during a scan or by using this interface. The widget shows the current position of the positioners used in this interface, and has controls for moving them a set step size. The scripting module will also have access to these functions for automation applications.

4.4.5 Scanning widget

This module is designed for systems that need scanning for acquisition of an image. We have implemented it to be used with a Nidaq card, but it can also be generalized to other DAQs. In the config file the user specifies the analog/digital lines to which the instruments are connected, and the ScanDesigner and SignalDesigner will create the analog/digital signals to send to them for scanning. Specific modalities can implement their own version of the designers, since they are abstract classes.

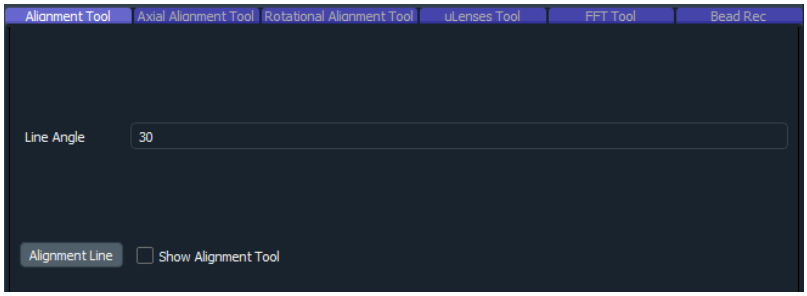


4.5 Alignment tools

The Alignment tools are a set of widgets that we use in the lab for aligning the MoNaLISA microscope. They do not control any hardware but instead perform operations on the images that provide easy feedback on the alignment process. They can be easily hidden or added by listing them in the configuration file. The general idea is that new tools can be implemented for different microscopy modalities and added to the library.

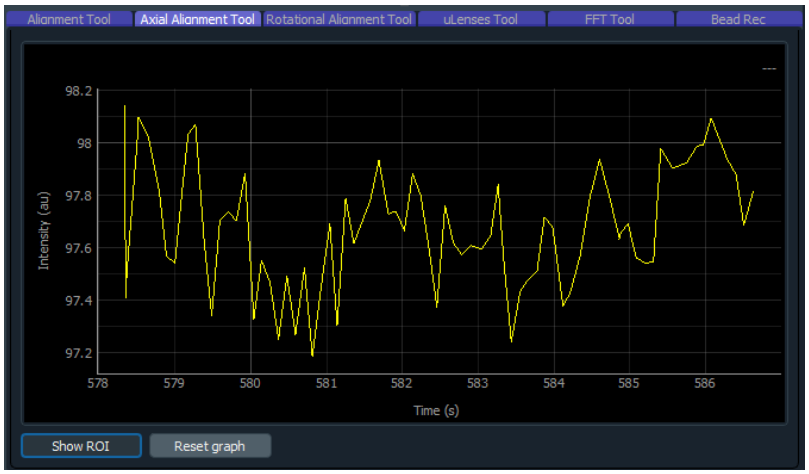
4.5.1 Alignment line

Displays a line with a certain angle on top of the images.



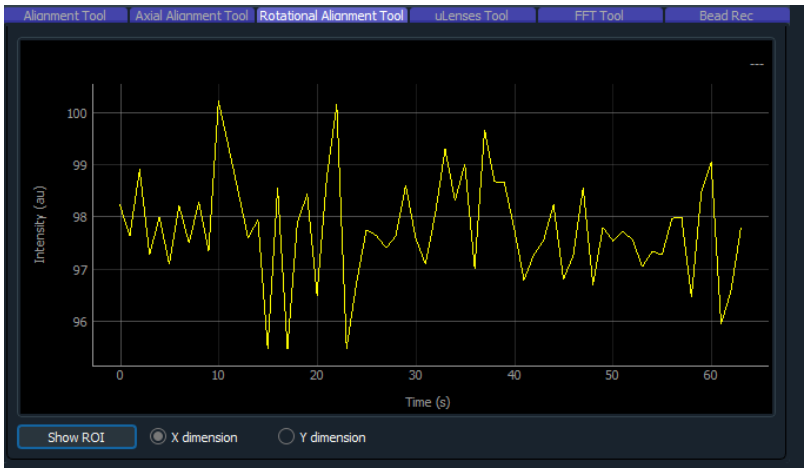
4.5.2 Axial alignment tool

The user selects a ROI and this tool will plot the mean value over time.



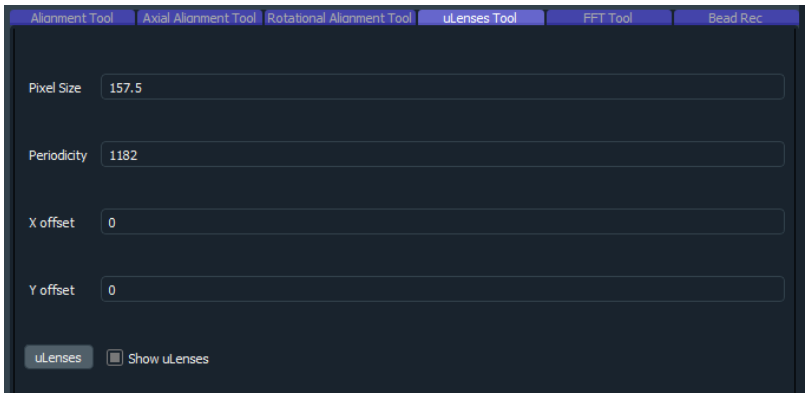
4.5.3 Rotational alignment tool

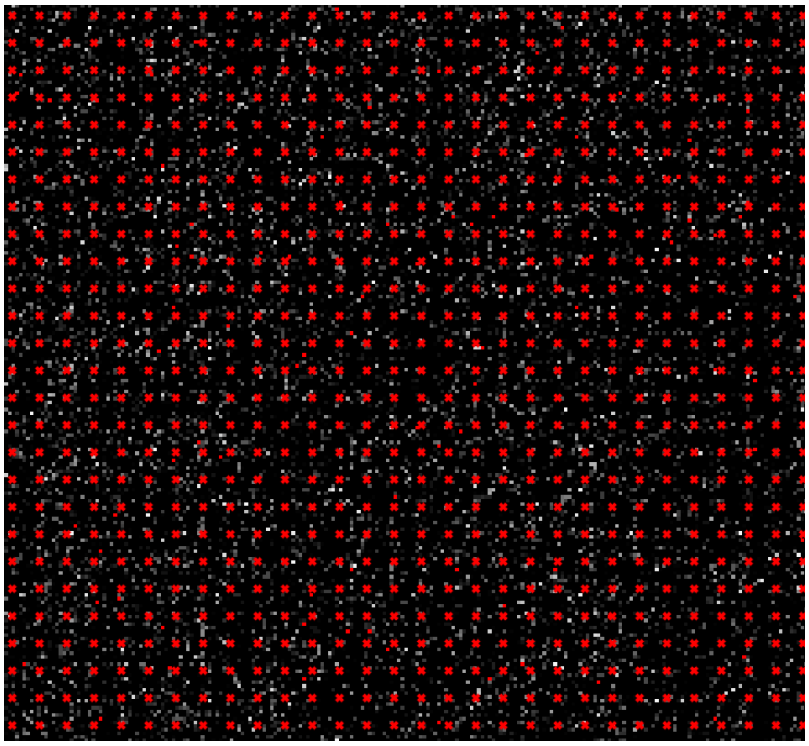
Similar as before but only over one axis (x or y).



4.5.4 uLenses tool

Will display an array of points with a certain periodicity in the image.



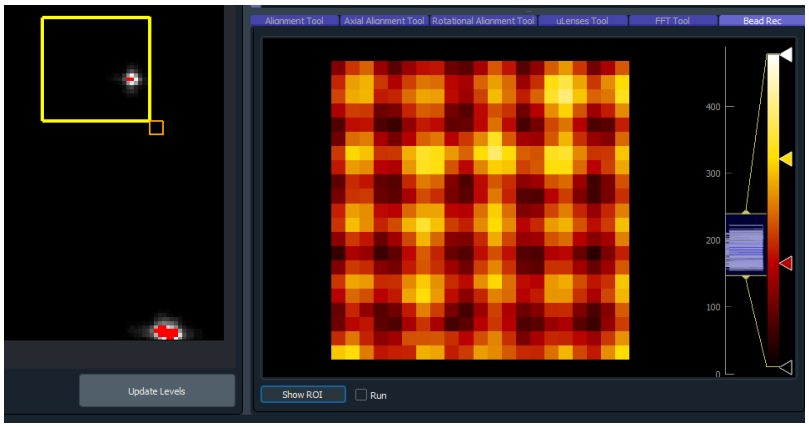


4.5.5 FFT tool

Performs the fourier transform of the incoming images in real time.

4.5.6 Bead rec tool

During a scan, this tool will integrate and reconstruct an image given a beadscan. Each step of the scan represents one pixel.

