ALICA Documentation

Release 0.3.2

Marcel Stefko, Kyle M. Douglass

Jun 07, 2018

Contents

Automated Laser Illumination Control Algorithm

Quickstart

This page describes a brief tutorial on how to install and begin working with ALICA. It necessarily avoids any details on how ALICA works; instead, its focus is on helping you become acquainted with working with ALICA.

1.1 Installation

1.1.1 Micro-Manager 2

If Micro-Manager 2.0 or greater is not already installed on your machine, then follow the steps in this section.

1. Navigate to https://valelab4.ucsf.edu/~MM/nightlyBuilds/2.0.0-gamma/Windows/ and download the latest nightly build for your system. (Note that ALICA currently only works with Micro-Manager 2.0gamma, **NOT** 2.0beta.)

2. Install Micro-Manager by following the directions provided on the previously mentioned website. Make note of the installation directory, which on Windows is usually something like *C:\Program Files\Micro-Manager-2.0*.

1.1.2 ALICA

ALICA is distributed as a .jar file and is easily installed by copying the file into the Micro-Manager plugins folder.

1. Navigate to https://github.com/MStefko/ALICA/releases and download the ALICA*.jar file corresponding to the latest release.

2. Copy ALICA*.jar to the *MM2ROOT/mmplugins* directory, where *MM2ROOT/* is the Micro-Manager installation directory.

3. Navigate to https://github.com/LEB-EPFL/ALICA_ACPack and download the ALICA_ACPack*.jar file corresponding to the latest release.

4. If you are using ALICA_ACPack version 0.2.0 or later, you will need to update a few jars that Micro-Manager uses. Download imglib2-5.3.0.jar from http://maven.imagej.net/service/local/repositories/releases/content/net/imglib2/imglib2/5.3.0/imglib2-5.3.0.jar and imglib2-roi-0.5.1.jar from http://maven.imagej.net/

service/local/repositories/releases/content/net/imglib2/imglib2-roi/0.5.1/imglib2-roi-0.5.1.jar and place them in the *pluginsMicro-Manager* folder inside the Micro-Manager installation directory. Delete the older versions of imglib2 and imglib2-roi that are already located there.

5. Copy ALICA_ACPack*.jar to the *MM2ROOT/mmplugins* directory, where *MM2ROOT/* is the Micro-Manager installation directory.

6. Verify that ALICA was installed and recognized by starting Micro-Manager and selecting *Plugins > Device Control > ALICA* in the Micro-Manager menu bar. (ALICA will not be located in the ImageJ menu bar.) The ALICA Setup window should appear, which will verify that ALICA is properly installed.
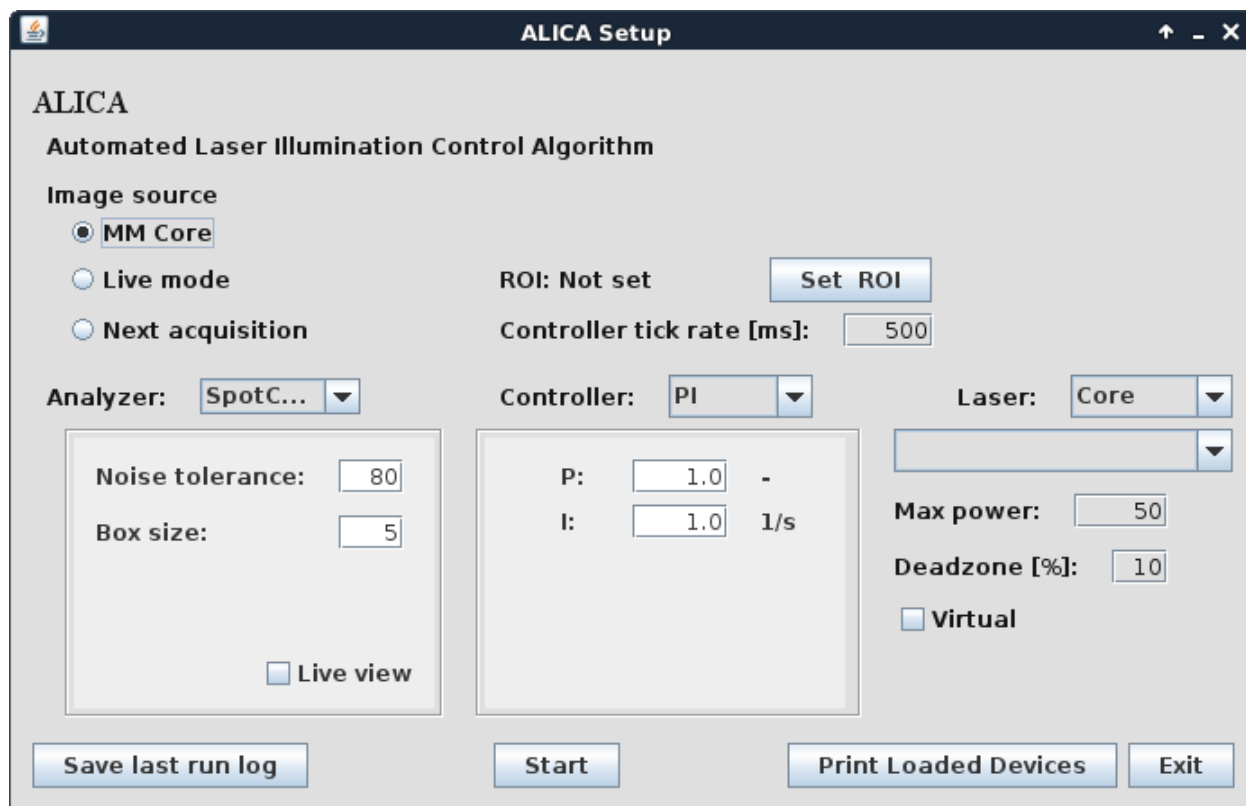

1.2 Using ALICA

ALICA reads an image stream from Micro-Manager and uses these images to estimate the real-time density of fluorescence emitting molecules within the microscope's field of view. As the estimated density of emitting fluorophores changes (due to bleaching or changes in the sample, for example), ALICA will automatically adjust the laser power to maintain a set emitter density.

1.2.1 Step 1: Select an image source

First, select a source for the image stream that ALICA will analyzer. Your options include

1. the Micro-Manager core, which contains unprocessed images from the camera;

2. the Micro-Manager Live mode, which contains the images that appear in Micro-Manager's Snap/Live View window. These images may be preprocessed by Micro-Manager's On-The-Fly Image Processors;

3. the next Multi-Dimensional Acquisition.

We suggest choosing the **Live mode** option when you are just starting to use ALICA because it is the most interactive option. During actual acquisitions, **MM Core** is recommended due to its superior performance, unless you need to perform some image preprocessing using the MicroManager processing pipeline before feeding the images to ALICA.
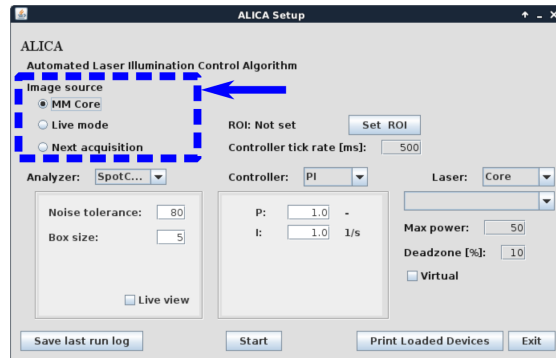

1.2.2 Step 2: Select and configure the analyzer

An analyzer is an algorithm that estimates the density of fluorophores that are visible in an image. At the time of this writing, ALICA included the following analyzers

1. a spot counter, which counts the number of fluorescent spots in the images;

2. AutoLase, an algorithm which estimates fluorophore densities by identifying the single pixel within the field of view that has been above a given threshold for the longest time;

3. QuickPALM, a tool which identifies fluorescent spots and then performs a subpixel localization of each spot;

4. an integrator, which simply computes the integrated intensity of an image.

The **spot counter** performs well for many samples and also offers a live view which provides real-time visual feedback of which spots it identifies.
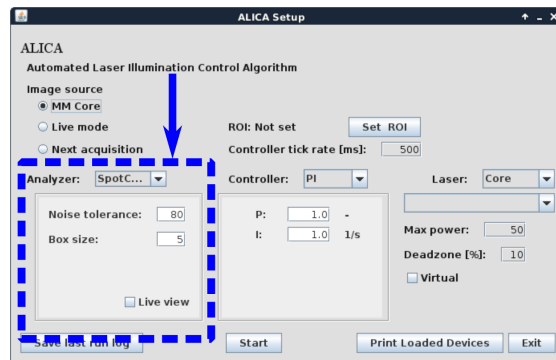

1.2.3 Step 3: Select and configure the controller

A controller is a feedback loop that adjusts the laser power so that the estimated density of emitters remains as close as possible to a previously determined set point. The difference between the current estimate and the set point is called the error signal. The choice of controllers includes

1. a proportional-integral (PI) controller, which responds both proportionately to the error signal and to the time integral of the error signal;

2. a manual controller, which gives control over the laser to the microscopist;

3. an inverter, which adjusts the laser by a factor that is proportional to the inverse of the error signal (e.g. high error signal > low laser power and vice versa);

4. a self-tuning (PI) controller, which uses a pulse of laser light to estimate the optimum values for the P and I parameters.

We recommend starting with **manual** control to first learn how the analyzer responds to changes in your sample. Once you understand a little bit about this, you can try a **self-tuning PI controller**. The self-tuning PI controller can only tune itself when the sample is already under STORM or PALM imaging conditions. For direct STORM, this means that the fluorophores should already be blinking.
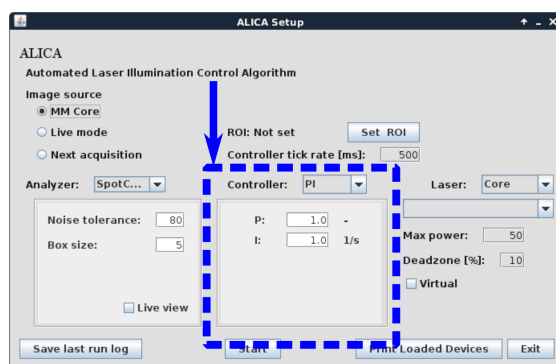

1.2.4 Step 4: Select the device to be controlled

A device and its property that corresponds to output power needs to be specified for the controller to actually do something. In most STORM and PALM experiments, the density of emitters is typically controlled using an ultraviolet laser. To be able select this laser, it needs to be added to the current Micro-Manager hardware configuration. Once the laser is selected, choose its power setting from the next drop-down menu.

To prevent a run-away laser illumination, you can set the maximum power for the controller. We typically do not set this above a few tens of milliWatts, but the actual value depends on the sample.