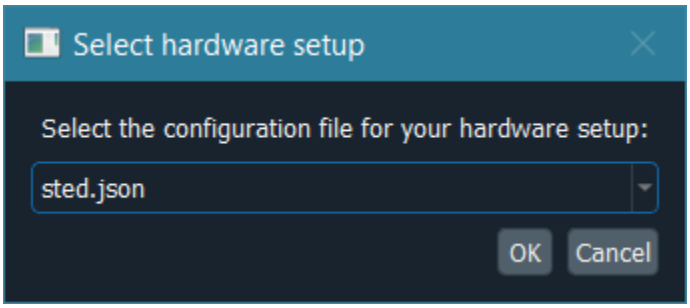


CHAPTER

FIVE

USE CASES

The first time the hardware control module is initialized, it will show a dialog to choose the hardware setup to be loaded. The user can change the setup option during execution in "Tools" -> "Pick hardware setup…" in the hardware control module's menu bar.



5.1 Parallelized confocal and RESOLFT (MoNaLISA)

Here we explain how we implemented ImSwitch for MoNaLISA. In the article, you will find more information about the setup and how the data is reconstructed.

5.1.1 Configuration file and hardware specifications

For this microscope use case, we created the JSON file `example_monalisa.json`, located at `/imswitch/_data/user_defaults/imcontrol_setups/example_monalisa.json`

We chose a National Instruments Data Acquisition (NIDAQ) card for managing the synchronization of the devices.

In the JSON file, two cameras are specified for two-color imaging: Green and Red. Both cameras are Hamamatsu, so they use `HamamatsuManager`. All the required camera properties are defined there, like the DAQ digital line for external triggering, readout speed, exposure time, field of view, etc.

There are five lasers in this setup, we use acousto-optic modulators (AOM) connected to the DAQ to control some of them, and others need the vendor interface as well, in this case Cobolt. The specific manager is defined for each type, `LantzLaserManager` or `NidaqLaserManager`, (see Hardware Control Configuration).

We use a X-Y-Z stage that we control through the DAQ as well, so the axes are defined as positioners using `NidaqPositionerManager`. The analog lines and conversion factors are specified as well. The modules that will create the signals for the scan are `BetaStageScanDesigner` for the Stage, and `BetaTTLCycleDesigner` for the instrument synchronization.

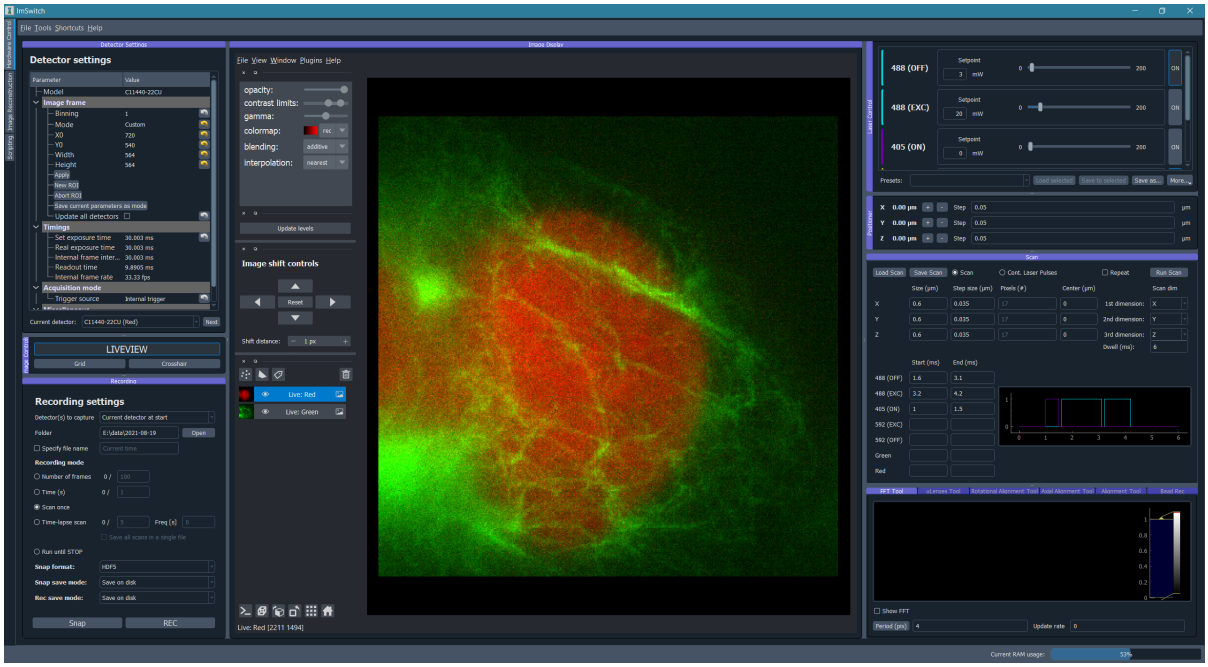
Other config parameters related to scanning, regions of interest (ROI) and a list of widgets to be loaded are added in this file.

5.1.2 Hardware control module

This module is useful to control the hardware and screen the sample using widefield or our other patterns. We have provided a more detailed explanation of the GUI *here*. To record a super-resolution image the user sets the camera to external-trigger mode and inserts the scan pulse scheme. The scanning module synchronizes the different instruments through the DAQ, and the raw data is displayed in the liveview.

The user can choose to save the raw data either in disk (hdf5) or RAM (or both) using the Recorder widget. So, for example, we program our scan and then click "Scan Once" in REC to start our acquisition. The metadata is also saved in the hdf5 and can be reloaded from the toolbar. It contains all the scanning pulses and hardware parameters related to the experiment.

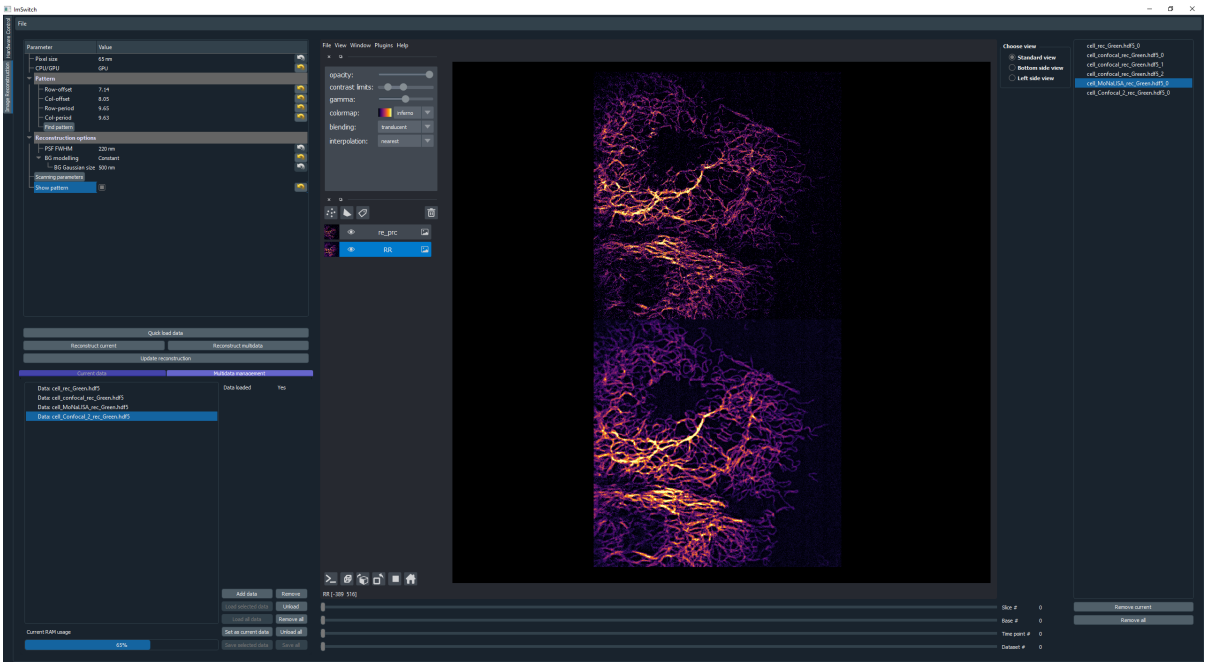
- GUI while using two-color widefield:



5.1.3 Image processing module for image reconstruction

The raw data can be either manually loaded into the reconstruction module or automatically retreived from the scanning if selected in the Recording widget. The user can further analyze the data using Napari image viewer. In this module we use our custom-designed DLLs for reconstruction, since this is a rather specific type of algorithm for our method. But the idea is that different microscope techniques implement their own modules as well. "Multidata management" stacks all the data incomming from the hardware control module.

- The Image processing module is illustrated in the following image:



5.2 Point-scanning confocal and STED

Here we explain how we implemented ImSwitch for a custom-built STED setup in the lab, previously controlled by a combination of closed-source software (image acquisition) and purpose-built software (hardware control). In the article, you will find more information about the setup, what hardware it contains, and the type of image acquisition we want to perform.

5.2.1 Configuration file and hardware specifications

For this microscope use case, we created the JSON file `example_sted.json`, located at `/imswitch/_data/user_defaults/imcontrol_setups/example_sted.json`

We chose a National Instruments Data Acquisition (NIDAQ) card for managing the synchronization of the devices and image acquisition.