If you are testing ALICA and do not want to select a device, then check the *Virtual* checkbox. This will instruct the controller that it should not affect the state of any hardware devices. Checking it will allow you to test ALICA's analyzers without performing any hardware control.
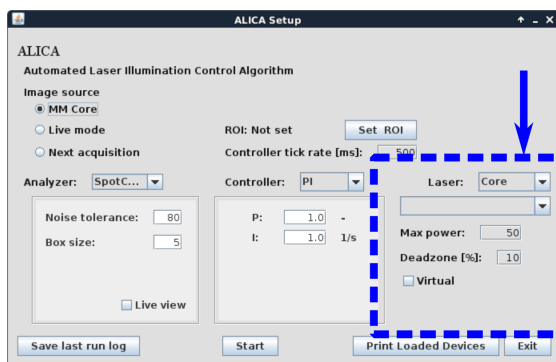
1.2.5 Step 5: Start the monitor

When ready, click *Start* in the ALICA Setup window. This will open the ALICA monitor window, which will look similar to the image below.
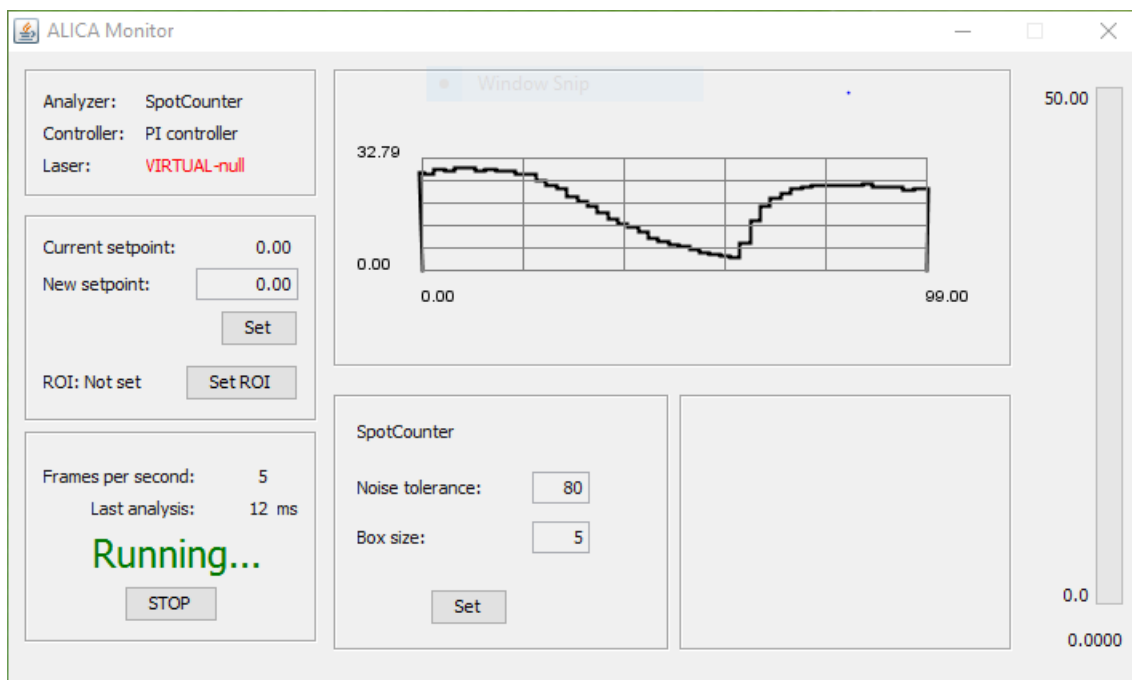
In the upper left, you can find a readout on the currently selected analyzer, controller, and laser. In this example image, the analyzer is the SpotCounter, controller is a PI controller, and the device is actually not set, i.e. the *Virtual* checkbox was checked in the ALICA Setup window.

Below this box you can set the desired density of fluorophores in the *New setpoint:* text box. After typing in a new value, click *Set* to activate the change. If you draw a region of interest (ROI) in the Snap/Live View window, you can set ALICA to only analyze this region by clicking the *Set ROI* button. You can also drag this ROI around the the Snap/Live View window in real-time and ALICA will respond in real-time.

Moving further down the left-hand side of the ALICA Monitor window, you will find information on the number of frames processed by the analyzer per second and the time taken to analyze the last frame. You may also close the ALICA Monitor window in this section by clicking the *Stop* button.

In the middle of the ALICA Monitor window on the top is a real-time plot of the output of the analyzer as a function of time. The units on the y-axis of this plot will depend on the output of the analyzer. For example, the SpotCounter outputs a number of spots, but AutoLase will output the longest "On" pixel in units of time.

Below this plot you may update the analyzer settings.

Finally, on the far right of the ALICA Monitor window is a status bar that reflects the current output of the laser. The maximum value of the status bar is the maximum value set in the ALICA Setup window.

1.2.6 Step 6: Start taking images

When you are ready, start taking images using the source of images that you set in the ALICA Setup window. For example, if you selected *Live mode*, then all you need to do is start a Live stream in Micro-Manager. The different parts of the ALICA Monitor window will begin to reflect the output of the analyzer and controller once images begin arriving in this stream.

If the controller was set to Manual, try adjusting the ultraviolet laser power and watching how the output of the analyzer changes in response. If you are using a PI controller, you may notice a slight oscillation in the readout of the analyzer. This is caused by the particular values you have set for P and I.

If you selected a self-tuning PI controller, Micro-Manager will pulse the laser a short time after the acquisition has started and observe how the density of fluorophores changes in response to the pulse. It will then determine the

optimum values for P and I. You may set the set point after the controller has tuned itself.
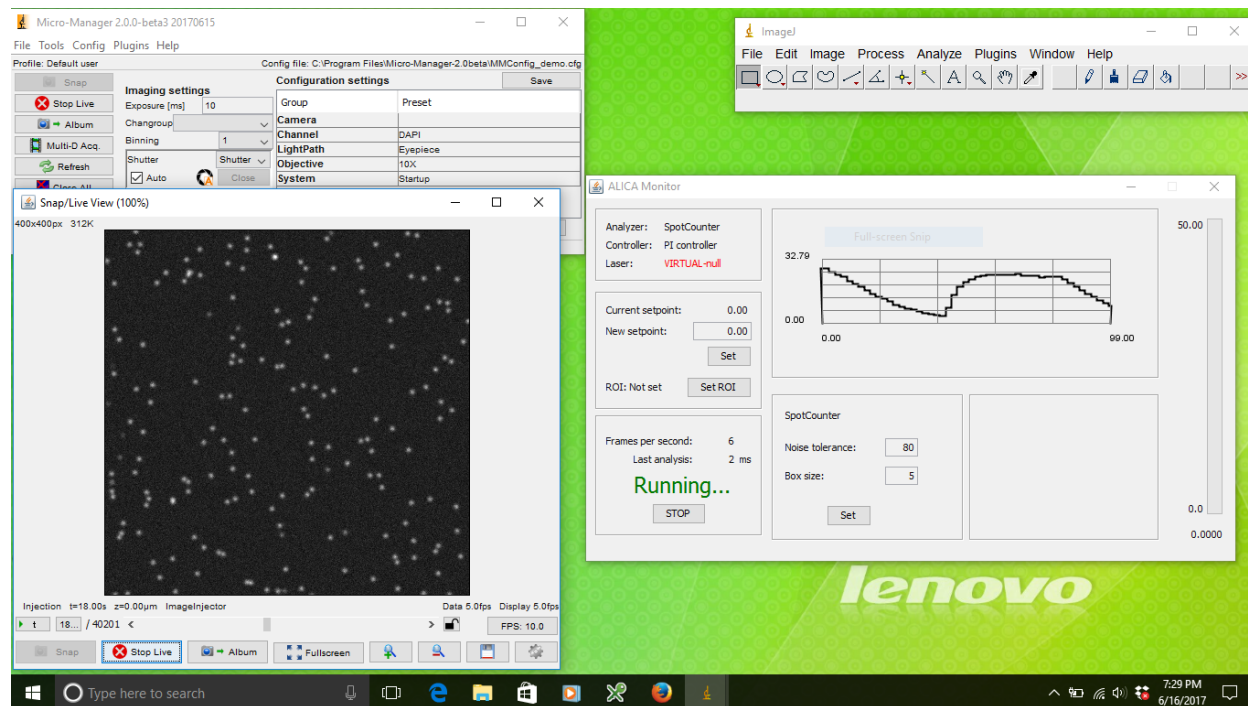
1.3 What's next?

Tuning the parameters in ALICA may take some time and experimentation, even with the self-tuning controller. Tuning may not be easy to do on real samples due to time constraints and costly sample preparations. To ease this process, we created a simulation environment to help you learn how ALICA works.

You may read about how to setup this environment on the simulation page.

Parameter Explanations

2.1 Image Source

ALICA offers 3 different options of acquiring images from MicroManager:

- **MM Core** Images are drawn directly from the circular buffer. This method is the fastest, and recommended in most cases, since it can smoothly function whenever the camera is acquiring images.

- **Live mode** Images are drawn from the Datastore associated with the current live mode view. Use this if you wish to do some on-the-fly processing using the MicroManager pipeline, before passing the image to Analyzer.

- **Next acquisition** Images will be drawn from the Datastore associated with the first acquisition that is started afterwards.

2.2 ROI

Using the live view, you can select a region of interest to constrain the analyzed area (for example if the density of fluorophores is uneven, or the analysis of full image takes too long).

2.3 Controller tick rate

This value in milliseconds defines how often the Controller queries the Analyzer, and adjusts the laser output.

2.4 Laser

- **Max Power** Maximal power setpoint of the laser. ALICA will not adjust the laser power above this value.

- **Deadzone [%]** The minimum adjustment to the power setpoint that the controller may make as a percentage of the current value. Adjustments to the laser by an amount less than this are not permitted, which prevents unnecessary fine-tuning of the laser.

- **Virtual** If checked, the output is not passed to the device. Useful for debugging or preview of parameters.

Photodynamics Simulations with ALICA and SASS

Extensibility is a core design principle of ALICA. If the builtin components do not suit the needs of your application, then you can write your own set of tools using the frameworks of ALICA and Micro-Manager. Alternatively, you may find that ALICA already suits your needs but you need to do some testing in a controlled environment prior to using it in your measurements. We developed the **STORM Acquistion Simulation Software** (SASS) to assist in both of these situations.

This document explains how to setup SASS to test ALICA in a fully controlled simulation environment.

3.1 Install the Simulation Environment

SASS and ALICA are both distributed as Java .jar files. In addition to these you will need to download our Image Injector plugin, a Micro-Manager plugin which allows you to simulate acquistions by feeding images from a .tif file into the Micro-Manager live window. To install these files, you simply download the latest .jar from the Releases page of the respective projects and copy the files into the appropriate directories.

3.1.1 Micro-Manager 2

Before starting, you need the latest nightly build of Micro-Manager 2.0 (or higher).

1. Go to https://www.micro-manager.org/wiki/Version_2.0 and download the latest nightly build for your system.

2. Install Micro-Manager. Make note of the installation directory since you will need it later to install the .jar files.

3.1.2 ALICA

1. Navigate to https://github.com/MStefko/ALICA/releases and download ALICA.jar from the latest release.

2. Copy ALICA.jar to the *MM2ROOT/mmplugins* directory, where *MM2ROOT* refers to the installation directory of Micro-Manager.

3.1.3 SASS

SASS is a Fiji plugin and is not intended to work with the same copy of ImageJ that is used by Micro-Manager. This is because SASS has its own internal copy of ALICA that conflicts with Micro-Manager's copy.

Do not install SASS in the same directory as Micro-Manager.

Instead, we will install SASS in a separate Fiji installation.

1. If you have not already done so, download a copy of Fiji from http://fiji.sc/ and unpack it. Make note of the directory in which you installed it.

2. Navigate to https://github.com/MStefko/SASS/releases and download SASS_VERSION.jar from the latest release. VERSION will vary depending on the latest release.

3. Copy the SASS .jar file to *FIJIROOT/plugins* directory, where *FIJIROOT* is the installation directory of Fiji. (Note that the folder this time is **plugins**, not **mmplugins**.)

3.1.4 Image Injector Plugin

1. Go to https://github.com/MStefko/ImageInjectorPlugin/releases and download the ImageInjector.jar file from the latest release.

2. Copy the .jar file to the *MM2ROOT/mmplugins* directory.
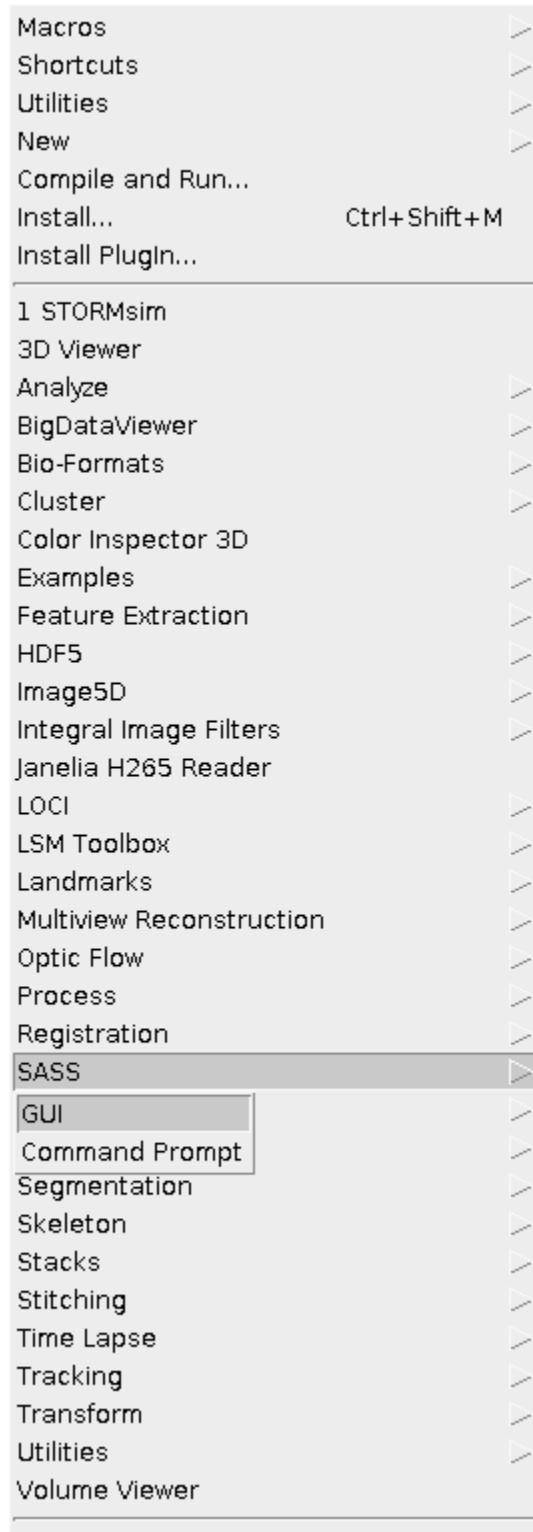
3.2 Simulation Workflow

The workflow goes as follows:

1. Use SASS to simulate a time series image stack of a PALM or STORM experiment and save the stack as a .tif file.

2. Use the Image Injector Plugin to feed the images in the stack into the Micro-Manager live window.

3. Run ALICA in virtual mode and observe how it responds to the simulated conditions in the image stack.

3.2.1 Step 1: Simulate a PALM/STORM Experiment with SASS

If you do not already have a .tif file of a time series image stack from a PALM/STORM experiment, you can simulate one by following the steps in this section.

1. Launch Fiji.

2. Verify that the SASS plugin is recognized by Fiji and runs by clicking to *Plugins > SASS > GUI* in the ImageJ menu bar.

Macros	⟩
Shortcuts	⟩
Utilities	⟩
New	⟩
Compile and Run...	
Install...	Ctrl+Shift+M
Install PlugIn...	
1 STORMsim	
3D Viewer	
Analyze	⟩
BigDataViewer	⟩
Bio-Formats	⟩
Cluster	⟩
Color Inspector 3D	
Examples	⟩
Feature Extraction	⟩
HDF5	⟩
Image5D	⟩
Integral Image Filters	⟩
Janelia H265 Reader	
LOCI	⟩
LSM Toolbox	⟩
Landmarks	⟩
Multiview Reconstruction	⟩
Optic Flow	⟩
Process	⟩
Registration	⟩
SASS	▷
GUI	⟩
Command Prompt	⟩
Segmentation	⟩
Skeleton	⟩
Stacks	⟩
Stitching	⟩
Time Lapse	⟩
Tracking	⟩
Transform	⟩
Utilities	⟩
Volume Viewer	

3. The GUI configuration window for the SASS simulation environment should appear. Select your parameters for the simulation. A full description of the simulation parameters is outside the scope of this documentation. However, you will want to set the Controller setting to **Manual** since we want only a simple simulation where we manually select the laser power.

4. Remember the value for the *Max output* parameter. This is the maximum output power of the simulated laser, and you will need it in a later step.

5. Once everything is set, click the *Initialize* button to initialize the simulation.

6. Set the *Set Point* value to something smaller than the value of *Max output*. This value determines the output power of the simulated laser.

7. When ready, start the simulation by clicking the *Start* button. This will begin to populate an image stack with simulated STORM/PALM images.
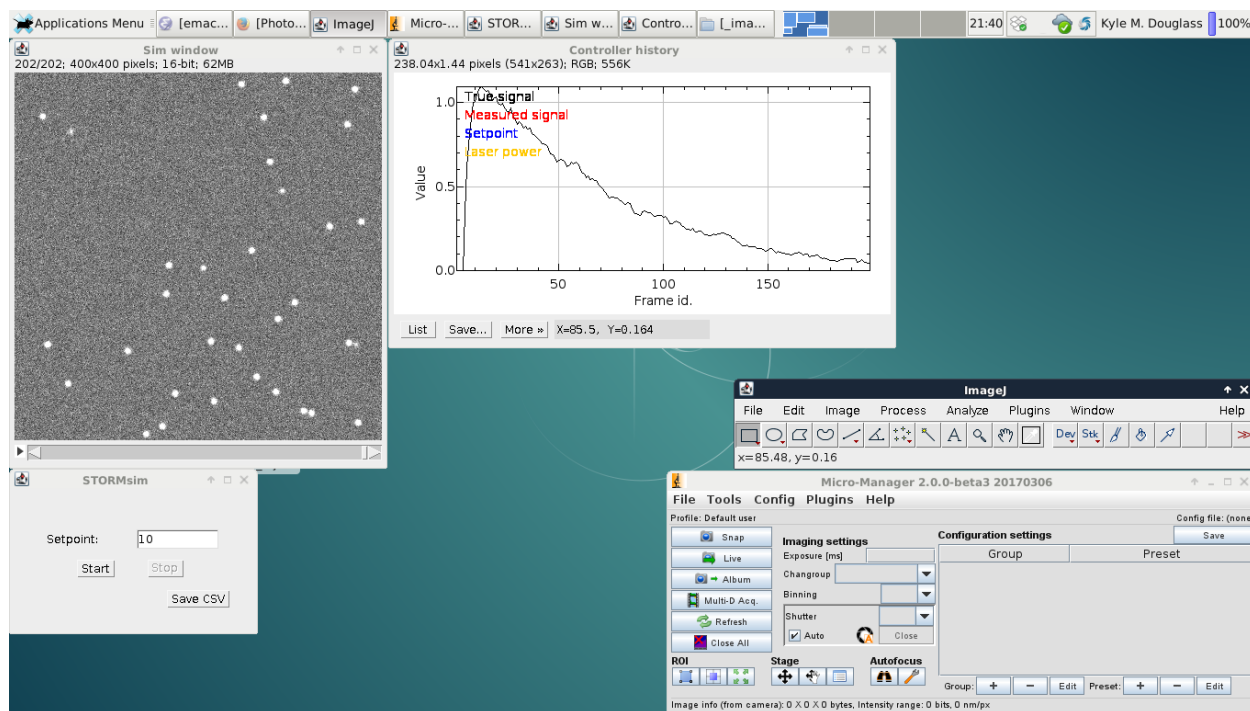
8. You may stop the simulation and change the laser output power by clicking *Stop* in the STORMsim window and adjusting the set point. Click *Start* to pick up where the simulation left off with the new laser power.

9. Once you have simulated a desired number of images in the stack, save the image stack by navigating to *File > Save As > Tiff. . .* in the ImageJ menu bar.

From this point you have two options for further exploration. You can use SASS to directly test the different analyzers and controllers. Or, you can continue further to directly test ALICA in a simulated Micro-Manager acquisition.

3.2.2 Step 2: Setup the Image Injector Plugin

Once you have a .tif stack, the next step is to setup the image injector to simulate a Micro-Manager acquisition.

1. Launch Micro-Manager. Select the MM Demo configuration when prompted to select a hardware configuration. (This Quickstart assumes that you are running Micro-Manager as an ImageJ plugin, which is the most common behavior.)

2. Open the *On-The-Fly Processor Pipeline* window by navigating to *Plugins > On-The-Fly Image Processing > Configure Processors. . .* in the Micro-Manager menu bar.
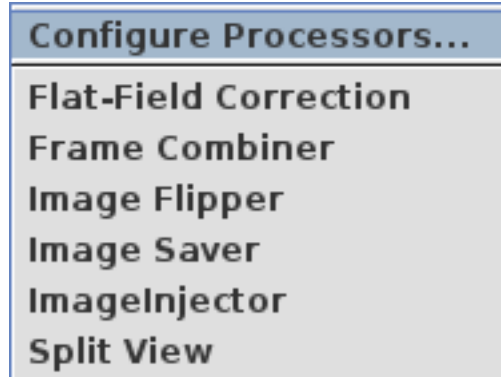
3. In the window that appears, verify whether an ImageInjector processor already exists in the pipeline. If not, add one by clicking *+ Add. . . > ImageInjector*.
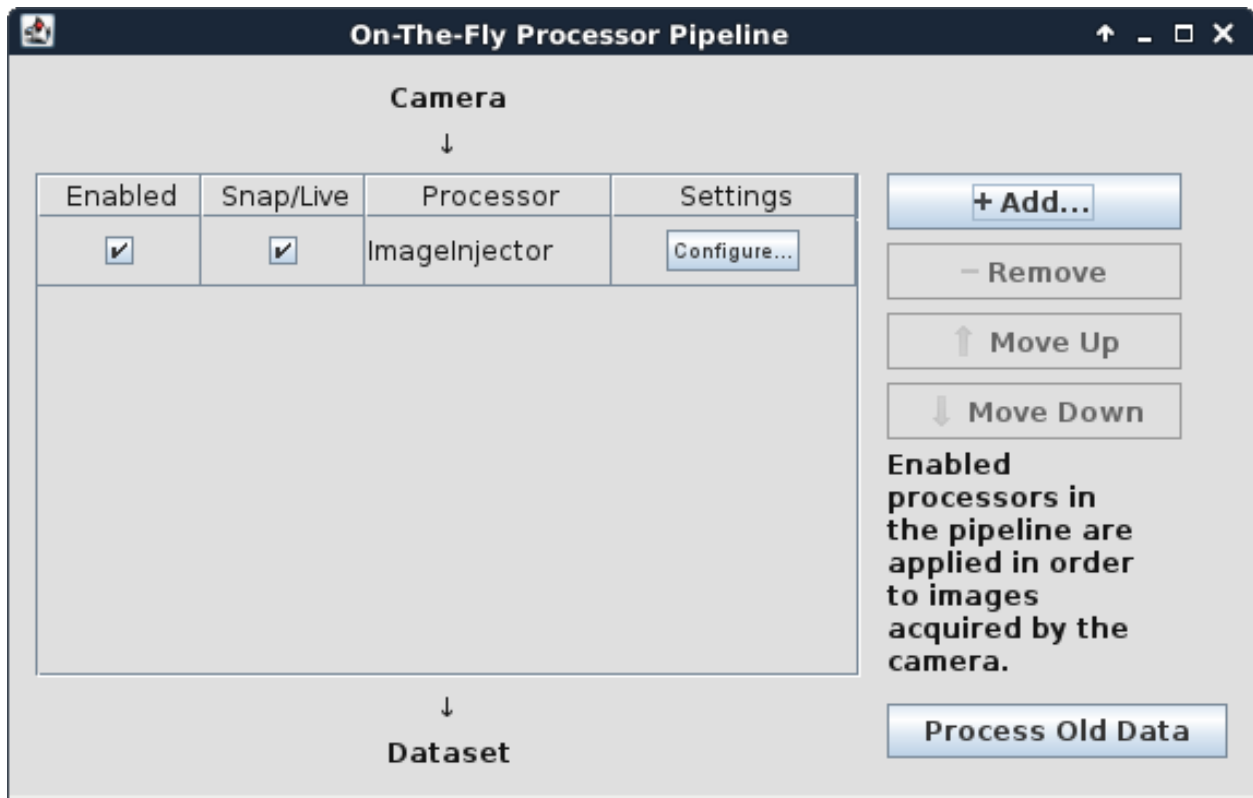
4. Click the *Configure. . .* button for the ImageInjector processor.