In the JSON file, two photon-counting point detectors (APD) are specified for two-color imaging: APDGreen and APDRed. These do not need any specific hardware control, but instead are read entirely through the Nidaq. Additionally two cameras are specified: one for widefield, for having an overview of the sample, and one for the focus lock, as described in detail in the cited article. Both cameras are The Imaging Source cameras, so they use `TISManager`. All the required camera properties are defined, like the camera index in the list of cameras, exposure, gain, brightness, and chip size in pixels.

There are three lasers in this setup, and all three have an associated AOM or AOTF to rapidly control the power, and hence there are six laser devices defined. Two of them controls only fast digital modulation through digital Nidaq lines (561 and 640 lasers); one controls fast digital modulation and analog modulation through digital and analog Nidaq lines (775AOM); one controls the 775 nm laser through RS232 communication and hence has an associated rs232device (775Katana); and the last two controls the power modulation of the multiple channels of the common AOTF for the 561 and 640 nm lasers through RS232 communication with an associated rs232device (561AOTF and 640AOTF). The speicfic manager is defined for each device, `NidaqLaserManager`, `AAAOTFLaserManager`, or `KatanaLaserManager`.

We use galvanometric mirros for the XY-scanning that we control through the DAQ, so the axes are defined as positioners using `NidaqPositionerManager`. The analog lines of the Nidaq used and conversion factors, for converting µm

of the user-input to V for the signal, are specificied as well. Additionally a piezo is used for Z-movement, controlled both through analog signals from the DAQ with a `NidaqPositionerManager` and through RS232 communication with a `PiezoconceptZManager`.

The modules that will create the signals for the scan are `GalvoScanDesigner` for the XY-scanning, and `PointScanTTLCycleDesigner` for the laser synchronization. The analog scan designer will create smooth scanning signals with linear acquisition regions for good control of the galvanometric mirrors. The TTL designer will create laser modulation signals that can be controlled on a sub-line level with the widget interface, with automatic turn off during the portions of the scan that are not during acquisition.

The Hamamtsu SLM used in the setup is managed through the `SLMManager`, and is simply controlled by connecting it as a monitor and showing a gray-scale image with the pixel values corresponding to the phase-shift you want to impose. The manager is responsible for building this image based on the user-input from the widget.

The focus lock does not have a separate manager, but instead is associated with one of the TIS cameras and the Z-piezo rs232device. The properties for the focus lock specifies what hardware devices it should associate with, what part of the camera frame should be cropped, and the update frequency (in Hz) of the PI control loop.

The RS232 communication channel protocol parameters necessary for the control of the hardware devices requiring so are also defined in the same file.

Other config parameters and a list of widgets to be loaded are added in this file as well.

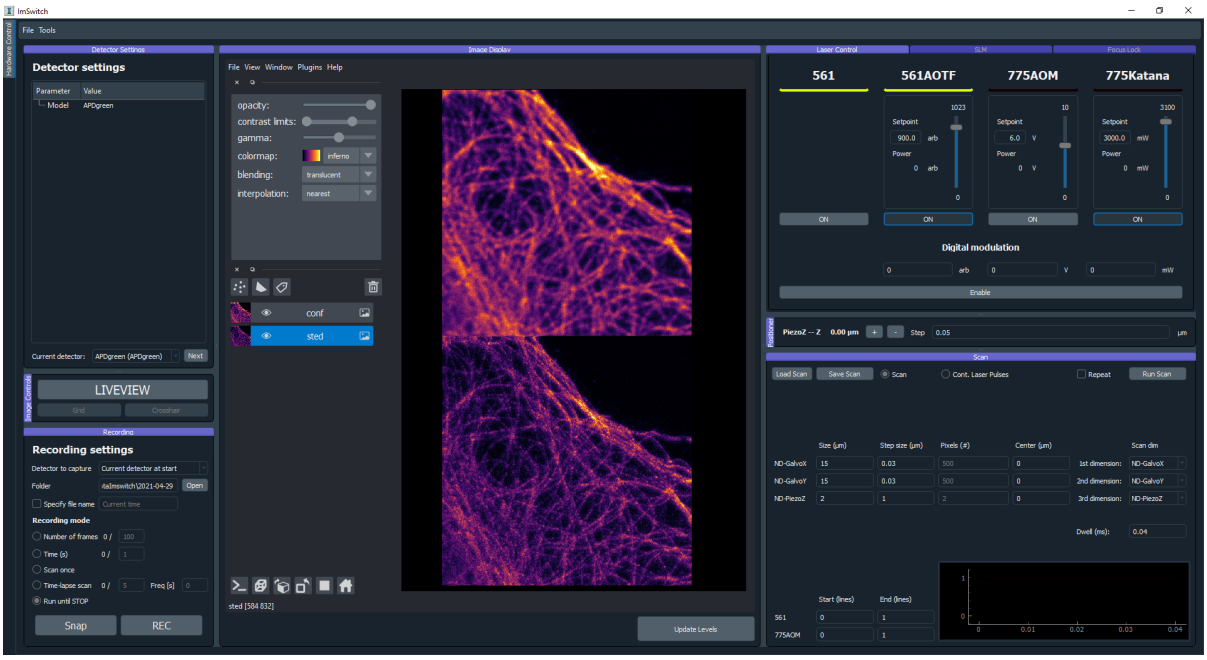
5.2.2 Main module

The main, and only, module for this use case is used to control all the hardware, screen the sample with widefield, acquiring the images, and inspecting them with the visualization tools. We have provided a more detailed explanation of the GUI *here*. To record a confocal image, the user sets the scan parameters that they want for each scan axis (length, pixel size, center position), the pixel dwell time, sets the laser powers they want to use, set the TTL start to 0 and end to 1 (units is lines) for the excitation laser they want to use, and runs the scan. The view of the detectors not in use can be hidden in the visualization tool. The scanning module will build the scanning curves, laser modulation curves, create those tasks in the Nidaq, and start them. The raw data is displayed in the liveview, where the image is updated line-by-line during the acquisition. For recording a STED image the procedure is much the same, with the addition that the use turns on the STED laser in the laser module, and sets the corresponding TTL start and end to the same values, and runs the scan. Before this the SLM has to be configured in order to create a desired depletion pattern, where for using a donut and tophat there are helpful tools in the SLM module to align the mask and the aberration correction that will be specific to each setup.

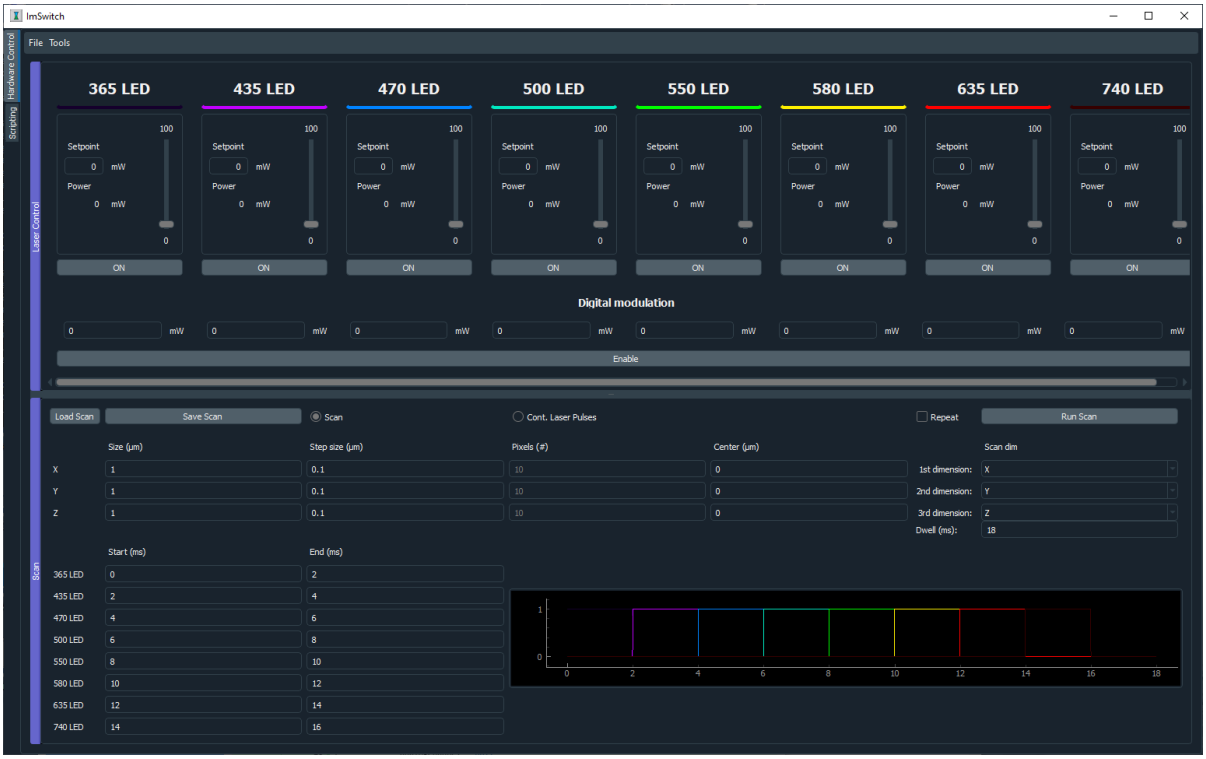
Previous to any image acquisition, while using either a repeating fast confocal scan or a widefield image, the sample has to be set in focus, and the focus lock can then be used to lock the sample in the focal plane. The focus lock acts independent from the image acquisition and can be continuously turned on for as long as wanted.