5. In the dialog that appears, click the *Choose file. . .* button and select the .tif stack of images to inject.

6. We find that it helps to set the *Frames per second* value to something small during your initial tests, such as 5.

7. Click *OK* when you are finished configuring the processor. You may close the configuration window at this point.

8. Click the *Live* button in the Micro-Manager GUI window or in the Snap/Live View window if it's already open. You should now see the images from the .tif stack stream through the Snap/Live View window.

9. You can stop and restart the live stream at will. The stream will cycle back to the start of the image stack once the end is reached.

3.2.3 Step 3: Launch ALICA in Virtual Mode

Now that Micro-Manager has been setup to stream pre-generated images through its Snap/Live View window, we can launch ALICA and run it in virtual mode.

1. Navigate to *Plugins > Device Control > ALICA* in the Micro-Manager menu bar.

2. Select *Live mode* as the Image source and check the *Virtual* box under the options for the control device.
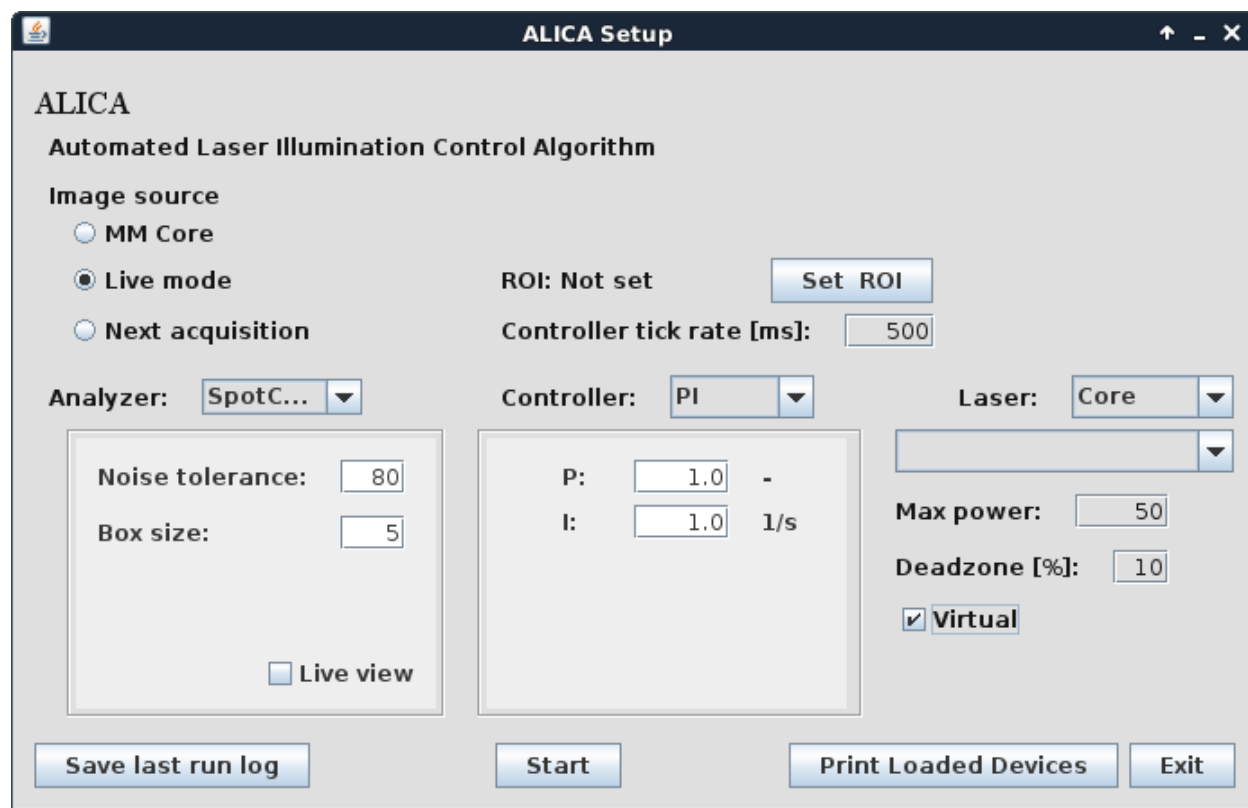
3. Click *Start*. This will open the monitor window which provides real-time reports about the ALICA's operation, such as fluorophore density estimates and the laser power.

4. Click the *Live* button in the main Micro-Manager GUI window. You should see the reports in ALICA's Monitor window respond to changes in the images streaming through the Snap/Live View window. If you don't immediately see any change in the monitors, try stopping and starting Live mode again in the Snap/Live View window.
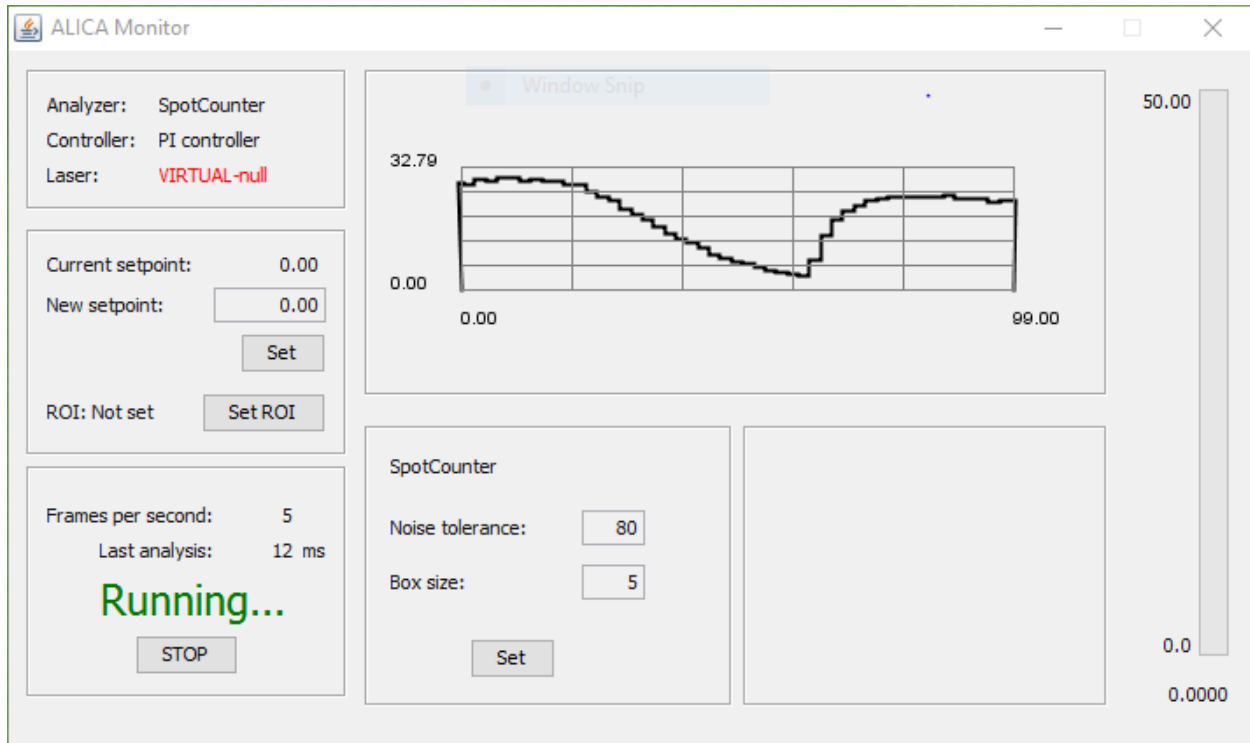
5. When you want to close the Monitor window, click *Stop* in the Monitor window.

3.3 What's Next?

Now that everything is setup, here are some further things that we recommend playing with to better understand how ALICA works.

- Check the *Live view* checkbox in the SpotCounter analyzer settings for a live view of the identified spots.

- Change the Analyzer from SpotCounter to AutoLase or QuickPALM for ways to estimate fluorophore densities in the images.

- Try ALICA's virtual mode on actual experimental image stacks.

- Restrict the fluorophore density estimates to a subregion of the images by selecting a rectangular region in the Snap/Live View and clicking the *Set ROI* button in the ALICA Monitor window. The best way to see how this works is to use Spot Counter's *Live view* setting. You can even drag the region around the field of view and watch the changes reflected in the SpotCounter's live view in real-time.

- Use SASS to directly test different Analyzer and Controller settings outside of ALICA.

Extending ALICA

This page describes how you can develop your own ALICA Analyzer or Controller to suit your needs.

This page will guide you through the process of creating your own Analyzer, but applies as well to creating a Controller.

General knowledge of Java programming is assumed and recommended.