The user can choose to save the acquired image to a desired folder and with a desired name by using the Snap button in the recording widget. It will be saved in hdf5 format, and will include all user-defined parameters from the GUI as metadata. Functionality to reload metadata parameters from a previously saved hdf5 file can be found in the toolbar, for easy and precise recreation of a previous experiment. Previously recorded images in tiff format can also be loaded in the visualization module in order to be directly compared with the last recorded image or each other.

- GUI after having acquired a confocal and a STED image:



5.3 CoolLED control through USB and TTLs using a NIDAQ



We got a CoolLED (https://www.coolled.com/) in the lab and decided to try ImSwitch out in a setting where we want to control the 8 lasers of the device, both by doing it manually using the sliders and buttons (using a USB port and RS232 communication protocol), but also being able to design and perform a sequence of TTLs and a X-Y-Z Stage controlled by a National Instruments card. This use case could be combined with the Napari viewer and a camera, or

a point scanning system, or any of the other widgets explained in the other Use Cases.

All the lasers are listed in the JSON file `example_coolLED.json`, located at `/imswitch/_data/user_defaults/imcontrol_setups/example_coolLED.json`, by specifying:

- Digital line of each laser in the NIDAQ.
- Wavelength and range (0 to 100).
- Channel name (A-H), each corresponding to the laser.

The `Positioners` define the stage axis with the settings, such as:

- Analog channel of the NIDAQ.
- Conversion factors.
- Min and Max voltages.
- Axis (X, Y, or Z).

Then, the `CoolLEDLaserManager` will communicate with the `RS232Manager` for sending the intensity and on/off commands. The parameters of the `RS232Manager` are the typical ones of a RS232 connection, such as:

- Port (Usually COMx).
- Encoding (ascii).
- Baudrate (57600).
- ByteSize (8)
- Parity (None)
- Stop bits (1)

The pulses will be directly handled by the National Instruments card and our TTLDesigner.

CHAPTER

SIX

SCRIPTING

ImSwitch provides a scripting module that can be used to automate tasks in the software. This scripting module allows you to freely write Python code that interacts with ImSwitch.

See the scripting API reference for more information about the available API modules and methods. Aside from the API modules, there are also some global-level functions that you can use, that are documented *here*.

The API modules may provide signals – events that can be bound to through e.g. the global getWaitForSignal scripting function.

There are a few example scripts that you can check out in the scripting module to see how the scripting functionality works in action.

CHAPTER

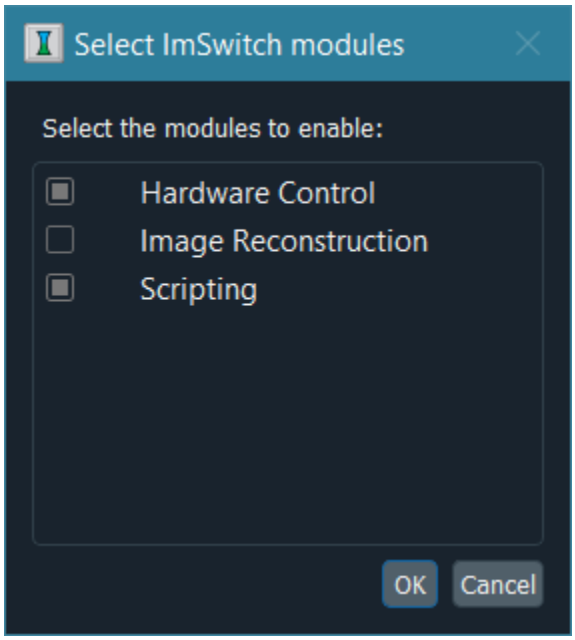
SEVEN

MODULE CONFIGURATION

ImSwitch consists of multiple different modules:

Name	ID
Hardware Control	`imcontrol`
Image Reconstruction	`imreconstruct`
Scripting	`imscripting`

One can select which modules to enable through the "Set active modules…" option in the "Tools" menu of the menu bar. By default, the hardware control and scripting modules are enabled. Modules that are not enabled will not show up in the program.



CHAPTER

EIGHT

HDF5 DATAFILES

In ImSwitch we use HDF5 files to store the images and metadata containing the experiment's parameters. The HDF5 files can be opened in for example ImageJ and Matlab with the right extensions. For the metadata, HDFView can display the attributes and datasets of the file.

It is possible to import the experiment parameters in ImSwitch from the File menu (File -> Load parameters from saved HDF5 file...), and the GUI will load and display all the parameters directly in the widgets. When recording multiple detectors simultaneously, one file will be created for each recorded detector.

8.1 Datasets

Each image recording is saved in a dataset with dimensions Z × Y × X, where Z is the number of frames while Y and X are the vertical and horizontal axes respectively. Two special parameters are stored in the dataset:

- `detector_name`: name of the detector (camera or point-detector) that provided the images.
- `element_size_um`: pixel size of the image, this parameter will be automatically read by ImageJ when opening the file.

8.2 Object attributes

The rest of the metadata is also stored as HDF5 attributes in the datasets, containing information about the detectors, lasers, recording, and scanning.

8.2.1 Detectors

There is one attribute for each detector property that is listed in the DetectorManager being used. For example, for HamamatsuManager: Binning, model, camera pixel size, readout time, ROI, etc.

The detector's attributes follow the form:

- `Detector:NameDetector:DetectorProperty`

8.2.2 Lasers

The power and whether it was ON/OFF for each laser is stored in the form:

- `Laser:LaserName:Enabled` (boolean)
- `Laser:LaserName:Value`

8.2.3 Positioners

The value for each positioner is stored to encode in which area the image was taken:

- `Positioner:PostionerName:PostionerAxis:Position`

8.2.4 Recording and scanning

The parameters of the recording and scanning are attributes as well. They include all the parameters in the RecordingWidget and ScanWidget, regarding the pulse scheme, the stage positions and step sizes, the type of recording, number of frames, etc. It can vary depending on the setup used, but they generally follow the form shown below:

- `Rec:PropertyName`
- `ScanStage:PropertyName`
- `ScanTTL:PropertyName`

CHAPTER

NINE

HARDWARE CONTROL CONFIGURATIONS

ImSwitch's hardware control module is designed to be flexible and be usable in a wide variety of microscopy setups. In order to provide this flexibility, hardware configurations are defined in .json files that are loaded when the hardware control module starts.

Hardware configuration files are loaded from the `imcontrol_setups` directory, which is automatically created inside your user directory for ImSwitch the first time the hardware control module starts. It contains some pre-made configuration files by default. The user directory is located at `%USERPROFILE%\Documents\ImSwitch` on Windows and `~/ImSwitch` on macOS/Linux.

The first time you start the hardware control module, you will be prompted to select a setup file to load. If you want to switch to another hardware configuration later, select "Tools" -> "Pick hardware setup…" in the hardware control module's menu bar.

9.1 How configurations are defined

Hardware configurations are defined in JSON format. Behind the scenes, they are automatically translated to Python class instances when loaded into the software.

A central concept in ImSwitch is that of device managers. Device managers define what kind of device you have, and how ImSwitch communicates with it. For example, if you have a Hamamatsu camera that you would like to control, you would define a detector that uses the `HamamatsuManager` in the hardware setup file and set its appropriate properties. The list of available managers and their properties can be found *here*. Each device must have a unique name, which is represented by its object key in the JSON.

Signal designers, which are relevant for users who use the scan functionality, are similar. Microscopy scans can be set up in different ways; in a point-scanning setup, for instance, you might want to set your scan settings to use the `PointScanTTLCycleDesigner` to generate the appropriate TTL signals. They are documented *here*.

As a very simple example, a hardware configuration file that allows you to control a single Cobolt 06-01 (non-DPL) laser connected to COM port 11 can look like this:

```
{
    "lasers": {
        "Cobolt405nm": {
            "managerName": "Cobolt0601LaserManager",
            "managerProperties": {
                "digitalPorts": ["COM11"]
            },
            "valueRangeMin": 0,
            "valueRangeMax": 200,
            "wavelength": 405
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "availableWidgets": [
        "Laser"
    ]
}
```

Note that the `digitalPorts` property is specific to `Cobolt0601LaserManager`.

9.2 Configuration file specification

class ViewSetupInfo

This is the object represented by the hardware configuration JSON file. All fields are optional, unless explicitly otherwise specified.

availableWidgets: Union[List[str], bool]

Which widgets to load. The following values are possible to include (case sensitive):

- `Settings` (detector settings widget)
- `View` (image controls widget)
- `Recording` (recording widget)
- `Image` (image display widget)
- `FocusLock` (focus lock widget; requires `focusLock` field to be defined)
- `SLM` (SLM widget; requires `slm` field to be defined)
- `Laser` (laser control widget)
- `Positioner` (positioners widget)
- `Scan` (scan widget; requires `scan` field to be defined)
- `BeadRec` (bead reconstruction widget)
- `AlignAverage` (axial alignment tool widget)
- `AlignXY` (rotation alignment tool widget)
- `AlignmentLine` (line alignment tool widget)
- `uLenses` (uLenses tool widget; requires `Image` widget)
- `FFT` (FFT tool widget)
- `Console` (Python console widget)

You can also set this to `true` to enable all widgets, or `false` to disable all widgets.

This field is required.

defaultLaserPresetForScan: Optional[str]

Default laser preset for scanning.

detectors: Dict[str, *DetectorInfo*]

Detectors in this setup. This is a map from unique detector names to DetectorInfo objects.

focusLock: Optional[*FocusLockInfo*]

Focus lock settings. Required to be defined to use focus lock functionality.

laserPresets: Dict[str, Dict[str, *LaserPresetInfo*]]

Laser presets available to select (map preset name -> laser name -> LaserPresetInfo).

lasers: Dict[str, *LaserInfo*]

Lasers in this setup. This is a map from unique laser names to LaserInfo objects.

nidaq: *NidaqInfo*

NI-DAQ settings.

positioners: Dict[str, *PositionerInfo*]

Positioners in this setup. This is a map from unique positioner names to DetectorInfo objects.

rois: Dict[str, *ROIInfo*]

Additional ROIs available to select in detector settings.

rs232devices: Dict[str, *RS232Info*]

RS232 connections in this setup. This is a map from unique RS232 connection names to RS232Info objects. Some detector/laser/positioner managers will require a corresponding RS232 connection to be referenced in their properties.

scan: Optional[*ScanInfo*]

Scan settings. Required to be defined to use scan functionality.

slm: Optional[*SLMInfo*]

SLM settings. Required to be defined to use SLM functionality.

9.3 Item types that may be included

class DetectorInfo

analogChannel: Optional[int]

Channel for analog communication. `null` if the device is digital or doesn't use NI-DAQ.

digitalLine: Optional[int]

Line for digital communication. `null` if the device is analog or doesn't use NI-DAQ.

forAcquisition: bool = False

Whether the detector is used for acquisition.

forFocusLock: bool = False

Whether the detector is used for focus lock.

managerName: str

Manager class name.

managerProperties: Dict[str, Any]

Properties to be read by the manager.

class LaserInfo

analogChannel: Optional[int]

Channel for analog communication. `null` if the device is digital or doesn't use NI-DAQ.

digitalLine: Optional[int]

Line for digital communication. null if the device is analog or doesn't use NI-DAQ.

managerName: str

Manager class name.

managerProperties: Dict[str, Any]

Properties to be read by the manager.

valueRangeMax: Optional[Union[int, float]]

maximum value of the laser. null if laser doesn't setting a value.

valueRangeMin: Optional[Union[int, float]]

Minimum value of the laser. null if laser doesn't setting a value.

valueRangeStep: float = 1.0

The default step size of the value range that the laser can be set to.

wavelength: Union[int, float]

Laser wavelength in nanometres.

class PositionerInfo

analogChannel: Optional[int]

Channel for analog communication. null if the device is digital or doesn't use NI-DAQ.

axes: List[str]

A list of axes (names) that the positioner controls.

digitalLine: Optional[int]

Line for digital communication. null if the device is analog or doesn't use NI-DAQ.

forPositioning: bool = False

Whether the positioner is used for manual positioning.

forScanning: bool = False

Whether the positioner is used for scanning.

isPositiveDirection: bool = True

Whether the direction of the positioner is positive.

managerName: str

Manager class name.

managerProperties: Dict[str, Any]

Properties to be read by the manager.

class RS232Info

managerName: str

RS232 manager class name.

managerProperties: Dict[str, Any]

Properties to be read by the RS232 manager.

class SLMInfo

angleMount: float

The angle of incidence and reflection of the laser line that is shaped by the SLM, in radians. For adding a blazed grating to create off-axis holography.

correctionPatternsDir: str

Directory of .bmp images provided by Hamamatsu for flatness correction at various wavelengths. A combination will be chosen based on the wavelength.

height: int

Height of SLM, in pixels.

monitorIdx: int

Index of the monitor in the system list of monitors (indexing starts at 0).

pixelSize: float

Pixel size or pixel pitch of the SLM, in millimetres.

wavelength: int

Wavelength of the laser line used with the SLM.

width: int

Width of SLM, in pixels.

class FocusLockInfo

camera: str

Detector name.

frameCroph: int

Height of frame crop.

frameCropw: int

Width of frame crop.

frameCropx: int

Starting X position of frame crop.

frameCropy: int

Starting Y position of frame crop.

positioner: str

Positioner name.

updateFreq: int

Update frequency, in milliseconds.

class ScanInfo

TTLCycleDesigner: str

Name of the TTL cycle designer class to use.

TTLCycleDesignerParams: Dict[str, Any]

Params to be read by the TTL cycle designer.

sampleRate: int

Scan sample rate.

scanDesigner: str

Name of the scan designer class to use.

scanDesignerParams: Dict[str, Any]

Params to be read by the scan designer.

class NidaqInfo

startTrigger: bool = False

Boolean for start triggering for sync.

timerCounterChannel: Optional[int] = None

Output for Counter for timing purposes.

class ROIInfo

h: int

Height of ROI, in pixels.

w: int

Width of ROI, in pixels.

x: int

Starting X position of ROI, in pixels.

y: int

Starting Y position of ROI, in pixels.

class LaserPresetInfo

value: float

Laser value.

9.4 Available managers

9.4.1 Detector managers

class APDManager

DetectorManager that deals with an avalanche photodiode connected to a counter input on a Nidaq card.

Manager properties:

- `terminal` – the physical input terminal on the Nidaq to which the APD is connected
- `ctrInputLine` – the counter that the physical input terminal is connected to

class HamamatsuManager

DetectorManager that deals with the Hamamatsu parameters and frame extraction for a Hamamatsu camera.

Manager properties:

- `cameraListIndex` – the camera's index in the Hamamatsu camera list (list indexing starts at 0); set this to an invalid value, e.g. the string "mock" to load a mocker
- `hamamatsu` – dictionary of DCAM API properties to pass to the driver

class PhotometricsManager

DetectorManager that deals with frame extraction for a Photometrics camera.

Manager properties:

- `cameraListIndex` – the camera's index in the Photometrics camera list (list indexing starts at 0)

class TISManager

DetectorManager that deals with TheImagingSource cameras and the parameters for frame extraction from them.

Manager properties:

- `cameraListIndex` – the camera's index in the TIS camera list (list indexing starts at 0); set this string to an invalid value, e.g. the string "mock" to load a mocker
- `tis` – dictionary of TIS camera properties

9.4.2 Laser managers

class AAAOTFLaserManager

LaserManager for controlling one channel of an AA Opto-Electronic acousto-optic modulator/tunable filter through RS232 communication.

Manager properties:

- `rs232device` – name of the defined rs232 communication channel through which the communication should take place
- `channel` – index of the channel in the acousto-optic device that should be controlled (indexing starts at 1)

class CoolLEDLaserManager

LaserManager for controlling the LEDs from CoolLED. Each LaserManager instance controls one LED.

Manager properties:

- `rs232device` – name of the defined rs232 communication channel through which the communication should take place
- `channel_index` – laser channel (A to H)

class NidaqLaserManager

LaserManager for analog-value NI-DAQ-controlled lasers.

Manager properties: None

9.4.3 Positioner managers

class MHXYStageManager

PositionerManager for control of a Marzhauser XY-stage through RS232 communication.

Manager properties:

- `rs232device` – name of the defined rs232 communication channel through which the communication should take place

class NidaqPositionerManager

PositionerManager for analog-value NI-DAQ-controlled positioners.

Manager properties:

- `conversionFactor` – float value
- `minVolt` – minimum voltage
- `maxVolt` – maximum voltage

class PiezoconceptZManager

PositionerManager for control of a Piezoconcept Z-piezo through RS232 communication.

Manager properties:

- `rs232device` – name of the defined rs232 communication channel through which the communication should take place

9.4.4 RS232 managers

class RS232Manager

A general-purpose RS232 manager that together with a general-purpose RS232Driver interface can handle an arbitrary RS232 communication channel, with all the standard serial communication protocol parameters as defined in the hardware control configuration.

Manager properties:

- `port`
- `encoding`
- `recv_termination`
- `send_termination`
- `baudrate`
- `bytesize`
- `parity`
- `stopbits`
- `rtscts`
- `dsrdtr`
- `xonxoff`

9.5 Available signal designers

9.5.1 Scan designers

class BetaScanDesigner

Scan designer for X/Y/Z stages that move a sample.

Designer params:

- `return_time` – time to wait between lines for the stage to return to the first position of the next line, in seconds.

class GalvoScanDesigner

Scan designer for scan systems with galvanometric mirrors.

Designer params: None

9.5.2 TTL cycle designers

class BetaTTLCycleDesigner

TTL cycle designer for camera-based applications where each pulse scheme is one frame.

Designer params: None

class PointScanTTLCycleDesigner

Line-based TTL cycle designer, for point-scanning applications. Treats input ms as lines.