Required JDK version: `1.6.0_31` (Same as MicroManager's)

4.1 Implementing a custom Analyzer
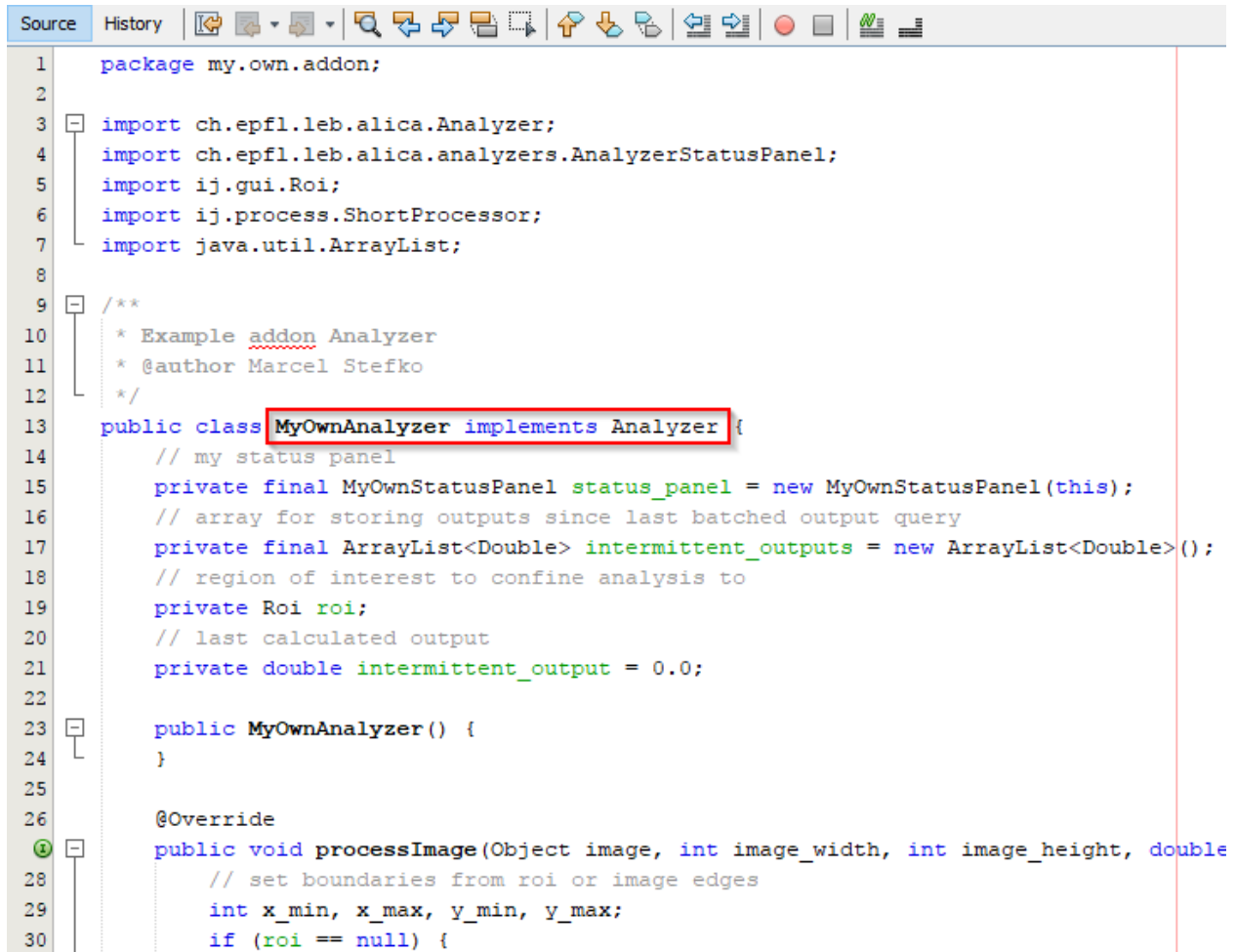
4.1.1 Step 1: Importing the Analyzer interface

First, download the `ALICA_dev.jar` file from the relevant release, and include it as a resource of your project (in NetBeans, add it to `Project Properties -> Libraries -> Compile-time Libraries` using the `Add JAR/Folder` button). This jar file contains all ALICA libraries, as well as necessary MicroManager and ImageJ libraries.

In the same pane, you have to ensure that your JDK version is 1.6 (same as MicroManager's).
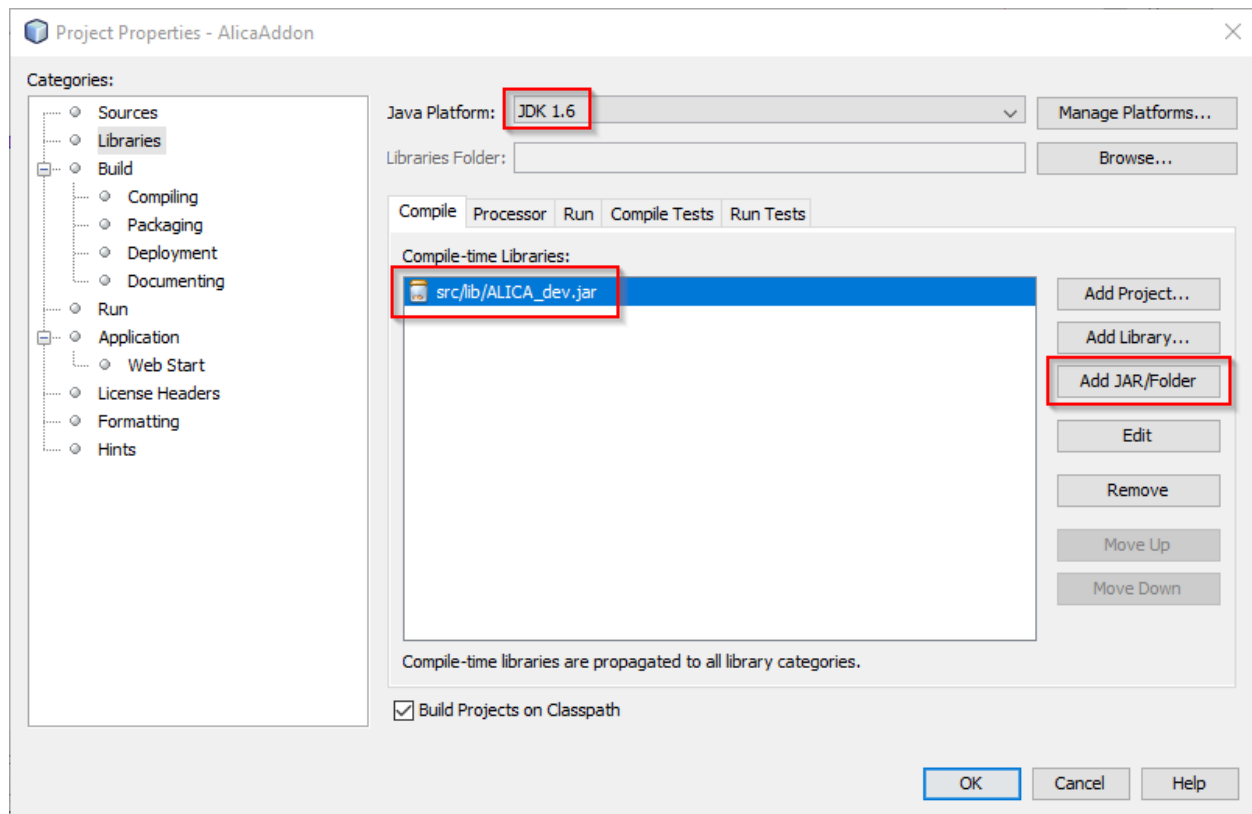
4.1.2 Step 2: Implementing the Analyzer and its Setup/Status panels

Implement the `Analyzer` interface from package `ch.epfl.leb.alica`, and extend the `AnalyzerSetupPanel` and (optionally) `AnalyzerStatusPanel` abstract classes from package `ch.epfl.leb.alica.analyzers`. Check the API documentation for details. You can also consult the source code for already implemented Analyzers on GitHub.

```
Source  History

 1    package my.own.addon;
 2
 3    import ch.epfl.leb.alica.Analyzer;
 4    import ch.epfl.leb.alica.analyzers.AnalyzerStatusPanel;
 5    import ij.gui.Roi;
 6    import ij.process.ShortProcessor;
 7    import java.util.ArrayList;
 8
 9    /**
10     * Example addon Analyzer
11     * @author Marcel Stefko
12     */
13    public class MyOwnAnalyzer implements Analyzer {
14        // my status panel
15        private final MyOwnStatusPanel status_panel = new MyOwnStatusPanel(this);
16        // array for storing outputs since last batched output query
17        private final ArrayList<Double> intermittent_outputs = new ArrayList<Double>();
18        // region of interest to confine analysis to
19        private Roi roi;
20        // last calculated output
21        private double intermittent_output = 0.0;
22
23        public MyOwnAnalyzer() {
24        }
25
26        @Override
27        public void processImage(Object image, int image_width, int image_height, double
28            // set boundaries from roi or image edges
29            int x_min, x_max, y_min, y_max;
30            if (roi == null) {
```

To give a little bit of intuition, the `AnalyzerSetupPanel` serves as a Builder for Analyzers. In the Panel, the user can modify initial settings of the Analyzer. When ALICA `Start` button is clicked, the `initAnalyzer()` method is triggered, which builds the Analyzer. This Analyzer can provide a `AnalyzerStatusPanel`, which (if provided) is embedded in the ALICA monitor GUI, and allows further interaction with the Analyzer.

In NetBeans, it is easier to first create a Swing JPanel form, implement user input fields, and then finally change the `implements javax.swing.JPanel` declaration to `extends ch.epfl.leb.alica.analyzers.AnalyzerSetupPanel`, and implement the required methods (similarly for StatusPanel).

```java
package my.own.addon;

import ch.epfl.leb.alica.Analyzer;
import ch.epfl.leb.alica.analyzers.AnalyzerSetupPanel;

/**
 * Example setup panel
 * @author Marcel Stefko
 */
public class MyOwnSetupPanel extends AnalyzerSetupPanel {

    public MyOwnSetupPanel() {...3 lines }

    /** This method is called from within the constructor to ...6 lines */
    @SuppressWarnings("unchecked")
    Generated Code

    // Variables declaration - do not modify
    private javax.swing.JLabel jLabel1;
    // End of variables declaration

    @Override
    public Analyzer initAnalyzer() {
        return new MyOwnAnalyzer();
    }

    @Override
    public String getName() {
        return "New analyzer!";
    }
}
```

4.1.3 Step 3: Compiling the created Analyzer

Once all required functionality is implemented, compile the project into a .jar file. Remember, that the jar filename **must** start with `ALICA_`, e.g. `ALICA_MyOwnAnalyzer.jar`. Place this jar file into the `mmplugins/` folder of MicroManager.

When you launch ALICA, all added Analyzers and Controllers should be accessible via their respective dropdown menus.
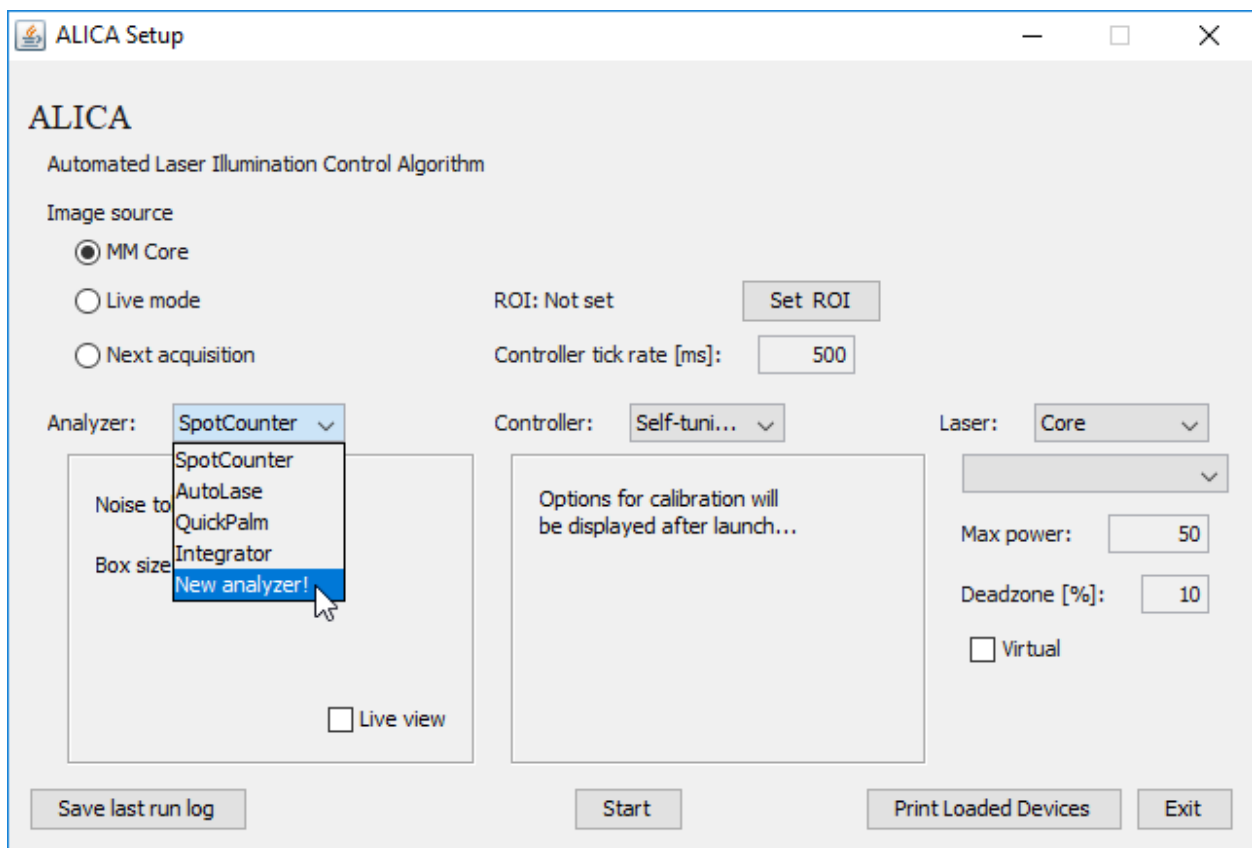
Frequently Asked Questions