Designer params: None

CHAPTER

TEN

ADDING SUPPORT FOR MORE DEVICES

There are three main device types that ImSwitch's hardware control module supports: **detectors**, **lasers** and **positioners**. In order to add support for a new detector, laser or positioner, a corresponding device manager class must be implemented in ImSwitch's code.

10.1 How device managers are implemented

Detector support is implemented in device manager classes derived from the abstract base class `DetectorManager`. The corresponding parent class for lasers is `LaserManager`, and for positioners it is `PositionerManager`. These derived classes are placed in the `detectors`, `lasers` and `positioners` sub-modules respectively in the `imswitch.imcontrol.model.managers` module.

The required constructor signature for the device managers is `__init__(deviceInfo, name, **lowLevelManagers)`. `deviceInfo` is the `DetectorInfo`, `LaserInfo` or `PositionerInfo` object which represents the device's entry in the setup file (see *the hardware control setup page* for further information). Inside it, the `managerProperties` dict field may contain manager-specific properties. `name` is a unique name that is used to identify the device, which is defined by the key of the device's entry in the setup file. `lowLevelManagers` is a dict containing objects that facilitate low-level device interaction, which are documented *here*. Note that `super().__init__` has a different signature, depending on which base class is used.

When creating a new device manager, you will need to implement all the abstract methods and properties defined in the base class. You should avoid overriding non-abstract properties. Overriding non-abstract methods is generally fine, but you should make sure that they continue to work as expected. The device manager class must be placed in a .py file with the same name as the class, in the appropriate location as outlined above. No other action is required for the device manager to be available to use; it will automatically be managed by a multi-manager as outlined in the paper.

You can find a simple example of a positioner manager implementation here.

10.2 Base class documentation

10.2.1 DetectorManager

class imswitch.imcontrol.model.managers.detectors.DetectorManager.**DetectorManager**(*detectorInfo*, *name: str*, *fullShape: Tuple[int, int]*, *supportedBinnings: List[int]*, *model: str*, ***, *parameters: Optional[Dict[str, DetectorParameter]] = None*, *actions: Optional[Dict[str, DetectorAction]] = None*, *croppable: bool = True*)

Abstract base class for managers that control detectors. Each type of detector corresponds to a manager derived from this class.

abstract __init__(*detectorInfo*, *name: str*, *fullShape: Tuple[int, int]*, *supportedBinnings: List[int]*, *model: str*, ***, *parameters: Optional[Dict[str, DetectorParameter]] = None*, *actions: Optional[Dict[str, DetectorAction]] = None*, *croppable: bool = True*) → None

Parameters

- **detectorInfo** – See setup file documentation.
- **name** – The unique name that the device is identified with in the setup file.
- **fullShape** – Maximum image size as a tuple `(width, height)`.
- **supportedBinnings** – Supported binnings as a list.
- **model** – Detector device model name.
- **parameters** – Parameters to make available to the user to view/edit.
- **actions** – Actions to make available to the user to execute.
- **croppable** – Whether the detector image can be cropped.

property actions: Dict[str, *DetectorAction*]

Dictionary of available actions.

property binning: int

Current binning.

abstract crop(*hpos: int*, *vpos: int*, *hsize: int*, *vsize: int*) → None

Crop the frame read out by the detector.

property croppable: bool

Whether the detector supports frame cropping.

finalize() → None

Close/cleanup detector.

abstract flushBuffers() → None

Flushes the detector buffers so that getChunk starts at the last frame captured at the time that this function was called.

property forAcquisition: bool

Whether the detector is used for acquisition.

property forFocusLock: bool

Whether the detector is used for focus lock.

property frameStart: Tuple[int, int]

Position of the top left corner of the current frame as a tuple `(x, y)`.

property fullShape: Tuple[int, int]

Maximum image size as a tuple `(width, height)`.

abstract getChunk() → ndarray

Returns the frames captured by the detector since getChunk was last called, or since the buffers were last flushed (whichever happened last). The returned object is a numpy array of shape (numFrames, height, width).

abstract getLatestFrame() → ndarray

Returns the frame that represents what the detector currently is capturing. The returned object is a numpy array of shape (height, width).

property image: ndarray

Latest LiveView image.

property model: str

Detector model name.

property name: str

Unique detector name, defined in the detector's setup info.

property parameters: Dict[str, *DetectorParameter*]

Dictionary of available parameters.

abstract property pixelSizeUm: List[int]

The pixel size in micrometers, in the format `[z, y, x]`. `z` is typically set to 1.

setBinning(*binning: int*) → None

Sets the detector's binning.

setParameter(*name: str*, *value: Any*) → Dict[str, *DetectorParameter*]

Sets a parameter value and returns the updated list of parameters. If the parameter doesn't exist, i.e. the parameters field doesn't contain a key with the specified parameter name, an AttributeError will be raised.

property shape: Tuple[int, int]

Current image size as a tuple `(width, height)`.

abstract startAcquisition() → None

Starts image acquisition.

abstract stopAcquisition() → None

Stops image acquisition.

property supportedBinnings: List[int]

Supported binnings as a list.

class imswitch.imcontrol.model.managers.detectors.DetectorManager.**DetectorAction**(*group: str*, *func: callable*)

An action that is made available for the user to execute.

func: callable

The function that is called when the action is executed.

group: str

The group to place the action in (does not need to be pre-defined).

class imswitch.imcontrol.model.managers.detectors.DetectorManager.**DetectorParameter**(*group: str*, *value: Any*, *editable: bool*)

Abstract base class for detector parameters that are made available for the user to view/edit.

class imswitch.imcontrol.model.managers.detectors.DetectorManager.**DetectorNumberParameter**(*group: str*, *value: float*, *editable: bool*, *valueUnits: str*)

Bases: *DetectorParameter*

A detector parameter with a numerical value.

editable: bool

Whether it is possible to edit the value of the parameter.

group: str

The group to place the parameter in (does not need to be pre-defined).

value: float

The value of the parameter.

valueUnits: str

Parameter value units, e.g. “nm” or “fps”.

class imswitch.imcontrol.model.managers.detectors.DetectorManager.**DetectorListParameter**(*group: str*, *value: str*, *editable: bool*, *options: List[str]*)

Bases: *DetectorParameter*

A detector parameter with a value from a list of options.

editable: bool
Whether it is possible to edit the value of the parameter.

group: str
The group to place the parameter in (does not need to be pre-defined).

options: List[str]
The available values to pick from.

value: str
The value of the parameter.

10.2.2 LaserManager

class imswitch.imcontrol.model.managers.lasers.LaserManager.**LaserManager**(*laserInfo*, *name: str*, *isBinary: bool*, *valueUnits: str*, *valueDecimals: int*)

Abstract base class for managers that control lasers. Each type of laser corresponds to a manager derived from this class.

abstract __init__(*laserInfo*, *name: str*, *isBinary: bool*, *valueUnits: str*, *valueDecimals: int*) → None

Parameters

- **laserInfo** – See setup file documentation.
- **name** – The unique name that the device is identified with in the setup file.
- **isBinary** – Whether the laser can only be turned on and off, and its value cannot be changed.
- **valueUnits** – The units of the laser value, e.g. "mW" or "V".
- **valueDecimals** – How many decimals are accepted in the laser value.

finalize() → None
Close/cleanup laser.

property isBinary: bool
Whether the laser can only be turned on and off, and its value cannot be changed.

property name: str
Unique laser name, defined in the laser's setup info.

abstract setEnabled(*enabled: bool*) → None

Sets whether the laser is enabled.

setScanModeActive(*active: bool*) → None

Sets whether the laser should be in scan mode (if the laser supports it).

abstract setValue(*value: Union[int, float]*) → None

Sets the value of the laser.

property valueDecimals

How many decimals are accepted in the laser value.

property valueRangeMax: float

The maximum value that the laser can be set to.

property valueRangeMin: float

The minimum value that the laser can be set to.

property valueRangeStep: float

The default step size of the value range that the laser can be set to.

property valueUnits: str

The units of the laser value, e.g. "mW" or "V".

property wavelength: int

The wavelength of the laser.

10.2.3 PositionerManager

class imswitch.imcontrol.model.managers.positioners.PositionerManager.**PositionerManager**(*positionerInfo*, *name: str*, *initialPosition: Dict[str, float]*)

Abstract base class for managers that control positioners. Each type of positioner corresponds to a manager derived from this class.

abstract __init__(*positionerInfo*, *name: str*, *initialPosition: Dict[str, float]*)

Parameters

- **positionerInfo** – See setup file documentation.
- **name** – The unique name that the device is identified with in the setup file.
- **initialPosition** – The initial position for each axis. This is a dict in the format `{ axis: position }`.

property axes: List[str]

The list of axes that are controlled by this positioner.

finalize() → None

Close/cleanup positioner.

property forPositioning: bool

Whether the positioner is used for manual positioning.

property forScanning: bool

Whether the positioner is used for scanning.

abstract move(*dist: float*, *axis: str*)

Moves the positioner by the specified distance and returns the new position. Derived classes will update the position field manually. If the positioner controls multiple axes, the axis must be specified.

property name: str

Unique positioner name, defined in the positioner's setup info.

property position: Dict[str, float]

The position of each axis. This is a dict in the format `{ axis: position }`.

abstract setPosition(*position: float*, *axis: str*)

Adjusts the positioner to the specified position and returns the new position. Derived classes will update the position field manually. If the positioner controls multiple axes, the axis must be specified.

10.3 Available low-level managers

10.3.1 lowLevelManagers['nidaqManager']

class imswitch.imcontrol.model.managers.NidaqManager.**NidaqManager**(*setupInfo*)

For interaction with NI-DAQ hardware interfaces.

setAnalog(*target*, *voltage*, *min_val=-1*, *max_val=1*)

Function to set the analog channel to a specific target to a certain voltage

setDigital(*target*, *enable*)

Function to set the digital line to a specific target to either "high" or "low" voltage

10.3.2 lowLevelManagers['rs232sManager']

class imswitch.imcontrol.model.managers.RS232sManager.**RS232sManager**(*rs232deviceInfos*, ***lowLevelManagers*)

RS232sManager is an interface for dealing with RS232 devices. It is a MultiManager for RS232 devices.

RS232Manager instances for individual RS232 devices can be accessed by `rs232sManager['your_rs232_device_name']`.

class imswitch.imcontrol.model.managers.rs232.RS232Manager.**RS232Manager**(*rs232Info*, *name*, ***_lowLevelManagers*)

A general-purpose RS232 manager that together with a general-purpose RS232Driver interface can handle an arbitrary RS232 communication channel, with all the standard serial communication protocol parameters as defined in the hardware control configuration.

Manager properties:

- port
- encoding
- recv_termination
- send_termination
- baudrate
- bytesize
- parity
- stopbits
- rtscts
- dsrdtr
- xonxoff

send(*arg: str*) → str

Sends the specified command to the RS232 device and returns a string encoded from the received bytes.

CHAPTER

ELEVEN

GLOBAL-LEVEL FUNCTIONS

getWaitForSignal(*self*, *signal: imswitch.imcommon.framework.qt.Signal*, *pollIntervalSeconds: float = 1.0*) → Callable[[], NoneType]

Returns a function that will wait for the specified signal to emit. The returned function will continuously check whether the signal has been emitted since its creation. The polling interval defaults to one second, and can be customized.

importScript(*self*, *path: str*) → Any

Imports the script at the specified path (either absolute or relative to the main script) and returns it as a module variable.

CHAPTER

TWELVE

API.IMCONTROL

class api.**imcontrol**

These functions are available in the api.imcontrol object.

getDetectorNames() → List[str]

Returns the device names of all detectors. These device names can be passed to other detector-related functions.

getLaserNames() → List[str]

Returns the device names of all lasers. These device names can be passed to other laser-related functions.

getPositionerNames() → List[str]

Returns the device names of all positioners. These device names can be passed to other positioner-related functions.

getPositionerPositions() → Dict[str, Dict[str, float]]

Returns the positions of all positioners.

loadScanParamsFromFile(*filePath: str*) → None

Loads scanning parameters from the specified file.

movePositioner(*positionerName: str*, *axis: str*, *dist: float*) → None

Moves the specified positioner axis by the specified number of micrometers.

runScan() → None

Runs a scan with the set scanning parameters.

saveScanParamsToFile(*filePath: str*) → None

Saves the set scanning parameters to the specified file.

setDetectorBinning(*detectorName: str*, *binning: int*) → None

Sets binning value for the specified detector.

setDetectorParameter(*detectorName: str*, *parameterName: str*, *value: Any*) → None

Sets the specified detector-specific parameter to the specified value.

setDetectorROI(*detectorName: str*, *frameStart: Tuple[int, int]*, *shape: Tuple[int, int]*) → None

Sets the ROI for the specified detector. frameStart is a tuple (x0, y0) and shape is a tuple (width, height).

setDetectorToRecord(*detectorName: Union[List[str], str, int]*, *multiDetectorSingleFile: bool = False*) → None

Sets which detectors to record. One can also pass -1 as the argument to record the current detector, or -2 to record all detectors.

setLaserActive(*laserName: str*, *active: bool*) → None

Sets whether the specified laser is powered on.

setLaserValue(*laserName: str*, *value: Union[int, float]*) → None

Sets the value of the specified laser, in the units that the laser uses.

setLiveViewActive(*active: bool*) → None

Sets whether the LiveView is active and updating.

setLiveViewCrosshairVisible(*visible: bool*) → None

Sets whether the LiveView crosshair is visible.

setLiveViewGridVisible(*visible: bool*) → None

Sets whether the LiveView grid is visible.

setPositioner(*positionerName: str*, *axis: str*, *position: float*) → None

Moves the specified positioner axis to the specified position.

setPositionerStepSize(*positionerName: str*, *stepSize: float*) → None

Sets the step size of the specified positioner to the specified number of micrometers.

setRecFilename(*filename: Optional[str]*) → None

Sets the name of the file to record to. This only sets the name of the file, not the full path. One can also pass None as the argument to use a default time-based filename.

setRecFolder(*folderPath: str*) → None

Sets the folder to save recordings into.

setRecModeScanOnce() → None

Sets the recording mode to record a single scan.

setRecModeScanTimelapse(*lapsesToRec: int*, *freqSeconds: float*, *timelapseSingleFile: bool = False*) → None

Sets the recording mode to record a timelapse of scans.

setRecModeSpecFrames(*numFrames: int*) → None

Sets the recording mode to record a specific number of frames.

setRecModeSpecTime(*secondsToRec: Union[int, float]*) → None

Sets the recording mode to record for a specific amount of time.

setRecModeUntilStop() → None

Sets the recording mode to record until recording is manually stopped.

signals() → Mapping[str, imswitch.imcommon.framework.qt.Signal]

Returns signals that can be used with e.g. the getWaitForSignal action. Currently available signals are:

- acquisitionStarted
- acquisitionStopped
- recordingStarted
- recordingEnded
- scanEnded

They can be accessed like this: api.imcontrol.signals().scanEnded

snapImage() → None

Take a snap and save it to a .tiff file at the set file path.

startRecording() → None

Starts recording with the set settings to the set file path.

stepPositionerDown(*positionerName: str*, *axis: str*) → None

Moves the specified positioner axis in negative direction by its set step size.

stepPositionerUp(*positionerName: str*, *axis: str*) → None

Moves the specified positioner axis in positive direction by its set step size.

stopRecording() → None

Stops recording.

CHAPTER

THIRTEEN

MAINWINDOW

class mainWindow

These functions are available in the mainWindow object.

setCurrentModule(*self*, *moduleId: str*) → None

Sets the currently displayed module to the module with the specified ID (e.g. “imcontrol”).

INDEX

Symbols

__init__() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 44
__init__() (*imswitch.imcontrol.model.managers.lasers.LaserManager.LaserManager method*), 47
__init__() (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager method*), 48

A

AAAOTFLaserManager (*built-in class*), 39
actions (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 44
analogChannel (*imswitch.imcontrol.model.SetupInfo.DetectorInfo attribute*), 35
analogChannel (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 35
analogChannel (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36
angleMount (*imswitch.imcontrol.model.SetupInfo.SLMInfo attribute*), 36
APDManager (*built-in class*), 38
api.imcontrol (*built-in class*), 53
availableWidgets (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 34
axes (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager property*), 48
axes (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36

B

BetaScanDesigner (*built-in class*), 40
BetaTTLCycleDesigner (*built-in class*), 41
binning (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 44

C

camera (*imswitch.imcontrol.model.SetupInfo.FocusLockInfo attribute*), 37
CoolLEDLaserManager (*built-in class*), 39
correctionPatternsDir (*imswitch.imcontrol.model.SetupInfo.SLMInfo attribute*), 37
crop() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 44
croppable (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45

D

defaultLaserPresetForScan (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 34
DetectorAction (*class in imswitch.imcontrol.model.managers.detectors.DetectorManager*), 46
DetectorInfo (*built-in class*), 35
DetectorListParameter (*class in imswitch.imcontrol.model.managers.detectors.DetectorManager*), 46
DetectorManager (*class in imswitch.imcontrol.model.managers.detectors.DetectorManager*), 43
DetectorNumberParameter (*class in imswitch.imcontrol.model.managers.detectors.DetectorManager*), 46
DetectorParameter (*class in imswitch.imcontrol.model.managers.detectors.DetectorManager*), 46
detectors (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 34
digitalLine (*imswitch.imcontrol.model.SetupInfo.DetectorInfo attribute*), 35
digitalLine (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 35
digitalLine (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36

E

editable (*imswitch.imcontrol.model.managers.detectors.DetectorManager attribute*), 47
editable (*imswitch.imcontrol.model.managers.detectors.DetectorManager attribute*), 46

F

finalize() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 45
finalize() (*imswitch.imcontrol.model.managers.lasers.LaserManager.LaserManager method*), 47
finalize() (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager method*), 48
flushBuffers() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 45
focusLock (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 34
FocusLockInfo (*built-in class*), 37
forAcquisition (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
forAcquisition (*imswitch.imcontrol.model.SetupInfo.DetectorInfo attribute*), 35
forFocusLock (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
forFocusLock (*imswitch.imcontrol.model.SetupInfo.DetectorInfo attribute*), 35
forPositioning (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager property*), 49
forPositioning (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36
forScanning (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager property*), 49
forScanning (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36
frameCroph (*imswitch.imcontrol.model.SetupInfo.FocusLockInfo attribute*), 37
frameCropw (*imswitch.imcontrol.model.SetupInfo.FocusLockInfo attribute*), 37
frameCropx (*imswitch.imcontrol.model.SetupInfo.FocusLockInfo attribute*), 37
frameCropy (*imswitch.imcontrol.model.SetupInfo.FocusLockInfo attribute*), 37
frameStart (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
fullShape (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
func (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorAction attribute*), 46