5.1 General

5.1.1 Doesn't AutoLase already do autonomous illumination control for STORM/PALM?

AutoLase was developed for one particular use-case: automated PALM imaging of relatively sparse bacteria populations on microscopes with small fields of view. In more general conditions, AutoLase can completely fail to maintain an optimum illumination for STORM/PALM imaging because it cannot easily distinguish between true fluorescence signals and those from other sources such as fiducial markers, dust, or sample autofluorescence. Recent advances that extend PALM/STORM to large fields of view[1] further compound these problems because the chances of capturing a signal from a foreign source are greatly increased. Simply put, AutoLase cannot adequately account for sample heterogeneity.

Recognizing that every sample has different illumination requirements and varying degrees of noise, we developed ALICA as an extensible, robust, and general-purpose tool for autonomous illumination control in PALM/STORM experiments.

5.1.2 How do I do determine the value for the set point?

The set point is the value from the analyzer that the controller tries to maintain. Because of this, the meaning of the set point will vary depending on the analyzer you choose. For example, the set point for the spot counter is in units of number of spots per $100\,\mu m \times 100\,\mu m$.

A pretty good way to empirically find the set point for any analyzer is to perform a STORM or PALM experiment and manually adjust the laser powers until your sample is blinking optimally. Then, use the real-time plot in the upper

---

[1] K. M. Douglass et al., "Super-resolution imaging of multiple cells by optimized flat-field epi-illumination," Nature Photonics 10, 705-708 (2016). http://www.nature.com/nphoton/journal/v10/n11/full/nphoton.2016.200.html ; Z. Zhao et al., "High-power homogeneous illumination for super-resolution localization microscopy with large field-of-view," Optics Express 25, 13382-13395 (2017). https://www.osapublishing.org/oe/abstract.cfm?uri=oe-25-12-13382 ; R. Diekmann, et al., "Chip-based wide field-of-view nanoscopy," Nature Photonics 11, 322-328 (2017). https://www.nature.com/nphoton/journal/v11/n5/abs/nphoton.2017.55.html

right of the ALICA Monitor window and take the y-value of the curve as the approximate value for the set point. This value is highlighted in the figure below:
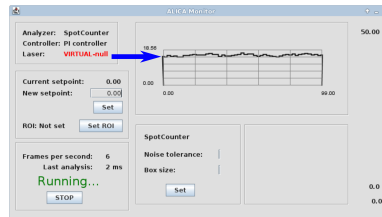
5.2 Software-specific

5.2.1 What version of Micro-Manager should I use?

ALICA was designed to work with Micro-Manager 2.0 or greater. See the Micro-Manager 2.0 website for more information.

5.2.2 Why doesn't ALICA work properly when SASS is installed?

SASS is a Fiji plugin providing a simulation environment that is used to develop and test ALICA. Because of this, the SASS .jar file contains a completely independent copy of ALICA which competes with Micro-Manager's copy, producing unexpected behavior.

For this reason, we highly recommend installing SASS with an installation of Fiji that is independent of the copy of ImageJ used by Micro-Manager and ALICA.

CHAPTER 6

Javadoc

6.1 ch.epfl.leb.alica

6.1.1 AlicaCore

public final class **AlicaCore**

The core's settings are controlled by MainGUI, and the Core then produces products from its factories, and initializes the Coordinator, and later terminates it.

Author stefko

Methods

getAnalyzerFactory

public AnalyzerFactory **getAnalyzerFactory**()

Returns AnalyzerFactory

getControllerFactory

public ControllerFactory **getControllerFactory**()

Returns ControllerFactory

getInstance

public static *AlicaCore* **getInstance**()

Returns the singleton instance, or an exception if it was not yet initialized

Returns the singleton instance of the core

getLaserFactory

public *LaserFactory* **getLaserFactory** ( )

>   **Returns** LaserFactory

initialize

public static *AlicaCore* **initialize** (Studio *studio*)
>   Initialize the Singleton core

>   **Parameters**

>   >   • **studio** – MicroManager studio

>   **Throws**

>   >   • *AlreadyInitializedException* – if it was already initialized

>   **Returns** the singleton instance of the core

isCoordinatorRunning

public boolean **isCoordinatorRunning** ( )
>   Checks if the stop flag of the coordinator was set.

>   **Returns** true if coordinator's stop flag was set, or if coordinator is null

printLoadedDevices

public void **printLoadedDevices** ( )
>   Print all loaded devices in the MMCore to the log.

setControlWorkerTickRate

public void **setControlWorkerTickRate** (int *controller_tick_rate_ms*)
>   Sets the tick rate for the controller.

>   **Parameters**

>   >   • **controller_tick_rate_ms** – delay between two runs of the ControlTask

setCurrentROI

public boolean **setCurrentROI** ( )
>   Sets currently selected ROI to the analyzer when it is initialized

>   **Returns** true if ROI is set, false if no ROI is set

setLaserPowerDeadzone

public void **setLaserPowerDeadzone** (double *laser_power_deadzone*)

Sets the deadzone of change of laser power output. For example, if set to 0.1, the laser would ignore requests for change of power that would be different by less than 10% from current output power.

> **Parameters**
>
> > • **laser_power_deadzone** – deadzone size (NOT in percent)

setLaserVirtual

public void **setLaserVirtual** (boolean *is_laser_virtual*)

Inform factories that the laser should only display its output, not really communicate with the hardware.

> **Parameters**
>
> > • **is_laser_virtual** – true if virtual, false if real

setMaxLaserPower

public void **setMaxLaserPower** (double *max_laser_power*)

Inform factories of maximal laser power value.

> **Parameters**
>
> > • **max_laser_power** – maximal laser power value

startWorkers

public void **startWorkers** (*ImagingMode imaging_mode*)

Builds products from their factories using current settings, and starts the Coordinator (analysis is started)

> **Parameters**
>
> > • **imaging_mode** –

stopWorkers

public void **stopWorkers** ()

Requests the coordinator to stop and then waits for it to join.

6.1.2 AlicaCore.AlreadyInitializedException

public static class **AlreadyInitializedException** extends RuntimeException

Thrown if a double initialization is requested

Constructors

AlreadyInitializedException

public **AlreadyInitializedException** (String *message*)

Parameters

- **message** – exception message

6.1.3 AlicaLogger

public class **AlicaLogger**

The ALICA logger logs analyzer and controller outputs during an acquisition.

Author Marcel Stefko

Methods

addBatchedOutput

public void **addBatchedOutput** (int *frame_no*, double *value*)

Add batched output of analyzer into the log.

Parameters

- **frame_no** –
- **value** – value of the output

addControllerOutput

public void **addControllerOutput** (int *frame_no*, double *value*)

Add output of controller into log

Parameters

- **frame_no** –
- **value** – value of the output

addIntermittentOutput

public void **addIntermittentOutput** (int *frame_no*, double *value*)

Add intermittent output of analyzer into log

Parameters

- **frame_no** –
- **value** – value of the output

addSetpoint

public void **addSetpoint** (int *frame_no*, double *setpoint*)

Add setpoint of controller into log

Parameters

- **frame_no** –
- **setpoint** – value of the output

addToLog

public void **addToLog** (int *frame_no*, String *value_name*, double *value*)

Add a parameter into the log.

> **Parameters**
>
> - **frame_no** – The acquisition frame number for this log entry.
>
> - **value_name** – name of parameter
>
> - **value** – value of parameter

addToLog

public void **addToLog** (int *frame_no*, String *value_name*, String *value*)

Add a parameter into log

> **Parameters**
>
> - **frame_no** –
>
> - **value_name** – name of parameter
>
> - **value** – value of parameter

clear

public final void **clear** ()

Resets logger, removes all data.

getInstance

public static *AlicaLogger* **getInstance** ()

> **Returns** AlicaLogger singleton

getLogMap

public LinkedHashMap<Integer, LinkedHashMap<String, Object>> **getLogMap** ()

Returns the current log.

> **Returns** The current log stored by this logger.

logDebugMessage

public void **logDebugMessage** (String *message*)

Log message to MicroManager or to a general logger

> **Parameters**
>
> - **message** – message to be logged

logError

public void **logError** ([Exception](#) *exc*, [String](#) *message*)

Log exception in MicroManager or in general logger

> **Parameters**
>
> - **exc** – exception to be logged
>
> - **message** – message to be logged

logMessage

public void **logMessage** ([String](#) *message*)

Log message to MicroManager or to a general logger.

> **Parameters**
>
> - **message** – The message to be logged.

saveLog

public boolean **saveLog** ()

Saves the log into a csv file chosen by file selection dialog.

> **Returns** true if save was successful, false otherwise

setStudio

public void **setStudio** (Studio *studio*)

Set studio to allow general logging.

> **Parameters**
>
> - **studio** – MMStudio

showError

public void **showError** ([Exception](#) *exc*, [String](#) *message*)

Show exception in MicroManager or in ImageJ

> **Parameters**
>
> - **exc** – exception to be shown
>
> - **message** – message to be shown

showMessage

public void **showMessage** ([String](#) *message*)

Show message in MicroManager or in ImageJ

> **Parameters**
>
> - **message** – message to be logged

6.1.4 AlicaLoggerTest

public class **AlicaLoggerTest**
>   Unit tests for the ALICA logger class.

>>   **Author** Kyle M. Douglass

Methods

setUp

public void **setUp**()

testAddToLogNoOverwrite

public void **testAddToLogNoOverwrite**()
>   Test of addToLog method, of class AlicaLogger.

testAddToLog_3args_1

public void **testAddToLog_3args_1**()
>   Test of addToLog method, of class AlicaLogger.

testAddToLog_3args_2

public void **testAddToLog_3args_2**()
>   Test of addToLog method, of class AlicaLogger.

6.1.5 AlicaPlugin

public class **AlicaPlugin** implements MenuPlugin, SciJavaPlugin
>   MicroManager2.0 MenuPlugin for automated laser illumination intensity control.

>>   **Author** Marcel Stefko

Methods

getCopyright

public String **getCopyright**()

>>   **Returns** plugin copyright

getCore

public *AlicaCore* **getCore**()

>>   **Returns** singleton core of ALICA plugin

getHelpText

public String **getHelpText**()

getName

public String **getName**()

> **Returns** name of the plugin

getSubMenu

public String **getSubMenu**()

> **Returns** Sub-menu location of the plugin

getVersion

public String **getVersion**()

> **Returns** current plugin version

onPluginSelected

public void **onPluginSelected**()
> Display the MainGUI singleton if it was hidden, if it doesn't exist, initialize it. AlicaCore must be initialized before calling this method.

setContext

public void **setContext**(Studio *studio*)
> Initialize the AlicaCore, if it already exists, do nothing.

> > **Parameters**

> > > • **studio** – MMStudio

6.1.6 ImagingMode

public enum **ImagingMode**
> Possible ways for the plugin to grab images from micromanager.

> > **Author** Marcel Stefko

Enum Constants

GRAB_FROM_CORE

public static final *ImagingMode* **GRAB_FROM_CORE**
> Query directly the MMCore getLastImage() method.

LIVE

public static final *ImagingMode* `LIVE`
> Get images from the Datastore associated with live() mode.

NEXT_ACQUISITION

public static final *ImagingMode* `NEXT_ACQUISITION`
> Get images from the Datastore which is associated with the next acquisition that will be started.

6.1.7 Laser

public interface `Laser`
> Laser recieves input from the controller and adjusts the laser power accordingly.

> > **Author** Marcel Stefko

Methods

getDeviceName

public String `getDeviceName()`

> > **Returns** unique device name (assigned by MicroManager)

getLaserPower

public double `getLaserPower()`
> Asks the hardware for current actual value of laser power

> > **Throws**

> > > • **Exception** – if error occurred during communication with hardware

> > **Returns** actual laser power value

getLaserPowerCached

public double `getLaserPowerCached()`
> Returns cached value of laser power, without querying the hardware for actual value.

> > **Returns** cached laser power value

getMaxPower

public double `getMaxPower()`

> > **Returns** maximal allowed value of laser power

getMinPower

public double **getMinPower** ()

> **Returns** minimal allowed value of laser power

getPropertyName

public String **getPropertyName** ()

> **Returns** unique device property name (assigned by MicroManager)

setLaserPower

public double **setLaserPower** (double *desired_power*)
> Set the laser power to desired value

> **Parameters**

> > • **desired_power** – desired laser power value

> **Throws**

> > • **Exception** – if error occurred during communication with hardware

> **Returns** actual laser power value

6.1.8 MainGUI

public final class **MainGUI** extends JFrame
> Main controlling GUI for the ALICA plugin. This is a singleton which is shown every time the plugin is invoked from MM menu.

> **Author** Marcel Stefko

Methods

getInstance

public static *MainGUI* **getInstance** ()

> **Returns** the GUI singleton instance

initialize

public static *MainGUI* **initialize** (*AlicaCore core*)
> Singleton initializer

> **Parameters**

> > • **core** – AlicaCore singleton

> **Returns** the GUI instance

6.1.9 MainGUI.AlreadyInitializedException

public static class **AlreadyInitializedException** extends RuntimeException
Thrown if the GUI singleton is attempted to be initialized for a second time.

Constructors

AlreadyInitializedException

public **AlreadyInitializedException** (String *message*)

> **Parameters**
>
> > • **message** – message of the exception.

6.2 ch.epfl.leb.alica.lasers

6.2.1 LaserFactory

public final class **LaserFactory**
LaserFactory

> **Author** Marcel Stefko

Constructors

LaserFactory

public **LaserFactory** (Studio *studio*)
Initialize the factory with the MM studio

> **Parameters**
>
> > • **studio** –

Methods

build

public *Laser* **build** ()
Build the laser using the current state

> **Returns** initialized Laser

getPossibleLasers

public StrVector **getPossibleLasers** ()
Query the MMCore for list of loaded devices

> **Returns** StrVector list of loaded devices in the core

getSelectedDeviceName

public String **getSelectedDeviceName**()

> **Returns** currently selected device identifier

getSelectedDeviceProperties

public StrVector **getSelectedDeviceProperties**()
> Query the MMCore for properties of the selected device

> > **Throws**

> > > • **Exception** – if hardware communication fails

> > **Returns** StrVector list of properties

selectDevice

public void **selectDevice**(String *name*)
> Select a device

> > **Parameters**

> > > • **name** – unique device identifier from the MMCore

selectProperty

public void **selectProperty**(String *property*)
> Select a property of the currently selected device

> > **Parameters**

> > > • **property** – unique property identifier from MMCore

setLaserPowerDeadzone

public void **setLaserPowerDeadzone**(double *laser_power_deadzone*)
> Sets the deadzone of change of laser power output. For example, if set to 0.1, the laser would ignore requests for change of power that would be different by less than 10% from current output power.

> > **Parameters**

> > > • **laser_power_deadzone** – deadzone size (NOT in percent)

setLaserVirtual

public void **setLaserVirtual**(boolean *is_laser_virtual*)
> If true, create a VirtualLaser, otherwise a MMLaser

> > **Parameters**

> > > • **is_laser_virtual** –

setMaxLaserPower

public void **setMaxLaserPower** (double *max_laser_power*)

Sets upper boundary for laser output, higher inputs from controller will be constrained.

Parameters

- **max_laser_power** –

6.2.2 MMLaser

public class **MMLaser** implements *Laser*

A MicroManager laser implementation

Author Marcel Stefko

Constructors

MMLaser

public **MMLaser** (Studio *studio*, String *device_name*, String *property_name*, double *min_power*, double *max_power*, double *laser_power_deadzone*)

Initialize the MicroManager laser

Parameters

- **studio** – MMStudio

- **device_name** – MM identifier of the device

- **property_name** – MM identifier of the property to be controlled

- **min_power** – minimal allowed property value

- **max_power** – maximal allowed property value

- **laser_power_deadzone** – deadzone of laser power change requests

Methods

getDeviceName

public String **getDeviceName** ()

getLaserPower

public double **getLaserPower** ()

getLaserPowerCached

public double **getLaserPowerCached** ()

getMaxPower

public double **getMaxPower** ( )

getMinPower

public double **getMinPower** ( )

getPropertyName

public String **getPropertyName** ( )

setLaserPower

public double **setLaserPower** (double *desired_power*)

6.2.3 VirtualLaser

public class **VirtualLaser** implements *Laser*
A virtual laser which does not actually output the values to the laser, only to the GUI and the debug MM log.

> **Author** Marcel Stefko

Constructors

VirtualLaser

public **VirtualLaser** (Studio *studio*, String *device_name*, String *property_name*, double *min_power*, double
max_power)
Initialize the virtual laser

> **Parameters**
>
> - **studio** – MMStudio
>
> - **device_name** – MM identifier of the device
>
> - **property_name** – MM identifier of the property to be controlled
>
> - **min_power** – minimal allowed property value
>
> - **max_power** – maximal allowed property value

Methods

getDeviceName

public String **getDeviceName** ( )

getLaserPower

public double **getLaserPower**()

getLaserPowerCached

public double **getLaserPowerCached**()

getMaxPower

public double **getMaxPower**()

getMinPower

public double **getMinPower**()

getPropertyName

public String **getPropertyName**()

setLaserPower

public double **setLaserPower**(double *desired_power*)

6.3 ch.epfl.leb.alica.workers

6.3.1 AnalysisWorker

public class **AnalysisWorker** extends Thread
>   This thread continuously queries either the MMCore, or the processing pipeline of the live mode for new images, and calls the analyzer's processImage() method on them as fast as it can. Always the latest image is taken for analysis, so it is possible for images to be skipped. It also gathers some statistics for display by the GUI.

>   >   **Author** Marcel Stefko

Constructors

AnalysisWorker

public **AnalysisWorker**(*Coordinator coordinator*, Studio *studio*, Analyzer *analyzer*, *ImagingMode imaging_mode*)
>   Initialize the worker.

>   >   **Parameters**

>   >   >   • **coordinator** – parent Coordinator

>   >   >   • **studio** – for logging and image queries

- **analyzer** – this Analyzer's processImage() method is called on gathered images

- **imaging_mode** –

Methods

acquisitionEnded

public void **acquisitionEnded**(AcquisitionEndedEvent *evt*)

  If the imaging mode is NEXT_ACQUISITION, the coordinator will asked to stop.

    Parameters

      • **evt** – acquisition stopped

acquisitionStarted

public void **acquisitionStarted**(AcquisitionStartedEvent *evt*)

  If the imaging mode is NEXT_ACQUISITION, the NewImageWatcher will be informed.

    Parameters

      • **evt** – new acquisition started event

getAnalyzerShortDescription

public String **getAnalyzerShortDescription**()

  Returns the current description of the analyzer's output.

    Returns A string describing the analyzer's current output.

getCurrentFPS

public int **getCurrentFPS**()

    Returns number of analyzed frames in the last second

getCurrentImageCount

public int **getCurrentImageCount**()

    Returns number of analyzed frames since last counter reset, which could be either caused by live mode start, or acquisition start.

getLastAnalysisTime

public long **getLastAnalysisTime**()

    Returns duration of last analysis in milliseconds

getNewImageFromCoreAndAnalyze

public void **getNewImageFromCoreAndAnalyze**()
> Acquire the new image directly from MMCore and send for analysis.

> > **Throws**
> >
> > > • **java.lang.InterruptedException** – if waiting is interrupted

getNewImageFromWatcherAndAnalyze

public void **getNewImageFromWatcherAndAnalyze**()
> Grabs new images from the Datastore associated with the NewImageWatcher, analyzes it.

> > **Throws**
> >
> > > • **java.lang.InterruptedException** –

liveModeStarted

public void **liveModeStarted**(LiveModeEvent *evt*)
> Called by the MMCore to signalize there is a new live mode. If the imaging mode is LIVE, the NewImage-Watcher will be informed.

> > **Parameters**
> >
> > > • **evt** – new live mode event

queryAnalyzerForBatchOutput

public double **queryAnalyzerForBatchOutput**()
> Analyzer's internal state might change, and the output is passed on to the controller.

> > **Returns** batched output of analyzer

queryAnalyzerForIntermittentOutput

public double **queryAnalyzerForIntermittentOutput**()
> Used for GUI rendering.

> > **Returns** intermittent output of the analyzer

requestStop

public void **requestStop**()
> Stops the analyzer after finalizing the current analysis.

run

public void **run**()

setLastImageCoords

void **setLastImageCoords** (Coords *coords*)
> Called by the NewImageWatcher to update last coords

>> **Parameters**

>>> • **coords** – new Coords

setROI

public void **setROI** (Roi *roi*)
> Set the ROI for Analyzer

>> **Parameters**

>>> • **roi** – ROI to be set

6.3.2 ControlTask

class **ControlTask** extends TimerTask
> This TimerTask is run periodically by the ControlWorker

>> **Author** Marcel Stefko

Constructors

ControlTask

public **ControlTask** (*AnalysisWorker analysis_worker*, Controller *controller*, *Laser laser*)
> Initialize the ControlTask

>> **Parameters**

>>> • **analysis_worker** – AnalysisWorker which will be queried for output

>>> • **controller** – Controller to which output of AnalysisWorker is fed

>>> • **laser** – Laser to which output of Controller is fed

Methods

getLastControllerOutput

public double **getLastControllerOutput** ()
> **Returns** last controller output

run

public void **run** ()

6.3.3 ControlWorker

public class **ControlWorker** extends Timer

> A Timer which schedules a task that regularly queries the AnalysisWorker for batched output, and passes it on to the controller, then gets the controller's output and passes it on to the laser.

> > **Author** Marcel Stefko

Constructors

ControlWorker

public **ControlWorker** (*AnalysisWorker analysis_worker*, Controller *controller*, *Laser laser*)

> Initialize the ControlWorker

> > **Parameters**

> > > - **analysis_worker** – AnalysisWorker which will be queried for output

> > > - **controller** – Controller to which output of AnalysisWorker is fed

> > > - **laser** – Laser to which output of Controller is fed

Methods

getLastControllerOutput

public double **getLastControllerOutput** ()

> > **Returns** last controller output

scheduleExecution

public void **scheduleExecution** (long *delay_ms*, long *period_ms*)

> The task of this worker will be executed regularly.