G

GalvoScanDesigner (*built-in class*), 40
getChunk() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 45
getDetectorNames() (*api.imcontrol method*), 53
getLaserNames() (*api.imcontrol method*), 53
getLatestFrame() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 45
getPositionerNames() (*api.imcontrol method*), 53
getPositionerPositions() (*api.imcontrol method*), 53
getWaitForSignal(), 51
group (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager attribute*), 46
group (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager attribute*), 47
group (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager attribute*), 46

H

h (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ROIInfo attribute*), 38
HamamatsuManager (*built-in class*), 38
height (*imswitch.imcontrol.model.SetupInfo.SLMInfo attribute*), 37

I

image (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
importScript(), 51
isBinary (*imswitch.imcontrol.model.managers.lasers.LaserManager.LaserManager property*), 47
isPositiveDirection (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36

L

LaserInfo (*built-in class*), 35
LaserManager (*class in imswitch.imcontrol.model.managers.lasers.LaserManager*), 47
LaserPresetInfo (*built-in class*), 38
laserPresets (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35
lasers (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35
loadScanParamsFromFile() (*api.imcontrol method*), 53

M

mainWindow (*built-in class*), 57
managerName (*imswitch.imcontrol.model.SetupInfo.DetectorInfo attribute*), 35
managerName (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 36
managerName (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36
managerName (*imswitch.imcontrol.model.SetupInfo.RS232Info attribute*), 36
managerProperties (*imswitch.imcontrol.model.SetupInfo.DetectorInfo attribute*), 35

managerProperties (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 36
managerProperties (*imswitch.imcontrol.model.SetupInfo.PositionerInfo attribute*), 36
managerProperties (*imswitch.imcontrol.model.SetupInfo.RS232Info attribute*), 36
MHXYStageManager (*built-in class*), 39
model (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
monitorIdx (*imswitch.imcontrol.model.SetupInfo.SLMInfo attribute*), 37
move() (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager method*), 49
movePositioner() (*api.imcontrol method*), 53

N

name (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
name (*imswitch.imcontrol.model.managers.lasers.LaserManager.LaserManager property*), 47
name (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager property*), 49
nidaq (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35
NidaqInfo (*built-in class*), 38
NidaqLaserManager (*built-in class*), 39
NidaqManager (*class in imswitch.imcontrol.model.managers.NidaqManager*), 49
NidaqPositionerManager (*built-in class*), 39

O

options (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorListParameter attribute*), 47

P

parameters (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
PhotometricsManager (*built-in class*), 38
PiezoconceptZManager (*built-in class*), 40
pixelSize (*imswitch.imcontrol.model.SetupInfo.SLMInfo attribute*), 37
pixelSizeUm (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
PointScanTTLCycleDesigner (*built-in class*), 41
position (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager property*), 49
positioner (*imswitch.imcontrol.model.SetupInfo.FocusLockInfo attribute*), 37
PositionerInfo (*built-in class*), 36
PositionerManager (*class in imswitch.imcontrol.model.managers.positioners.PositionerManager*), 48
positioners (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35

R

ROIInfo (*built-in class*), 38
rois (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35
rs232devices (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35
RS232Info (*built-in class*), 36
RS232Manager (*built-in class*), 40
RS232Manager (*class in imswitch.imcontrol.model.managers.rs232.RS232Manager*), 49
RS232sManager (*class in imswitch.imcontrol.model.managers.RS232sManager*), 49
runScan() (*api.imcontrol method*), 53

S

sampleRate (*imswitch.imcontrol.model.SetupInfo.ScanInfo attribute*), 37
saveBeamParamsToFile() (*api.imcontrol method*), 53
scan (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35
scanDesigner (*imswitch.imcontrol.model.SetupInfo.ScanInfo attribute*), 37
scanDesignerParams (*imswitch.imcontrol.model.SetupInfo.ScanInfo attribute*), 38
ScanInfo (*built-in class*), 37
send() (*imswitch.imcontrol.model.managers.rs232.RS232Manager.RS232Manager method*), 50
setAnalog() (*imswitch.imcontrol.model.managers.NidaqManager.NidaqManager method*), 49
setBinning() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 45
setCurrentModule() (*mainWindow method*), 57
setDetectorBinning() (*api.imcontrol method*), 53
setDetectorParameter() (*api.imcontrol method*), 53
setDetectorROI() (*api.imcontrol method*), 53
setDetectorToRecord() (*api.imcontrol method*), 53
setDigital() (*imswitch.imcontrol.model.managers.NidaqManager.NidaqManager method*), 49
setEnabled() (*imswitch.imcontrol.model.managers.lasers.LaserManager.LaserManager method*), 47
setLaserActive() (*api.imcontrol method*), 53
setLaserValue() (*api.imcontrol method*), 54
setLiveViewActive() (*api.imcontrol method*), 54
setLiveViewCrosshairVisible() (*api.imcontrol method*), 54

setLiveViewGridVisible() (*api.imcontrol method*), 54
setParameter() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 45
setPosition() (*imswitch.imcontrol.model.managers.positioners.PositionerManager.PositionerManager method*), 49
setPositioner() (*api.imcontrol method*), 54
setPositionerStepSize() (*api.imcontrol method*), 54
setRecFilename() (*api.imcontrol method*), 54
setRecFolder() (*api.imcontrol method*), 54
setRecModeScanOnce() (*api.imcontrol method*), 54
setRecModeScanTimelapse() (*api.imcontrol method*), 54
setRecModeSpecFrames() (*api.imcontrol method*), 54
setRecModeSpecTime() (*api.imcontrol method*), 54
setRecModeUntilStop() (*api.imcontrol method*), 54
setScanModeActive() (*imswitch.imcontrol.model.managers.lasers.LaserManager.LaserManager method*), 48
setValue() (*imswitch.imcontrol.model.managers.lasers.LaserManager.LaserManager method*), 48
shape (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 45
signals() (*api.imcontrol method*), 54
slm (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ViewSetupInfo attribute*), 35
SLMInfo (*built-in class*), 36
snapImage() (*api.imcontrol method*), 54
startAcquisition() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 45
startRecording() (*api.imcontrol method*), 55
startTrigger (*imswitch.imcontrol.model.SetupInfo.NidaqInfo attribute*), 38
stepPositionerDown() (*api.imcontrol method*), 55
stepPositionerUp() (*api.imcontrol method*), 55
stopAcquisition() (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager method*), 46
stopRecording() (*api.imcontrol method*), 55
supportedBinnings (*imswitch.imcontrol.model.managers.detectors.DetectorManager.DetectorManager property*), 46

T

timerCounterChannel (*imswitch.imcontrol.model.SetupInfo.NidaqInfo attribute*), 38
TISManager (*built-in class*), 39
TTLCycleDesigner (*imswitch.imcontrol.model.SetupInfo.ScanInfo attribute*), 37
TTLCycleDesignerParams (*imswitch.imcontrol.model.SetupInfo.ScanInfo attribute*), 37

U

updateFreq (*imswitch.imcontrol.model.SetupInfo.FocusLockInfo attribute*), 38

V

value (*imswitch.imcontrol.model.managers.detectors.DetectorManager.Det attribute*), 47
value (*imswitch.imcontrol.model.managers.detectors.DetectorManager.Det attribute*), 46
value (*imswitch.imcontrol.view.guitools.ViewSetupInfo.LaserPresetInfo attribute*), 38
valueDecimals (*imswitch.imcontrol.model.managers.lasers.LaserManage property*), 48
valueRangeMax (*imswitch.imcontrol.model.managers.lasers.LaserManage property*), 48
valueRangeMax (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 36
valueRangeMin (*imswitch.imcontrol.model.managers.lasers.LaserManage property*), 48
valueRangeMin (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 36
valueRangeStep (*imswitch.imcontrol.model.managers.lasers.LaserManag property*), 48
valueRangeStep (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 36
valueUnits (*imswitch.imcontrol.model.managers.detectors.DetectorMana attribute*), 46
valueUnits (*imswitch.imcontrol.model.managers.lasers.LaserManager.La property*), 48
ViewSetupInfo (*built-in class*), 34

W

w (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ROIInfo attribute*), 38
wavelength (*imswitch.imcontrol.model.managers.lasers.LaserManager.La property*), 48
wavelength (*imswitch.imcontrol.model.SetupInfo.LaserInfo attribute*), 36
wavelength (*imswitch.imcontrol.model.SetupInfo.SLMInfo attribute*), 37
width (*imswitch.imcontrol.model.SetupInfo.SLMInfo attribute*), 37

X

x (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ROIInfo attribute*), 38

Y

y (*imswitch.imcontrol.view.guitools.ViewSetupInfo.ROIInfo attribute*), 38