> > **Parameters**

> > > - **delay_ms** – initial delay

> > > - **period_ms** – period of the task

6.3.4 Coordinator

public class **Coordinator**

> Coordinates workhorses of the analysis.

> > **Author** Marcel Stefko

Constructors

Coordinator

public **Coordinator** (Studio *studio*, Analyzer *analyzer*, Controller *controller*, *Laser laser*, *ImagingMode imaging_mode*, int *controller_tick_rate_ms*, Roi *ROI*, boolean *headless*)
Initialize the coordinator

> **Parameters**
>
> - **studio** – MM studio
> - **analyzer** –
> - **controller** –
> - **laser** –
> - **imaging_mode** –
> - **controller_tick_rate_ms** –
> - **ROI** – roi for analyzer

Methods

dispose

public void **dispose** ()
Clear windows opened by analyzers and controllers.

getAnalyzerStatusPanel

public AnalyzerStatusPanel **getAnalyzerStatusPanel** ()

> **Returns** status panel of associated analyzer

getControllerStatusPanel

public ControllerStatusPanel **getControllerStatusPanel** ()

> **Returns** status panel of associated controller

getTimeMillis

public final long **getTimeMillis** ()
Returns time in milliseconds since the worker was initialized

> **Returns** elapsed time in milliseconds

isRunning

public boolean **isRunning** ( )
> True if still running, false if stopped

> > **Returns** boolean

requestStop

public void **requestStop** ( )
> Request the threads to stop.

setCurrentROI

public boolean **setCurrentROI** ( )
> Get the currently selected ROI in active MM display, and set it as analysis ROI.

> > **Returns** true if ROI has been set, false if no ROI is set

setSetpoint

public void **setSetpoint** (double *value*)
> Set the controller setpoint to value

> > **Parameters**

> > > • **value** – new value of controller setpoint

6.3.5 Grapher

class **Grapher**
> Wrapped around GraphData for easier processing

> > **Author** Marcel Stefko

Constructors

Grapher

public **Grapher** (int *n_points*)
> Initialize a grapher with set length of point plotting

> > **Parameters**

> > > • **n_points** – no. of points to be plotted

Methods

addDataPoint

public void **addDataPoint** (double *value*)
>   Add the next point to the grapher

>   **Parameters**

>   • **value** – value to be added

getGraphData

public GraphData **getGraphData** ( )
>   Return GraphData which can then be plotted

>   **Returns** GraphData

6.3.6 MonitorGUI

public class **MonitorGUI** extends javax.swing.JFrame
>   Display for monitoring the current Coordinator state. This display is controlled by the Coordinator.

>   **Author** Marcel Stefko

Fields

l_fps

public javax.swing.JLabel **l_fps**

l_laser_power_max

public javax.swing.JLabel **l_laser_power_max**

l_last_analysis_duration

public javax.swing.JLabel **l_last_analysis_duration**

p_realtime_plot_parent

public javax.swing.JPanel **p_realtime_plot_parent**

pb_laser_power

public javax.swing.JProgressBar **pb_laser_power**

Constructors

MonitorGUI

public **MonitorGUI** (*Coordinator* *coordinator*, String *analyzer_name*, String *controller_name*, String *laser_name*, String *analyzer_description*, double *start_setpoint*)

Creates new form MonitorGUI

> **Parameters**
>
> - **coordinator** – Coordinator parent
>
> - **analyzer_name** – name of the used analyzer
>
> - **controller_name** – name of the used controller
>
> - **laser_name** – name of the used laser
>
> - **analyzer_description** – A short description of the analyzer's units.
>
> - **start_setpoint** – setpoint value to display at startup

Methods

setLaserPowerDisplayMax

public void **setLaserPowerDisplayMax** (double *value*)

Adjust the displayed laser power maximal value and store it for progressbar calculations.

> **Parameters**
>
> - **value** – max laser power value

setRoiStatus

public void **setRoiStatus** (boolean *is_set*)

Update the GUI display of ROI status

> **Parameters**
>
> - **is_set** – true if ROI is set

setStopped

public void **setStopped** ( )

Displays the STOPPED message in GUI.

updateAnalyzerDescription

public void **updateAnalyzerDescription** (String *description*)

Update analyzer description.

> **Parameters**
>
> - **description** – A short description of the analyzer's outputs.

updateFPS

public void **updateFPS** (int *value*)
>   Update displayed FPS to new value

>> **Parameters**

>>> • **value** – new value of FPS

updateLaserPowerDisplay

public void **updateLaserPowerDisplay** (double *value*)
>   Update displayed laser power to new value

>> **Parameters**

>>> • **value** – new value of laser power

updateLastAnalysisDuration

public void **updateLastAnalysisDuration** (int *duration_ms*)
>   Update last analysis duration to new value

>> **Parameters**

>>> • **duration_ms** – last analysis duration in ms

updatePlot

public void **updatePlot** (GraphData *data*)
>   Update the plow with new data

>> **Parameters**

>>> • **data** – data to be plotted

6.3.7 MonitorTask

class **MonitorTask** extends TimerTask
>   This TimerTask updates GUI with recent information from other workers

>> **Author** Marcel Stefko

Constructors

MonitorTask

public **MonitorTask** (*MonitorGUI gui*, *AnalysisWorker analysis_worker*, *ControlWorker control_worker*)
>   Initialize new task with relevant members

>> **Parameters**

>>> • **gui** – MonitorGUI to be updated

>>> • **analysis_worker** –

- **control_worker** –

Methods

run

public void **run** ( )

6.3.8 MonitorWorker

public class **MonitorWorker** extends Timer
>   Updates GUI with recent information from other workers

>   **Author** Marcel Stefko

Constructors

MonitorWorker

public **MonitorWorker** (*MonitorGUI* *gui*, *AnalysisWorker* *analysis_worker*, *ControlWorker* *control_worker*)
>   Initialize new worker for monitoring

>   **Parameters**

>   - **gui** – MonitorGUI to be updated

>   - **analysis_worker** –

>   - **control_worker** –

Methods

cancel

public void **cancel** ( )

scheduleExecution

public void **scheduleExecution** (long *delay_ms*, long *period_ms*)
>   The task of this worker will be executed regularly.

>   **Parameters**

>   - **delay_ms** – initial delay

>   - **period_ms** – period of the task

6.3.9 NewImageWatcher

class **NewImageWatcher**

> The watcher is subscribed to a Datastore by the AnalysisWorker, and then it informs the AnalysisWorker of any new images in the Datastore.
>
> > **Author** Marcel Stefko

Constructors

NewImageWatcher

public **NewImageWatcher** (Object *object_to_lock*, *AnalysisWorker thread_to_notify*)

Methods

getLatestDatastore

public Datastore **getLatestDatastore** ()

newImageAcquired

public void **newImageAcquired** (DataProviderHasNewImageEvent *evt*)

> Notify the thread that new image is available and send it the coords.
>
> > **Parameters**
> >
> > - **evt** – event containing coords

setLatestDatastore

public void **setLatestDatastore** (Datastore *store*)

> Sets the latest datastore, and registers for its events.
>
> > **Parameters**
> >
> > - **store** –

CHAPTER 7

About
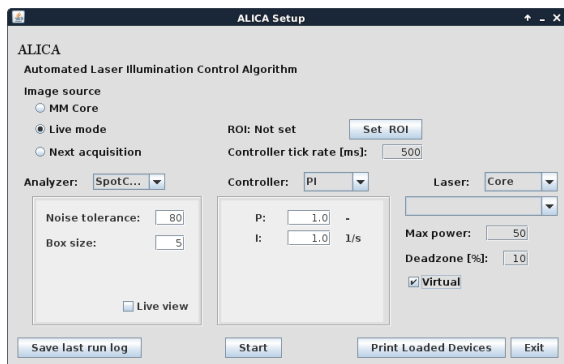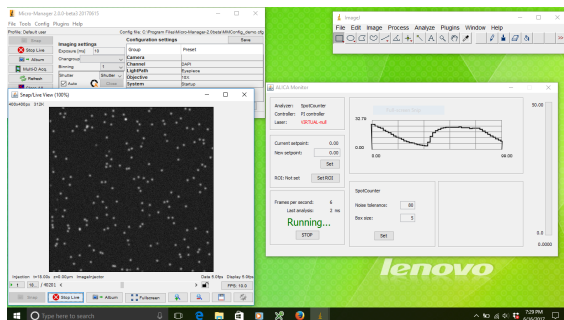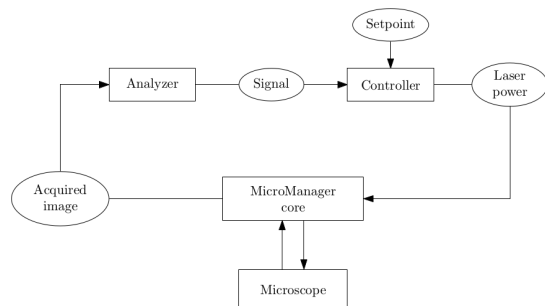
ALICA is an open-source Micro-Manager plugin for real-time control of single molecule photodynamics using adaptive illumination. In particular, **ALICA enables autonomous super-resolution fluorescence imaging using techniques such as STORM and PALM**[1][2][3].

ALICA works by analyzing the incoming images during an acquisition to produce an estimate of the number of fluorescence emitting molecules in a region of interest. The estimates are then fed into a control system that automatically adjusts the illumination intensity to maintain the optimum density of emitting molecules for the desired application. Example applications include

- STORM and PALM super-resolution fluorescence microscopy

- *in vitro* single molecule assays

- single particle tracking

A primary design goal of ALICA is extensibility. A minimal knowledge of the Java programming language will allow you to write your own analyzers for deriving quantitative information from an image stream and controllers for implementing closed-loop feedback for your hardware. As a Micro-Manager plugin, ALICA easily integrates with many types of cameras and illumination sources.

ALICA was designed and written by Marcel Stefko and Kyle M. Douglass in the Laboratory of Experimental Biophysics at the EPFL to automate the lab's STORM and PALM microscopes.

For more information, please see the FAQ or Javadoc

---

[1] M. J. Rust, M. Bates, and X. Zhuang, "Sub-diffraction-limit imaging by stochastic optical reconstruction microscopy (STORM)", Nature Methods 3, 793-796 (2006). http://www.nature.com/nmeth/journal/v3/n10/abs/nmeth929.html

[2] E. Betzig, et al., "Imaging intracellular fluorescent proteins at nanometer resolution," Science 313, 1642-1645 (2006). http://science.sciencemag.org/content/313/5793/1642

[3] S. T. Hess, T. P. K. Girirajan, and M. D. Mason, "Ultra-high resolution imaging by fluorescence photoactivation localization microscopy," Biophysical Journal 91, 4258-4272 (2006). http://www.sciencedirect.com/science/article/pii/S0006349506721403

Acknowledgments

8.1 Thanks

- The Laboratory of Experimental Biophysics and Suliana Manley
- The École Polytechnique Fédérale de Lausanne
- Micro-Manager and the Micro-Manager community
- AutoLase by Seamus Holden and Thomas Pengo
- QuickPALM by Ricardo Henriques, et al.
- SpotCounter by Nico Stuurman

8.2 Authors

- Marcel Stefko
- Kyle M. Douglass

CHAPTER 9

See Also

- SASS - STORM Acquisition Simulation Software

Index